Systematic Review

Edge AI in Practice: A Survey and Deployment Framework for Neural Networks on Embedded Systems

Ruth Cordova-Cardenas [1,2], Daniel Amor [2] and Álvaro Gutiérrez [1,*]

[1] ETSI Telecomunicación, Universidad Politécnica de Madrid, Av. Complutense 30, 28040 Madrid, Spain; rcordova@rbz.es
[2] RBZ Robot Design S.L., C. Casas de Miravete 24A, 28031 Madrid, Spain; damor@rbz.es
* Correspondence: a.gutierrez@upm.es

Abstract

The growing demand for intelligent and autonomous devices has accelerated the integration of neural networks into embedded systems, a paradigm known as Edge AI. While this approach enables real-time, low-latency processing with improved privacy, it remains constrained by strict limitations in memory, computation and energy. This paper presents a systematic review, aligned with PRISMA principles, that examines the current landscape of deep learning deployment on embedded hardware. The review analyzes key optimization techniques—including pruning, quantization and inference-level improvements—together with lightweight architectures such as CNNs, RNNs and compact networks, as well as a diverse ecosystem of hardware platforms and software frameworks. From the recurring patterns identified in the literature, we derive a practical five-stage methodology that guides developers through requirement definition, model selection, optimization, hardware alignment and deployment. Unlike existing surveys that mainly provide descriptive taxonomies, this methodology offers a structured and reproducible workflow explicitly designed to support multi-objective trade-offs in resource-constrained environments. The review also identifies emerging trends such as TinyML and hybrid architectures and highlights persistent gaps, including limited support for ultra-low-precision inference, variability in hardware toolchains and the absence of standardized holistic benchmarking. By synthesizing these insights into a coherent framework, this work aims to facilitate more efficient, robust and scalable Edge AI implementations.

Keywords: neural networks; embedded systems; Edge AI; model compression; inference optimization; TinyML

Academic Editor: Cecilio Angulo

Received: 23 November 2025
Revised: 3 December 2025
Accepted: 5 December 2025
Published: 11 December 2025

Citation: Cordova-Cardenas, R.; Amor, D.; Gutiérrez, Á. Edge AI in Practice: A Survey and Deployment Framework for Neural Networks on Embedded Systems. *Electronics* **2025**, *14*, 4877. https://doi.org/10.3390/electronics14244877

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Inspired by the functioning of the human brain, neural networks stand out for their ability to learn complex patterns and make accurate inferences from data [1]. Their development has enabled major advances in areas such as image recognition, natural language processing and real-time decision-making [2], driving the emergence of increasingly intelligent and autonomous devices across multiple domains. At the same time, the growing demand for on-device intelligence has motivated the search for technologies capable of performing efficient, low-latency inference even in environments with limited compute and memory resources [3]. In this context, embedded systems have become a key platform for deploying Artificial Intelligence (AI), especially in scenarios where connectivity is unreliable or where strict privacy and security constraints preclude reliance on cloud

services [4]. The integration of neural models directly on the device enables intelligent behaviour at the network edge (Edge AI), reducing latency and increasing robustness in safety-critical applications [5].

However, the deployment of Deep Learning (DL) models on embedded hardware introduces significant technical challenges [6]. These systems operate under stringent constraints—including limited memory, restricted computing capability and strict power budgets—that require careful optimization of model architectures and inference pipelines [7]. Although the literature reports a wide range of model compression techniques, hardware-aware optimizations and real-time inference strategies, several open questions remain regarding scalability, energy efficiency and adaptability across heterogeneous platforms [6]. A systematic and critical review is therefore required to assess the current landscape, identify persistent limitations and guide future research toward solutions aligned with the needs of industry and the scientific community.

In this context, the present manuscript provides a detailed analysis of the state of the art in the deployment of neural models on embedded systems. The review focuses primarily on studies published within the last five years, ensuring alignment with the most recent developments in lightweight architectures, optimization strategies and dedicated acceleration hardware. Beyond synthesizing existing techniques, the review culminates in a practical five-stage methodology that operationalizes the findings into an actionable workflow for optimizing and deploying DL models under resource constraints. This structured pipeline differentiates the present review from prior surveys, which predominantly provide taxonomies without offering an integrated, end-to-end design framework.

2. Literature Search Methodology

This systematic review was conducted in accordance with the "Preferred Reporting Items for Systematic Reviews and Meta-Analyses" (PRISMA) 2020 guidelines [8]. The literature identification, screening, eligibility, and inclusion processes were structured to ensure methodological transparency and reproducibility. A PRISMA flow diagram is included as Figure 1 to visually summarize the selection process, and the completed PRISMA checklist is provided in the Supplementary Materials. Additionally, the protocol for this review was prospectively registered in the Open Science Framework (OSF) under the DOI: 10.17605/OSF.IO/GNRH4.

A systematic literature search was carried out across major academic databases and repositories, including IEEE Xplore, ACM Digital Library, MDPI, SpringerLink, ScienceDirect, and arXiv. The search covered publications up to the year 2025 to ensure technological relevance and currency. The query was applied to the title, abstract, and keyword fields.

The search strategy was implemented using Boolean logic and applied to the title, abstract, and keywords fields in each database. Two thematic blocks were constructed and combined with the AND operator:

- Block A (core concepts): "Edge AI" OR "embedded deep learning" OR "TinyML".
- Block B (techniques and deployment): "neural network optimization" OR "quantization" OR "pruning" OR "compact networks" OR "NPU acceleration" OR "Edge TPU" OR "FPGA inference" OR "YOLO optimization" OR "model compression" OR "hardware-software co-design".

The complete Boolean query was:

("Edge AI" OR "embedded deep learning" OR "TinyML") AND ("neural network optimization" OR "quantization" OR "pruning" OR "compact networks" OR "NPU acceleration" OR "Edge TPU" OR "FPGA inference" OR "YOLO optimization" OR "model compression" OR "hardware-software co-design").

To increase specificity, the following terms were excluded from titles and abstracts using the NOT operator:

"cloud computing" OR "cloud inference" OR "data center"

The inclusion criteria considered studies that proposed model optimization techniques (e.g., pruning, quantization, distillation, NAS), efficient network architectures (CNNs, RNNs, Transformers) designed for embedded systems, experimental validation on edge hardware platforms (NPUs, FPGAs, MCUs, Edge TPUs), and reporting of key performance metrics such as inference latency, energy consumption, memory usage, and frames per second (FPS). Exclusion criteria removed theoretical or review works lacking quantifiable experimental results, studies based solely on simulation without practical deployment, and articles missing complete methodology or results sections. In cases where both a preliminary conference version and an extended journal version of the same study were available, the journal version was prioritized due to its greater methodological detail.

The initial search yielded 1020 records. After removing 70 duplicates, 950 titles and abstracts were screened, leading to the exclusion of 710 studies that did not meet the predefined eligibility criteria. Full-text evaluation was performed on 240 articles, with 180 subsequently excluded for non-compliance with the inclusion standards. A total of 60 studies were included in this review. The complete selection process is illustrated in Figure 1, following the PRISMA framework. Risk of bias and certainty of evidence were not formally assessed, as this review focuses on technical evaluations of embedded AI models and systems, rather than clinical or behavioral outcomes.

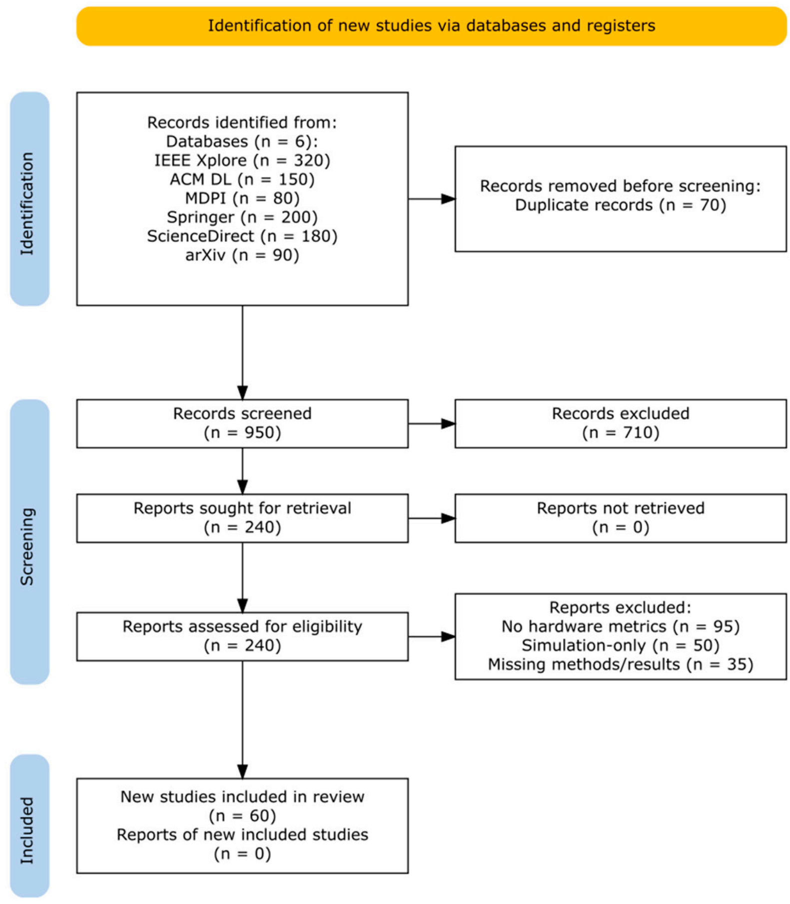

Figure 1. PRISMA-style flow diagram.

3. Core Concepts and Metrics

3.1. The Inference Process

Inference is the process of using a pre-trained neural network to make predictions on new data [9]. In embedded systems, optimizing inference is crucial to achieve low power consumption, high processing speed and efficient memory usage. The inference process (see Figure 2) generally includes the following stages [10]:

- Pre-processing: Input data are transformed into a format compatible with the neural network. This may involve data type conversion, value normalization, or encoding of categorical variables, including the creation of tensors or arrays with the appropriate structure.
- Model Execution: Pre-processed data are fed into the neural network, where information flows through the network layers, performing mathematical operations and non-linear transformations.
- Post-processing: The neural network output, which may be a vector of numbers or an internal representation, is transformed into a format useful for the specific application or task. This may involve conversion to classes, probabilities, numerical values, or other required representations.

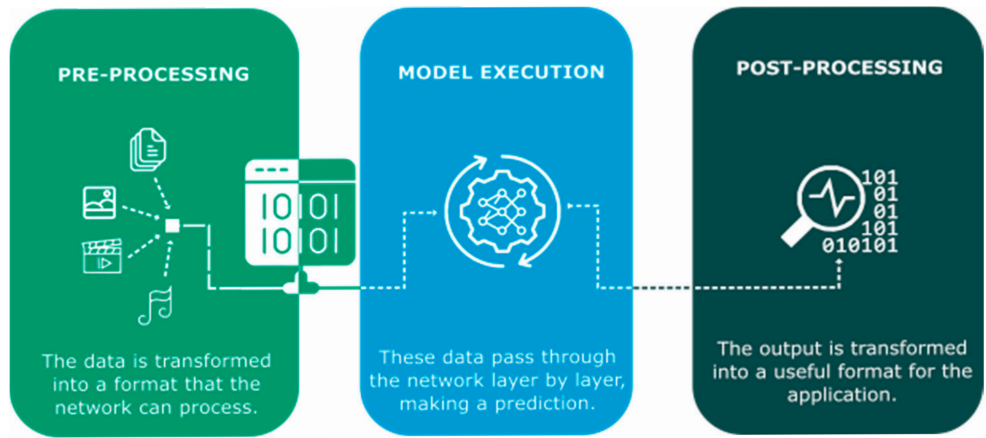

Figure 2. Inference Process.

3.2. Inference Evaluation Metrics

Evaluating the efficiency of inference on embedded systems requires a specific set of metrics that go beyond model accuracy. While task-specific accuracy metrics (such as Mean Average Precision (mAP) for detection, Rank-N accuracy for identification, or Word Error Rate (WER) for speech recognition) remain crucial for assessing functional correctness, the constraints of edge deployment necessitate additional metrics focused on performance and resource utilization.

3.2.1. Model Performance Metrics

- Latency: This refers to the time it takes for the model to process a single input, typically measured in milliseconds (ms) or microseconds (µs). Lower latency indicates a faster response in real-time scenarios. However, latency alone does not fully capture the overall processing capacity of the system; rather, it primarily reflects how quickly each individual request can be handled [10].
- Throughput: The number of inputs that the model can process per unit of time. It is measured in Inferences Per Second (IPS) or Frames Per Second (FPS). Higher throughput means that the model can process more inputs in the same amount of time.

- Energy Consumption: Energy consumption is the amount of energy that the model consumes during inference. It is measured in Watts (W) or milliWatts (mW). Lower power consumption means that the model is more energy-efficient.
- Memory Usage: Memory usage is the amount of RAM that the model uses during inference. It is measured in Kilobytes (KB) or Megabytes (MB). Lower memory usage allows larger models to run on devices with limited memory.

3.2.2. Hardware Accelerator Metrics

In addition to model performance metrics, specific metrics are used to evaluate the efficiency of hardware accelerators in neural network inference:

- Tera Operations Per Second (TOPS): This represents the theoretical maximum throughput, measured in trillions of operations per second. While a higher value suggests greater computing capacity, it does not necessarily reflect real-world performance, as it depends on implementation details and actual workload distribution.
- Tera Operations Per Second per Watt (TOPS/W): This metric measures the energy efficiency of the accelerator, indicating the number of operations performed per watt consumed. Although a higher value suggests better energy efficiency, it may not accurately represent performance if the workload is low or highly variable.
- Tera Operations Per Second per Watt per Megahertz (TOPS/W/MHz): This relates energy efficiency to clock frequency, facilitating comparisons between accelerators operating at different speeds. However, not all architectures scale linearly with frequency, making this metric less universally applicable.
- Giga Operations Per Second per Watt (GOPS/W): Similar to TOPS/W but expressed in giga-operations, this metric is commonly used for lower-power devices, such as mobile platforms and embedded systems.
- Floating-Point Operations Per Second (FLOPS): This measures the number of floating-point operations a hardware accelerator can perform per second. Unlike GOPS and TOPS, which account for both integer and floating-point operations, FLOPS specifically quantifies floating-point performance, making it crucial for applications requiring high numerical precision.

These metrics enable comparisons across different accelerators; however, the peak values reported are often obtained under highly specific conditions and may not accurately reflect real-world performance across all applications [11]. ResNet-50 is frequently used as a standard benchmark to showcase peak values, but relying solely on this model can provide a limited perspective, as it does not necessarily represent more complex tasks or significantly different architectures, such as those used in natural language processing [12].

In [13], M^3ViT has been proposed as a more efficient alternative to ResNet-50 for edge devices, reducing computational cost and energy consumption while maintaining accuracy, measured by Mean Intersection Over Union (mIoU) (see Table 1).

Table 1. Benchmark Comparison: ResNet-50 vs. M^3ViT.

Metric	ResNet-50	M^3ViT
Accuracy (mIoU)	44.2%	45.6%
FLOPs (inference)	192 G	100 G
Energy (W·s)	2.145	0.845

The results indicate that M^3ViT achieves comparable accuracy to ResNet-50 while reducing inference FLOPs by 48% and significantly enhancing energy efficiency on FPGA hardware. Therefore, while these metrics provide valuable insights, it is crucial to consider

them alongside other factors such as latency, power consumption, memory usage and scalability. Additionally, a single benchmark, such as the one presented, can lead to narrow conclusions or result in accelerators that are over-optimized for a specific scenario and less adaptable to other applications. A comprehensive evaluation of multiple architectures and use cases provides a more accurate assessment of the performance and versatility of an accelerator, enabling more robust and efficient designs across a wide range of tasks [14].

To illustrate these metrics in a practical, real-world scenario, Table 2 presents a subset of benchmark results from a recent competitive analysis conducted by [15]. The study evaluated various AI accelerators on a demanding multi-stream video inference task, simulating a 14-camera security system processing 1080p video streams in real time. The data highlight the trade-offs between different hardware platforms when running a modern YOLOv8s object detection model. While the report focuses on system-level throughput (FPS) and energy efficiency (Joules/Frame), it provides a clear example of how these key metrics are used to compare the practical performance of different Edge AI solutions.

Table 2. Comparative Performance Metrics for YOLOv8s on a 14-Stream Video Task [15].

Hardware Platform	Peak Compute	Task	Throughput (FPS)	Energy Efficiency (Joules/Frame)
Intel Core i7-13700K (CPU Only)	1.1 TFLOPS (FP32)	YOLOv8s	11.74	29.983
NVIDIA GeForce RTX 3060	12.7 TFLOPS (FP32)	YOLOv8s	154.30	2.301
Hailo-8 M.2	26 TOPS (INT8)	YOLOv8s	121.50	1.276
Axelera Metis AIPU	214 TOPS (INT8)	YOLOv8s	178.40	1.087

The results in the table underscore several key concepts discussed in this review. Firstly, they demonstrate the massive performance gap between general-purpose Central Processing Unit (CPUs) and specialized hardware, with even the consumer-grade Graphics Processing Unit (GPU) offering over 13 times the throughput of the CPU-only baseline. Secondly, the data reveal significant differences in energy efficiency. While the dedicated M.2 accelerators and the GPU have comparable throughput on this task, the Axelera and Hailo devices consume approximately half the energy per frame compared to the GPU. This highlights a critical trade-off for developers: a general-purpose GPU may offer high performance, but a dedicated accelerator often provides a superior balance of performance and power efficiency, which is crucial for deployments at the edge with strict power budgets. Such real-world data are essential for making informed decisions during the hardware selection phase of any Edge AI project.

3.2.3. System-Level Metrics

In addition to model-level performance and hardware-level efficiency, embedded AI deployments must also account for system-level metrics. These include end-to-end latency, runtime memory footprint, throughput (e.g., FPS), workload distribution across heterogeneous compute units, and thermal stability during sustained operation. System-level metrics capture the interaction between the model, the runtime environment and the scheduling behavior of embedded hardware, providing a holistic view of deployment performance. This provides a foundation for evaluating real-world feasibility, especially in systems limited by power and memory budgets or designed for continuous operation.

3.3. The Role of Standardized Benchmarking in Edge AI

The expansion of hardware and software in the Machine Learning (ML) inference landscape—with over 100 organizations developing inference chips and dozens of software frameworks—makes performance benchmarking a nearly intractable task without a

standardized framework. To address this challenge, benchmarks have been developed that establish a neutral, representative, and reproducible evaluation methodology.

Specific Benchmarks for the Edge: ML Perf Tiny and Edge AIBench

The MLPerf consortium has established the industry standard for benchmarking ML systems. Its MLPerf Inference suite is designed to be architecturally neutral and simulates realistic use cases across four evaluation scenarios: Single-stream, Multi-stream, Server, and Offline. A fundamental principle of MLPerf is the requirement for robust quality targets: performance cannot be achieved at the expense of accuracy. It therefore requires that nearly all implementations achieve a quality within 1% of the FP32 reference model's accuracy.

For the ultra-low-power systems domain, MLPerf Tiny was developed [16], the first standard benchmark for TinyML. It evaluates the inherent trade-offs in these systems by measuring three fundamental metrics: accuracy, latency, and energy. Its suite includes four reference tasks: keyword spotting, visual wake words, image classification, and anomaly detection.

In parallel, other benchmarking efforts have emerged. EdgeAIBench focuses on evaluating the performance of the complete computation chain (device–edge–cloud), while academic benchmarks like DeepEdgeBench [17] conduct comparative evaluations of DNN performance on a variety of commercial edge devices.

3.4. Empirical Analysis of Performance Metrics on Edge Platforms

To demonstrate how inference metrics (latency, throughput, energy, and memory) are applied in practice, it is essential to analyze the results of empirical studies that compare real-world edge hardware. Evaluations like DeepEdgeBench and other comparative analyses examine popular platforms such as the Google Coral Dev Board (equipped with an Edge TPU accelerator) and the Nvidia Jetson Nano (equipped with a GPU). Table 3 synthesizes the results of these comparisons, focusing on models from the MobileNet family, which are representative of edge vision tasks [17].

Table 3. Comparison of Inference Metrics on Edge Platforms.

Metric	Google Coral Dev Board (Edge TPU)	Nvidia Jetson Nano (GPU)
Throughput	~240.5 FPS (MobileNetV2 Quant.) 417 FPS (MobileNet V1)	~48.5 FPS (MobileNetV2 Non-Quant.) 80 FPS (MobileNet V1)
Latency (approx)	~4 ms (MobileNetV2 Quant.)	~20 ms (MobileNetV2 Non-Quant.)
Energy Consumption (Active)	1.543 watt-minutes (MobileNetV2) 5.5 Watts (average execution)	13.630 watt-minutes (MobileNetV2) ~6.05 Watts (10% more than Coral)
Energy Consumption (Idle)	2.757 Watts (LAN off) 4.8 Watts	0.903 Watts (LAN off) ~2.4 Watts (Less than half of Coral)
Memory Usage (DRAM)	42–131 MB	~1.2 GB

- Performance and Latency: The Google Coral Dev Board offers much better throughput and latency for quantized models. This is due to its specialized hardware, such as its 8 MB SRAM cache and its systolic array architecture, which is optimized for the matrix operations of neural networks and reduces memory access.
- Energy Consumption: Edge TPU is significantly more energy-efficient during active inference, consuming much less power to complete the same task. However, the Jetson Nano demonstrates much lower idle consumption. This introduces a key trade-off: Coral is superior for continuous AI computation, whereas the Jetson might be more efficient for sporadic tasks where the device spends most of its time in an idle state.
- Memory Usage: The difference in memory usage is notable. Edge TPU, thanks to its architecture and the use of quantized models, operates using only a fraction of the

RAM (42–131 MB), while Jetson Nano's GPU requires a substantially larger amount (approx. 1.2 GB) to manage the models.

4. State of the Art: Model and Inference Optimization

The successful deployment of neural networks on embedded devices is contingent on two core pillars: advanced optimization techniques that shrink models to fit resource and constraints and the selection of neural network architectures that are inherently efficient.

The convergence of neural networks and embedded systems has driven research into optimization techniques for Artificial Intelligence (AI) models to enable their deployment in resource-constrained environments. These techniques aim to improve model performance and efficiency while addressing inherent integration challenges, such as resource limitations [18]. One of the primary objectives of this research is to design a methodology for prioritizing and reducing parameters in neural architectures. This section explores compression strategies, inference optimization, software frameworks and associated technical challenges. Optimization techniques are essential for reducing a model's computational footprint without significantly degrading its performance.

4.1. Model Compression Techniques

Model compression is fundamental for the efficient deployment of neural networks in resource-constrained embedded systems. These techniques enable the substantial reduction in model size, the improvement of inference speed and the minimization of memory consumption. Core techniques in this domain include pruning and quantization.

While pruning and quantization (detailed in Table 4) are often applied in isolation, cutting-edge research is pivoting toward hybrid techniques and Neural Architecture Search (NAS) to achieve optimal compression ratios.

These advanced methodologies move beyond treating compression as isolated sequential steps. Instead, they co-optimize the architecture, pruning scheme and quantization strategy simultaneously, frequently adopting a hardware-software co-design paradigm.

Table 4. Description of model compression technique.

Compression Technique	Key Techniques	Description	Advantages	Limitations
Pruning	Unstructured (magnitude) & structured pruning	Removes less important weights or neurons from the network.	Reduces model size and computational complexity.	Requires retraining to mitigate accuracy loss. Determining what to prune can be challenging.
Quantization	Standard (e.g., 8-bit)	Reduces the numerical precision of weights and activations (e.g., 8 bits instead of 32 bits).	Reduces model size and speeds up inference.	May affect model accuracy if quantization is too aggressive or introduces rounding errors.
	Sub-4-bit (e.g., 4, 3 and 2-bits)	Reduces precision to fewer than 4 bits, requiring advanced training to manage instability and accuracy loss.	Significant reduction in model size and memory footprint; enables ultra-efficient hardware acceleration.	Highly susceptible to accuracy degradation; requires complex and computationally expensive retraining (QAT).

Table 4. *Cont.*

Compression Technique	Key Techniques	Description	Advantages	Limitations
Quantization	Non-Uniform	Uses non-linear quantization steps that match the natural distribution of weights (often clustered near zero).	Preserves accuracy better than uniform quantization for the same bit-width by assigning more precision to common values.	Can be more complex to implement in hardware; not all inference engines offer native support.
	Binarization (1-bit)	Constrains weights to two values ({−1, +1}). To mitigate accuracy loss, this is combined with advanced training strategies like Ternary Weight Splitting (initializing the binary network from a pre-trained ternary model), Progressive Quantization (gradually increasing the level of quantization), or distillation (training the binary network to mimic a larger, more accurate model).	Reduces memory usage by up to 32x and replaces multiplications with logical operations (XNOR), boosting computational efficiency.	Significant accuracy loss and complex training if the mitigation strategies mentioned in the description are not used.
	Ternarization (2-bit)	Represents weights with three values ({−1, 0, +1}), striking a balance between full-precision fidelity and binarization.	Significantly decreases model size and inference cost.	Preserving accuracy may require distillation or specialized retraining, especially in NLP tasks.
Knowledge distillation	Output-Based Knowledge Distillation (OBKD) & Feature Imitation Knowledge Distillation (FIKD)	Trains a smaller model using knowledge from a larger pretrained model.	Improves performance in smaller models while reducing size.	Requires a pretrained larger model and additional computational resources for training.
Tensor factorization	Singular Value Decomposition (SVD)	Decomposes weight tensors into smaller tensors using techniques.	Reduces parameters and computational requirements for inference.	Computationally expensive; not suitable for all models.
Hashing	Parameter hashing (HashedNet)	Groups similar data to reduce redundancy.	Saves memory and accelerates model inference.	May trade off compression efficiency with retrieval accuracy. Sensitivity to hyperparameters.
Optimization Techniques	Hybrid Techniques and Automated Search (NAS)	Quantization-Aware Pruning (QAP). QAT + Dynamic/Static Pruning (QADS). Architecture, Pruning and Quantization Search (APQ/NAS)	Jointly optimizes pruning and quantization (QAP, QADS), or even the architecture (APQ), in a single training process.	Achieves much higher compression rates and better accuracy than the sequential application of techniques. Optimizes the model for specific hardware metrics (latency, BOPs).

4.1.1. Pruning

Pruning reduces a model's size and computational complexity by removing redundant or less important parameters, such as weights with the smallest magnitude or entire neurons. While it can be implemented during or after training, it often requires a fine-tuning or retraining step to recover any accuracy lost during the pruning process [19].

4.1.2. Quantization

Quantization is one of the most effective compression strategies, reducing the numerical precision of a model's weights and, in some cases, its activations. The standard approach involves converting 32-bit floating-point numbers (FP32) to 8-bit integers (INT8), which significantly cuts down on model size and can accelerate inference speeds on compatible hardware [20]. While 8-bit quantization is widely adopted, more aggressive techniques are crucial for ultra-constrained devices.

- Binarization and Ternarization: These are the most aggressive forms, reducing weights to just one bit ($\{-1, +1\}$) or two bits ($\{-1, 0, +1\}$), respectively. These techniques offer the highest level of compression but require specialized training strategies, often involving knowledge distillation, to mitigate a significant loss of accuracy, as detailed in Table 4.
- Sub-4-bit Quantization (4, 3 and 2 bits): This presents a promising frontier for efficiency but introduces severe challenges. The drastic precision reduction often leads to a considerable degradation in model performance, a problem especially acute in already optimized architectures like MobileNet, which have less inherent redundancy. A key cause for this is the Activation Instability Induced by Weight Quantization (AIWQ), where training becomes unstable and fails to converge because small weight updates cause large, destabilizing oscillations in the quantized output activations.

To overcome these issues, Quantization-Aware Training (QAT) is indispensable. Unlike Post-Training Quantization (PTQ), QAT simulates the low-precision behavior during the training loop, allowing the model to learn weights that are robust to quantization. Advanced QAT methods like PROFIT use techniques such as progressive freezing to stabilize training, successfully quantizing MobileNet models to 4 bits with minimal accuracy loss. To reach even lower precisions like 3 and 2 bits, state-of-the-art frameworks often combine QAT with Knowledge Distillation (KD). A prominent example is the BitDistiller framework, which employs a "teacher–student" distillation strategy to train models successfully at these ultra-low precisions. These advanced methods also leverage Non-Uniform Quantization. Unlike standard linear quantization with evenly spaced steps, non-uniform approaches align better with the natural distribution of neural network weights—which are often concentrated near zero—by assigning more precision levels to that region, thus preserving information more faithfully. The combination of these techniques enables mixed-precision strategies for an optimal balance between efficiency and performance.

4.1.3. Knowledge Distillation

In knowledge distillation, a smaller "student" model is trained to mimic the behavior and outputs of a larger, pre-trained "teacher" model. This process transfers the "knowledge" of the teacher, allowing the student to achieve a performance that would be difficult to reach by training on the data alone. As noted previously, this technique is also a critical component in enabling extreme quantization [21].

4.1.4. Tensor Factorization

This technique decomposes weight tensors into smaller tensors, for instance, by using Singular Value Decomposition (SVD) to decompose weight tensors. It reduces the number

of parameters and operations required for inference. However, it can be implemented using linear algebra libraries such as Eigen or BLAS (Basic Linear Algebra Subprograms). It can be computationally expensive and not all models are suitable for tensor factorization [22].

4.1.5. Hashing

Hashing-based techniques group model parameters into buckets, forcing all parameters within the same bucket to share a single value. This technique, notably used in HashedNet, effectively reduces the memory required to store the model's weights. However, it introduces a trade-off between compression efficiency and potential accuracy loss, as hash collisions can cause unrelated parameters to be assigned the same value [23].

4.1.6. Hybrid Techniques and Automated Search

Although the aforementioned techniques are powerful, the most advanced methods combine them to solve complex optimization problems and achieve superior efficiency.

1. Quantization-Aware Pruning (QAP): This hybrid method integrates pruning and Quantization-Aware Training (QAT) into a single process. Instead of pruning a model and then quantizing it (which can aggravate errors), QAP trains the model to be simultaneously sparse (pruned) and robust to low precision. The objective is for both techniques to be complementary. This strategy has proven to be highly efficient, achieving drastic reductions in the BOPs (Bit Operations) computational complexity metric. In [24], it achieved a 50-fold reduction in BOPs on a 6-bit model pruned to 80%, while maintaining the same accuracy as the original 32-bit model.
2. Dynamic/Static Pruning with QAT (QADS): A key challenge in combining QAT with dynamic pruning methods (where weights can regrow) is instability during training [25]. This instability is attributed to the effect of a "double approximation" of the gradient, as both QAT and dynamic pruning rely on the same Straight-Through Estimator (STE) for backpropagation. To address this, the QADS method [26] utilizes an intelligent alternation strategy. Initially, it employs dynamic pruning to explore and determine the optimal sparse structure. Subsequently, once a target sparsity rate is reached, it transitions to static pruning. By applying a fixed mask, static pruning eliminates the need to approximate the gradient with STE in that phase [24]. This approach ensures stable training and the achievement of high accuracy.
3. Neural Architecture Search (NAS) for Compression: This is the most advanced approach and addresses the problem that the best architecture for a full-precision model is not necessarily the best architecture once compressed [27]. Methods like Joint Search for Network Architecture, Pruning and Quantization Policy (APQ) and Neural Architecture Search for Bert (NAS-BERT) perform a joint search for the architecture, pruning, and quantization policy.

- APQ utilizes a Once-For-All (OFA) network and an accuracy predictor trained with knowledge transfer (Predictor–Transfer) to estimate the performance of a sub-network without the cost of a full retraining. This drastically reduces the search cost and achieves superior accuracy under the same latency constraints [21].
- NAS-BERT applies Neural Architecture Search (NAS) to the compression of language models by training a supernet [27]. To handle the massive search space (~10^{34} architectures), the method employs techniques such as Block-wise Search and Progressive Shrinking. These techniques succeed in reducing the search space to a manageable size (~10^{20}).

4.1.7. Comparative Summary of Compression Techniques Across Studies

After examining pruning, quantization, distillation, tensor factorization, hashing and hybrid optimization approaches in Sections 4.1.1–4.1.6, this subsection provides an integrated comparison of the main techniques reported in the literature. Table 5 consolidates these methods by summarizing their compression ratios, accuracy impact and hardware compatibility. Three consistent patterns emerge from this overview: (i) post-training techniques such as pruning or INT8 quantization typically achieve moderate compression with limited accuracy degradation; (ii) joint optimization strategies—including pruning combined with quantization or distillation—offer substantially higher reductions while preserving accuracy; and (iii) emerging approaches such as NAS-based compression and mixed-precision inference provide superior efficiency but depend on more complex training procedures and stricter hardware support.

Table 5. Comparative Summary.

Technique	Reported Compression Ratio	Accuracy Impact	Hardware Compatibility Mentioned
Structured/Unstructured Pruning	10–80% parameter reduction (various CNNs)	Low to moderate drop; requires fine-tuning	CPU, GPU, FPGA, ASIC
Knowledge Distillation (ResNet, WRN, ResNeXt)	5–16% FLOPs and parameter reduction	Often improves accuracy vs. baseline	CPU, GPU
Extreme Distillation (BUnit-Net)	Up to 97% FLOPs and 96% parameter reduction	Slight accuracy drop depending on the task	MCU, low-power accelerators
INT8 Quantization	4× model-size reduction	~1–3% drop (model-dependent)	Edge TPU, NPU, GPU, CPU
Sub-4-bit Quantization (4/3/2-bit)	8–16× reduction	Significant drop unless QAT + KD used	Specialized NPUs, FPGA
Binarization/Ternarization (1–2 bits)	16–32× reduction	High accuracy drops unless distilled	ASIC, FPGA
Non-Uniform Quantization	4–16× effective compression	Lower accuracy loss than uniform	NPUs with LUT support
Tensor Factorization (SVD, CP)	Moderate reduction in FLOPs/params	Minimal accuracy loss	CPU, GPU
Hashing (HashedNet)	Large memory reduction	Accuracy varies with collision rate	CPU, GPU
Joint Pruning + Quantization (QAP, QADS)	Up to 50× BOP reduction; 80% sparsity	Comparable to FP32 in the best cases	NPUs, FPGA, CPU
NAS-assisted Compression (APQ, NAS-BERT)	Superior compression–accuracy ratio via search	Often matches or exceeds FP32	NPU, GPU, CPU
Mixed Precision (4/8-bit)	4–8× model-size reduction	Minimal loss with QAT + KD	Edge TPU, NPU, GPU

Although NAS-based compression, joint pruning–quantization strategies, and mixed-precision inference represent some of the most advanced techniques, their adoption in embedded environments remains limited by practical constraints such as hardware-specific bit-width support, memory-bandwidth restrictions, and heterogeneous toolchains across NPUs, Edge-TPUs, and microcontrollers. These limitations underscore a persistent gap between state-of-the-art compression algorithms and the capabilities of real embedded platforms, emphasizing the need for more systematic and hardware-aware methodologies.

4.2. Inference Optimization Techniques

Various techniques can be applied to optimize neural network inference in embedded systems, classified based on their approach:

4.2.1. Hardware Acceleration

- Dedicated Accelerators: They offer high performance and energy efficiency but can be costly and inflexible.
- Heterogeneous Computing: It utilizes different processing architectures (CPU, GPU, FPGA) to optimize application performance by distributing workloads and leveraging each architecture's strengths [10].
- ISA Extensions (Instruction Set Architecture): These are special instructions added to a processor's architecture to accelerate common neural network operations. They are less complex to implement and more flexible but may have limited performance. Modern ARM processors include ISA extensions, achieving a 74% reduction in clock cycles for OCR tasks [28].

4.2.2. Software-Level Optimization

- Hardware-Specific Compilation: This adapts deep learning models for efficient execution on specific hardware using tools like TensorFlow Lite Converter and XLA (Accelerated Linear Algebra) [29].
- Memory Optimization: This includes techniques to efficiently manage memory during inference, such as quantization, memory compression and buffer reuse.
- Attention Mechanism Optimization: This enhances the efficiency of attention mechanisms in terms of memory and speed. Techniques like FlashAttention [30] and PagedAttention [31] have been developed to reduce the memory usage and execution time of the attention mechanism.

4.2.3. Emerging Techniques

- Analog In-Memory Computing (AIMC): This offers high energy efficiency by performing calculations directly in memory but has precision limitations [32].
- Processing-In-Memory (PIM): PIM architectures integrate processing and memory on the same chip, reducing the need to transfer data between memory and processor, significantly improving energy efficiency and latency [4].

Table 6 presents a summary of the advantages and disadvantages of different inference optimization approaches.

Table 6. Comparison of Various Inference Optimization Approaches.

Technique/Approach	Advantages	Disadvantages/Limitations	Applications/Examples
Hardware Acceleration	High performance, energy efficiency	Costly, less flexible	Computer vision in IoT devices, real-time object detection, autonomous driving, and NLP on mobile devices.
Software-Level Optimization	Greater efficiency, lower latency, optimized resource usage	Requires specific tools and knowledge may affect accuracy	Efficient implementation on mobile devices, memory optimization in NPUs, and attention mechanism acceleration on GPUs.
Emerging Techniques	High energy efficiency, potential for low latency	Precision limitations, developing technology	Smart sensors, high-performance applications.

5. State of the Art: Architectures, Platforms and Frameworks

5.1. Neural Network Architectures for Embedded Systems

Choosing an efficient neural network architecture is critical to ensure its successful implementation in embedded systems. These architectures must be designed to operate under constraints of memory, processing capacity and energy consumption, maintaining a balance between accuracy and efficiency [33].

Hardware limitations in embedded systems necessitate the optimization of both neural network architecture and training and inference processes. Limited storage capacity demands lightweight and compact models, while the requirement for fast response times necessitates a low number of operations. Additionally, energy efficiency is crucial for extending battery life and enabling deployment in power-constrained environments.

Neural networks are applied in various areas within embedded systems. Diverse architectures exist, each optimized for specific tasks such as computer vision, audio processing, natural language processing and time series prediction, among others, with specific applications in embedded systems [2]. The following section provides a description of some of the most used architectures in embedded systems.

Selecting the right architecture is as important as optimization. Modern architectures are often designed with efficiency in mind.

5.1.1. Convolutional Neural Networks (CNNs)

CNNs are widely used in embedded systems for computer vision tasks. They are capable of extracting relevant image features through the use of convolutional and pooling layers. These networks are ideal for applications such as classification, object detection and segmentation. However, traditional CNNs are often too complex for resource-constrained devices, which has driven the development of lighter and more efficient architectures designed specifically for embedded systems [34].

(a) Main Architectures and Optimized Versions

The most widely used architecture within this group is YOLO (You Only Look Once), a real-time object detection algorithm known for its efficiency and speed. From YOLOv1, which introduced single-pass detection, to YOLOv11, each version has improved in accuracy and speed, incorporating additional functions such as classification and segmentation [35]. For devices with limited resources, "tiny" versions of YOLO have been developed (such as YOLOv3-tiny and YOLOv4-tiny), which sacrifice some accuracy in exchange for greater speed [36]. In addition to YOLO, other efficient architectures include [37]:

- MobileNet: Uses depth-wise separable convolutions to reduce the number of parameters and operations, achieving greater speed and energy efficiency.
- SqueezeNet: Significantly reduces the number of parameters through "fire" modules, making it lightweight and fast.

(b) Complexity and Accuracy

The complexity of a CNN architecture can be measured by the number of operations required to process an input image. Generally, architectures designed for detection tend to be more complex than those for classification [38].

The evaluation of a CNN's functional performance varies depending on the task, utilizing specific accuracy metrics. In classification, it is common to use Top-1 or Top-5 error as a metric to measure how often the correct class is among the top N predictions. For individual identification tasks, metrics like Rank-1 or Rank-5 are used to assess whether the correct identity is found among the N most probable matches. In object detection, mean Average Precision (mAP) is used (often with a specific IoU threshold, such as mAP@0.5 or

mAP@0.5:0.95) to evaluate both the class correctness and the localization accuracy of the bounding box. For segmentation, metrics like Intersection over Union (IoU) are used [39].

In general, more complex architectures tend to offer superior performance, reflecting a correlation between their complexity and the quality of the results obtained.

In [34], a comparative study was conducted with different types of architectures for classification and detection tasks (see Table 7), whose results are typically compiled from standard benchmarks such as ImageNet for classification and COCO/Pascal VOC for detection.

Table 7. Comparison of neural network architectures, CNNs.

Architecture	Task	Complexity (GOP)	Accuracy
AlexNet	Classification	0.7	16% error
VGG-16	Classification	15	7.4% error
GoogLeNet	Classification	1.4	6.7% error
ResNet50	Classification	3.9	5.3% error
MobileNet v2	Classification	0.3	9% error
ShuffleNet	Classification	0.26	10% error
Tiny Yolo	Detection	6.9	60% mAP
Yolo V2	Detection	39.4	76.8% mAP
Fast RCNN	Detection	300	78.2% mAP
Faster RCNN	Detection	150	76.4% mAP
SSD 300	Detection	34.9	74.3% mAP
SSD 512	Detection	52	76.8% mAP
SSD con MobileNet v2	Detection	1.2	72.7% mAP

According to Table 7, some architectures are more accurate than others, although the complexities reported in the sources correspond to architectures designed to handle a large number of classes. In practical applications, it is possible to reduce complexity by working with a smaller number of classes. Both the complexity and performance of a CNN architecture can be adjusted according to the specific requirements of the application [11].

The latest versions of YOLO were evaluated, highlighting the improvements introduced in YOLOv11, especially in accuracy, speed and robustness [40]. For the analysis, a dataset of annotated images for vehicle detection was used, including various types of vehicles. The dataset captures real-world conditions like daytime and nighttime lighting, weather variations (rain, fog), occlusions and different distances from the camera [34]. The performance of each model was measured using metrics such as mAP, precision and recall. The results indicate that YOLOv11 significantly outperforms its previous versions, establishing itself as an effective tool for real-time object detection and demonstrating its potential in practical applications that require high efficiency and accuracy (see Table 8).

Table 8. YOLO model comparison [40].

Architecture	Task	Speed	Accuracy
YOLOv8	Detection	200 FPS	mAP@0.5: 73.9%
YOLOv10	Detection	280 FPS	mAP@0.5: 74.3%
YOLOv11	Detection	290FPS	mAP@0.5: 76.8%

In this context, YOLOv11 represents the latest iteration of this family of models, with significant improvements in efficiency, accuracy and robustness. However, significant challenges remain, such as distinguishing between similar classes and adapting to variable environmental conditions [12]. These aspects continue to be active areas of research that could further optimize the performance of CNN architectures in practical applications.

While CNNs like YOLO and MobileNet have been the foundation of edge vision thanks to their efficiency in local feature extraction, their intrinsic design limits their ability to model long-range global context relationships within an image. To overcome this limitation, hybrid architectures have emerged that combine the efficiency of CNNs with the global modeling power of Vision Transformers (ViTs) [41].

One such architecture is MobileViT, introduced in [42] and designed as a lightweight vision transformer for mobile devices. MobileViT integrates the strengths of both architectures: it uses standard convolutions to capture spatial inductive biases and local features but employs Transformer blocks to efficiently model global inter-patch relationships. Experimental results demonstrate its superiority over equivalent lightweight CNNs and ViTs; the MobileViT-S model (with 5.6 M parameters) [42] achieves 78.4% Top-1 accuracy on ImageNet-1k, surpassing MobileNetv3 by 3.2% and DeIT by 6.2% with a similar parameter count. The impact of integrating MobileViT into existing architectures is significant. In [43], the CNN backbone of YOLOv8n was replaced with a MobileViT variant (MobileViTSF), achieving not only a 4.5% increase in mAP@0.5:0.95% accuracy but also a 51.9% reduction in FLOPs and a 41.9% reduction in model size, underscoring its suitability for edge devices.

Even so, MobileViT's spatial self-attention still incurs quadratic computational complexity. More recent architectures, such as EdgeNeXt, proposed in [44], optimize this further. EdgeNeXt introduces the Split Depth-wise Transpose Attention (SDTA) encoder, a key innovation that applies attention across the channel dimensions rather than the spatial dimensions. This design change reduces the computational complexity of self-attention from quadratic to linear with respect to the input size, making it ideal for the Edge [7]. In direct comparisons on ImageNet-1k, EdgeNeXt-S outperformed MobileViT-S, achieving 1.0% higher accuracy (79.4% vs. 78.4%) while using 35% fewer MAdds (1.30 G vs. 2.01 G), thus setting a new benchmark in the efficiency of hybrid vision architectures for the Edge.

The success of these hybrid architectures lies in optimizing the self-attention mechanisms, replacing the quadratic complexity of standard Transformers with linear or near-linear complexity approaches, making them computationally viable for resource-constrained devices.

5.1.2. Recurrent Neural Networks (RNNs)

RNNs are designed to process sequential data, such as text, speech and time series. Their main characteristic is the ability to maintain information from previous inputs through recurrent connections, making them ideal for tasks that require modeling temporal dependencies, such as natural language processing and speech recognition [45].

These networks store previous information in their internal memory, allowing them to generate predictions based on sequential contexts. However, they have limitations such as the vanishing gradient problem, which affects their ability to learn long-term dependencies. Despite this, they have been widely used in various applications [46].

Table 9 provides a comparison of the complexity and accuracy of different RNN architectures [45].

- Long Short-Term Memory (LSTM): Introduced to solve the vanishing gradient problem, they use gate mechanisms to control the flow of information, allowing them to maintain and update their internal state for long periods. This significantly extends their memory capacity compared to traditional RNNs [47]. Accuracy metrics for RNN tasks

are also specific; metrics such as Word Error Rate (WER) for speech recognition, the BLEU score for machine translation, or Root Mean Square Error (RMSE) for time series prediction can be employed [48].
- Gated Recurrent Unit (GRU): This is a simpler variant of LSTMs, which combines the forget and input gates into a single update gate, thus reducing computational complexity while maintaining similar performance [46].
- Transformers: Although they also address sequential tasks, they are not RNNs, as they replace recurrent connections with attention mechanisms that process global relationships in parallel, improving efficiency and the ability to model long-term dependencies. This makes them more suitable for tasks that require large volumes of data and high levels of parallelism [49,50].

Table 9. Performance of RNNs (LSTM/GRU) at the Edge [46,51].

Architecture	Advantages	Disadvantages
RNNs (LSTM/GRU)	Optimized variants (GRUs) can be exceptionally lightweight and energy-efficient, rendering them suitable for MCUs. LSTMs may offer a robust trade-off between high accuracy and a manageable RAM footprint (~4.4 GB) for specific tasks. They can achieve real-time performance on MCUs with optimizations (low power consumption and RAM overhead).	Standard (non-optimized) RNNs may exhibit prohibitive RAM requirements (>22 GB), making them unfeasible for edge deployment. Their inherently sequential nature constrains parallelism on hardware accelerators [Inferred]. The irregular memory access patterns (typical of LSTMs) can degrade performance on NPUs reliant on DMA (Direct Memory Access).
Transformers	NPUs can demonstrate high efficiency for LLMs (e.g., TinyLlama, 3.2× speedup over GPUs) owing to the predominance of matrix-vector operations. They may exhibit a reduced RAM footprint compared to standard RNNs (1.8–3.7 GB vs. >22 GB in one study). Their parallelizable architecture is well-suited for future compatible accelerators.	Poor compatibility with current mobile accelerators (GPUs, DSPs, NPUs); execution frequently reverts to the CPU, or acceleration is minimal to non-existent. GPU execution may compromise model accuracy. Standard self-attention possesses high computational complexity (quadratic) and significant memory overhead during computation. May demonstrate inferior accuracy compared to LSTMs for certain time-series tasks.

While establishing a direct, generalized comparison is complex owing to the inherent diversity of models, tasks, and hardware, specific studies provide insight into the requisite resource demands [52]. Table 10 presents a comparative latency analysis for an LSTM model and a compact Transformer (TinyLlama) deployed on a heterogeneous edge platform (Intel AIPC equipped with GPU and NPU), utilizing FP16 operations.

Table 10. Latency Comparison: LSTM vs. Transformer [37,53].

Model	Type	Parameters	Latency (GPU)	Latency (NPU)	Preferred Unit
LSTM	RNN	1.6 M	1.48 ms	4.10 ms	GPU (2.7×)
TinyLlama	Transformer	1.1 B	8.30 seg	2.49 seg	NPU (3.2×)

As illustrated in Table 10, the key finding is the absence of universally superior architecture in the edge environment. Specifically, the NPU excels in the performance of the Transformer model (TinyLlama) due to its efficiency in matrix-vector operations, which are dominant in the decoding of LLMs [53]. In contrast, the GPU offers better handling of the LSTM in this case [48], since the irregular memory access patterns inherent to the LSTM reduce the NPU's performance.

5.1.3. Compact Neural Networks

These architectures are designed to optimize resource usage in embedded systems with limited computational capabilities, such as Tiny Machine Learning (TinyML). Their primary goal is to reduce memory consumption and improve computational efficiency without significantly compromising accuracy [54]. The development of compact neural networks is based on minimizing weights and connections within the model. This is achieved through techniques such as pruning, quantization and the use of optimized architectures. A relevant strategy is knowledge distillation, which allows information to be transferred from a complex model to a lighter one [55].

Among the strategies used in compact neural networks, self-distillation has proven to be an effective technique, allowing a neural network to transfer knowledge from its deeper layers to the shallower ones. This approach contributes to reducing inference times and energy consumption while maintaining competitive performance [56].

To evaluate its impact, tests have been carried out on various architectures, the results of which (see Table 11) demonstrate that self-distillation improves the accuracy and efficiency of compact neural networks. Additionally, it is compatible with other model compression methods, allowing for further optimization of computational resources [57].

Table 11. Comparison of different networks applying self-distillation.

Experiment	Architecture	Dataset	Reduction of FLOPS	Parameter Reduction
Image Classification	ResNet-56	CIFAR-100	16.6%	14.8%
	ResNet-110	CIFAR-100	15.9%	13.5%
	ResNet-50	ImageNet	10.8%	9.6%
	WideResNet-28-10	CIFAR-100	10.5%	9.3%
	ResNeXt-29-4x64d	CIFAR-100	12.1%	10.7%
	SE-ResNet-56	CIFAR-100	14.2%	12.8%
	ResNeSt-50	ImageNet	8.5%	7.9%
	MobileNetV2	ImageNet	6.3%	5.8%
	ShuffleNetV2-1.0x	ImageNet	5.1%	4.7%
Point Cloud Classification	ResGCN	ModelNet40	11.2%	10.1%

An innovative proposal in this field is BUnit-Net [54], which achieves a significant reduction in the number of parameters and FLOPs compared to traditional neural networks, while maintaining comparable accuracy in object detection tasks (see Table 12).

Table 12. Performance and efficiency of BUnit-Net across different datasets.

Dataset	Model	Accuracy (%)	FLOPs Reduction (%)	Parameters Reduction (%)
MNIST	MLP	91.95	97	96
CIFAR-10	VGG-16	93.25	79.87	72.90
	ResNet-20	94.52	55.02	49.35
	MobileNetV2	91.58	38.56	36.81
CIFAR-100	VGG-16	72.51	73.05	70.26
	ResNet-20	76.48	57.46	62.56
	MobileNetV2	71.54	62.61	60.48
ImageNet	ResNet-34	71.51	42.05	35.31
	ResNet-50	75.33	43.80	43.67
Tiny-ImageNet	VGG-19	53.54	63.42	54.63

To evaluate the performance of compact neural networks in embedded systems, the authors of [11] conducted experiments on devices such as the NVIDIA Jetson TX2 and Jetson Nano. In this study, it was demonstrated that applying quantization and pruning techniques allows for significant latency reduction without affecting model accuracy. The key results are presented below (see Table 13).

Table 13. Results using compact networks in embedded systems.

Model	(Top-1) Accuracy (%)	FLOPs Reduction (%)	Latency Predictor Error (%)
VGG-19	83.78	83.02	6.12
	83.12	83.02	6.12
ResNet-50	84.80	76.50	6.12
	85.64	23.79	6.12
GoogLeNet	82.80	77.22	6.12
	85.12	23.18	6.12

The results show that the average latency predictor error is 6.12%, which indicates a good predictive capability for inference latency. It has also been identified that the ZeroBN-90 architecture offers an optimal combination of accuracy and reduction in computational costs.

5.1.4. FeedForward Neural Networks (FNNs)

FNNs are architectures in which information flows in only one direction, from the input layer to the output layer, without recurrences or feedback connections. They are commonly used for regression and classification tasks due to their simplicity and ease of implementation [58]. FNNs operate through a feedforward process, where input data are propagated through the network, layer by layer, until they reach the output layer. To determine the error, the network's output is compared with the desired values and the error is calculated, thus leading to backpropagation [59]. They have limitations, such as sensitivity to the initialization of weights and the inability to effectively model data sequences with temporal dependencies.

Despite their limitations compared to more advanced networks, FNNs are still widely used in various applications, including image recognition, natural language processing

and financial forecasting. Their ability to model complex relationships makes them ideal for various tasks such as stock prediction, image classification or text generation [60].

The study in [59] focused on the efficient implementation of feedforward neural networks in FPGA-based embedded systems, aiming to optimize performance for virtual sensor applications. The proposed methodology decomposes the FNN into elementary layers (summation, multiplication, activation functions) and uses high-level synthesis (HLS) tools to generate an efficient implementation on the FPGA (see Table 14).

Table 14. Performance of different FNN implementations on FPGA.

Implementation	Time per Sample (µs)	Total Time (for 10,000 Samples)	Design Complexity	Accuracy
Simple FNN-High Precision 1	0.0555	555 µs	Low	High
Simple FNN-High Precision 2	0.0586	586 µs	Low	High
Simple FNN-Medium Precision 1	0.0620	620 µs	Low	Medium
Simple FNN-Medium Precision 2	0.0578	578 µs	Low	Medium
Complex FNN-High Precision	1.04	10.4 ms	High	High
Medium FNN-Medium Precision	0.08067	806.7 µs	Medium	Medium

5.2. Hardware Platforms for Implementation

Efficient implementation of neural networks in embedded systems depends heavily on the chosen hardware platform. These platforms must balance processing capacity, energy efficiency, memory and cost to meet the specific constraints and requirements of applications [14].

- Microcontrollers (MCUs): A popular choice for low-power, cost-effective applications, especially in the TinyML domain. Modern MCUs like the ARM Cortex-M series offer a good balance of processing capability and energy efficiency for simple tasks [14].
- Field-Programmable Gate Arrays (FPGAs): These are highly flexible integrated circuits that can be configured by the user after manufacturing. Their reconfigurability and parallel processing capabilities make them ideal for implementing custom neural network accelerators with high energy efficiency [58].
- Neural Processing Units (NPUs): Designed specifically for neural network operations, these processors deliver high performance and energy efficiency, making them ideal for AI workloads. Their massively parallel architecture enables simultaneous calculations, crucial for deep learning algorithms with extensive matrix operations. NPUs have a tailored memory hierarchy, minimizing data movement and maximizing resource utilization through on-chip memory and dataflow architectures [14].
- Edge TPUs: These are low-power versions of TPUs, designed for neural network inference on embedded devices. They offer a good balance between performance and

energy efficiency. Their main advantages are performance, energy efficiency and ease of use [14].

- Deep Learning Accelerators for RISC-V Processors: Known for their open source nature and flexibility, they are being used to optimize the implementation of deep neural networks (DNNs) on edge devices. Their advantages are flexibility and customization, extensibility, and community support [28].

Software Framework

Software frameworks play a fundamental role in the implementation of neural networks in embedded systems, as they provide tools and libraries that facilitate the development, optimization and execution of Deep Learning models on devices with limited resources. The choice depends on the hardware and the model's origin.

In [14], a comparative analysis of different frameworks is carried out, with an emphasis on DeepliteRT [61], an inference engine for ARM CPUs that specializes in executing deep learning models quantized at ultra-low precisions (less than 4 bits). DeepliteRT accelerates inference using optimized convolution kernels that take advantage of the Arm hardware's low-level intrinsic instructions, such as Neon instructions [62]. Additionally, DeepliteRT uses tiling and parallelization techniques to further optimize performance. Compared to other inference engines, such as TensorFlow Lite (TFLite) [63], it improves inference speed through XNNPACK (eXtended Neural Network Package) and its high-performance kernels and ONNX Runtime (Open Neural Network Exchange Runtime) [64]. Moreover, DeepliteRT has shown superior performance in running quantized models. The test results show that DeepliteRT can achieve speedups of up to 2.2× and 3.2× over TFLite with XNNPACK and ONNX Runtime, respectively [65].

For compact models, such as YOLOv5n [14], DeepliteRT uses a mixed precision approach to minimize precision loss. This approach involves keeping some layers sensitive to quantization in higher precision formats like FP32 (32-bit Floating Point) or FP16 (16-bit Floating Point), while the rest are quantized to a lower precision, typically INT8. DeepliteRT integrates with Deeplite Neutrino™ [61], a quantization framework that enables training-aware quantization. This means that the model is trained to work with less precision, which minimizes precision loss compared to post-training quantization.

In addition to the one mentioned in the aforementioned research, there are other relevant software frameworks for the implementation of neural networks in embedded systems [33]:

- TensorFlow Lite: It is a lightweight version of TensorFlow optimized for mobile and embedded devices. It offers a set of tools for model conversion, performance optimization and implementation on different hardware platforms. TensorFlow Lite is used in a wide variety of applications, such as image recognition, natural language processing and object detection [63]. A specialized version is TensorFlow Lite for Microcontrollers (TFLM). TFLM is explicitly designed to run on microcontrollers and other devices with extremely limited memory (in the kilobyte range) [66]. It can operate without an operating system or a file system, and is considered the standard inference engine for TinyML use cases [67].
- Cortex Microcontroller Software Interface Standard for Neural Networks (CMSIS-NN): It is a library of optimized functions for neural network operations on ARM Cortex-M processors. CMSIS-NN accelerates the execution of neural networks on microcontrollers, allowing the implementation of AI applications on devices with very limited resources [68].
- PyTorch Mobile: It is a lightweight version of PyTorch optimized for mobile and embedded devices. PyTorch Mobile offers tools for model conversion, performance optimization and implementation on different platforms [69].

- OnnxRuntime: It is a high-performance inference engine for models in ONNX (Open Neural Network Exchange) format. ONNX is an open format that allows interoperability between different Deep Learning frameworks. OnnxRuntime can be used to run models from different frameworks on a variety of hardware platforms [64].
- Glow: A Machine Learning compiler designed to optimize and generate code for various hardware architectures, including accelerators. Glow takes computational graphs from frameworks such as PyTorch or ONNX, performs graph-level optimizations, and generates specific code (compiled library packages) [28]. Facebook (Meta) uses it as an intermediate layer in its inference acceleration platform.
- Apache TVM: It is a complete, open-source compilation stack that aims to close the gap between deep learning frameworks and hardware backends [70]. TVM supports model importing from multiple frameworks (PyTorch, TensorFlow, Keras, ONNX, etc.) and can generate optimized code for a wide range of hardware, including CPUs, GPUs, FPGAs, and bare-metal microcontrollers (via microTVM). It incorporates a powerful auto-tuning engine to optimize the execution order and memory access of tensor operations. Frameworks such as MATCH extend TVM to improve compilation on heterogeneous edge devices with custom accelerators [71].

5.3. Comparative Analysis

The manuscript in [37] examines current tools and techniques for efficient deep learning inference on resource-constrained edge devices. In particular, various model compression techniques, such as pruning, quantization and knowledge distillation, are analyzed with the aim of reducing the size and computational complexity of deep neural networks. Different edge hardware platforms, such as FPGAs, MCUs, TPUs and ASICs, are also explored, along with algorithm-hardware co-design techniques for efficient inference (see Table 15) [72]. It also describes metrics to evaluate edge inference performance, such as model size, accuracy, power consumption, latency and throughput.

Table 15. Comparative analysis of neural network architectures.

Architecture	Description	Applications	Compression	Hardware Acceleration
FNN	Fully connected feedforward neural network.	Classification, regression.	Pruning, quantization.	FPGA, ASIC
CNN	Convolutional neural network.	Computer vision, image processing.	Pruning, quantization, knowledge distillation.	GPU, FPGA, ASIC, TPU
RNN	Recurrent neural network.	Sequence processing, time series analysis.	Pruning, quantization.	FPGA, ASIC
LSTM	Long Short-Term Memory (a type of RNN).	Natural language processing, machine translation.	Pruning, quantization.	FPGA, ASIC
GRU	Gated Recurrent Unit (a type of RNN).	Natural language processing, machine translation.	Pruning, quantization.	FPGA, ASIC
Transformer	Attention-based architecture.	Natural language processing, machine translation.	Pruning, quantization.	TPU, GPU
Compact Nets	Models with a reduced number of parameters.	Resource-constrained edge applications.	Efficient design, quantization.	Various hardware platforms.

According to Table 15, each architecture has strengths and weaknesses, making them suitable for different applications. Compression and hardware acceleration techniques are essential for implementing these architectures on edge devices [12]. Table 16 provides a comparative analysis of neural network architectures across various hardware platforms, based on data from [37], which compiles findings from prior research and manufacturer specifications.

Table 16. Comparison of neural network architectures across different platforms for Image classification and speech recognition [73].

Architecture	Hardware Accelerator	Task	Quantization	Energy Efficiency
CNN	XCZU7EV (FPGA)	Classification (MobileNetV2)	12/10 bits	0.37 GOPS/W
CNN	XC7Z020 FPGA (SCE Co-Design)	TSR (LeNet-5)	-	~14.7 GFLOPS/W
RNN (GRU)	XC7Z007S FPGA (EdgeDRNN)	Reconocimiento Voz (TIDIGITS)	INT16/8	8.8 GOPS/W
RNN (LSTM)	Arria 10 GX1150 FPGA	Reconocimiento Voz	INT16	15.9 GOPS/W
CNN/RNN	ASIC (LNPU—KAIST)	Inferencia/Aprendizaje (CONV/FC)	Float8	25,300 GPOS/W
RNN (Delta)	XC7Z100 FPGA	Reconocimiento de Voz	16 bits	26.3 GOPS/W

The selection of the neural network architecture, hardware platform and optimization techniques (pruning, quantization, distillation, among others) for inference in embedded systems depends on the nature of the task and the available resource limitations [74]. Each family of networks has specific strengths:

- CNNs excel in computer vision.
- RNNs/LSTMs/GRUs are more suitable for processing sequences (text, speech, time series).
- Transformers offer great capacity for parallelism and handling long-term dependencies.
- Compact networks, combined with compression strategies, are ideal when very limited resources are available.

Furthermore, hardware acceleration can reduce latency and energy consumption without sacrificing accuracy. Ultimately, achieving a balance between performance, inference speed and resource consumption is essential to efficiently implement these solutions at the edge, meeting the demands of each real application [75].

5.4. Comparative Positioning Against Existing Surveys

To clearly situate the contribution of this review within the existing body of literature, it is essential to compare its scope, methodological focus, and practical impact with the major surveys most frequently cited in the field. Although previous works have extensively covered model compression techniques, lightweight architectures, TinyML constraints, and accelerator design, these surveys typically approach the topic from a descriptive or component-specific perspective. None of them provides a unified framework that integrates model selection, optimization strategies, hardware-aware decisions, and real-world validation into a coherent and reproducible deployment pipeline. Accordingly, Table 17 summarizes the characteristics of the principal surveys referenced in this work—covering embedded architectures, quantization for microcontrollers, accelerator design, object-detection frameworks, and embedded machine-learning taxonomies. Table 17

highlights their respective scopes and limitations and contrasts them with the operational perspective of the present review.

Table 17. Comparative Positioning of Major surveys Referenced in This Review.

Survey	Scope	Limitations	Additional Contribution of This Review
Chen et al., "Deep Learning on Mobile and Embedded Devices: State-of-the-Art, Challenges, and Future Directions" (2021) [6]	Broad examination of DL techniques applied to mobile and embedded environments.	Lacks an operational deployment workflow; limited integration of model optimization with hardware-specific constraints.	Provides a structured five-stage methodology that links optimization, architecture selection, and hardware requirements into a unified deployment process.
Chen et al., "DNN-Based Vehicle and Pedestrian Detection for Autonomous Driving: A Survey" (2021) [38]	Comprehensive survey focused on perception tasks for autonomous driving.	Application-specific; does not generalize to Edge AI deployment or resource-constrained optimization.	Extends analysis beyond automotive tasks, offering a generalizable workflow for embedded AI across multiple domains.
Ali & Zhang, "The YOLO Framework: A Comprehensive Review" (2024) [76]	Detailed taxonomy of the YOLO family and its evolution.	Architecture-specific; lacks discussion of compression, quantization, or hardware alignment.	Integrates YOLO within a broader system-level methodology that includes model compression and accelerator-aware deployment.
Novac et al., "Quantization and Deployment of Deep Neural Networks on Microcontrollers" (2021) [77]	Focused review on TinyML and quantization techniques for MCUs.	Limited to microcontroller-class systems; excludes NPUs, TPUs, and heterogeneous edge hardware.	Covers the full spectrum of hardware platforms, from MCUs to high-performance NPUs, within a common methodological framework.
Berthelier et al., "Deep Model Compression and Architecture Optimization for Embedded Systems: A Survey" (2021) [78]	Extensive review of pruning, quantization, and architecture-level optimization.	Does not incorporate hardware benchmarking nor present a reproducible deployment workflow.	Provides integrated empirical evaluation and a multi-stage methodology grounded in both literature evidence and real hardware trials.
Dhilleswararao et al., "Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey" (2022) [79]	Survey of hardware architectures for DNN acceleration, including FPGA and ASIC platforms.	Hardware-centric; lacks alignment between model-level techniques and accelerator constraints.	Bridges model-level optimization with hardware characteristics, enabling informed co-design decisions within the methodology.
Akkad et al., "Embedded Deep Learning Accelerators: A Survey on Recent Advances" (2024) [28]	Review of recent embedded DNN accelerators and their architectural design.	Does not address resource-aware model optimization or end-to-end deployment strategies.	Positions accelerators within a complete deployment workflow, including model selection, quantization, and validation.
Alam et al., "Survey of Deep Learning Accelerators for Edge and Emerging Computing" (2024) [80]	Examines accelerator trends in emerging edge-computing hardware.	Lacks methodological guidance for selecting and optimizing models for each hardware platform.	Enhances hardware discussions with a reproducible five-stage decision-making methodology and cross-platform benchmark synthesis.
Biglari & Tang, "A Review of Embedded Machine Learning Based on Hardware, Application, and Sensing Scheme" (2023) [11]	Broad overview of embedded ML applications and sensing modalities.	Does not integrate optimization techniques or performance-driven hardware selection.	Provides a holistic framework that unifies optimization, architecture design, and empirical hardware validation.

Overall, this comparative analysis demonstrates that, while existing surveys offer valuable insights into specific dimensions of embedded deep learning—such as lightweight architectures, quantization strategies, or the evolution of accelerator hardware—none provides an integrated, end-to-end methodology suitable for real-world deployment. The present review fills this gap by consolidating the fragmented evidence found across the literature into a coherent five-stage workflow that spans requirement definition, model selection, optimization, hardware alignment, and empirical validation on target devices.

By grounding the methodology in both systematic evidence and practical experimentation, this work advances the field beyond descriptive taxonomies toward a structured and reproducible decision-making framework for Edge AI implementation.

The evidence synthesized across Sections 3–5 reveals coherent patterns in model-level optimization, lightweight architecture design and hardware constraints that consistently shape the feasibility of embedded deployment. Across the reviewed studies, it becomes clear that achieving practical, resource-aware implementation requires a structured process that links requirement definition, model selection, optimization techniques and system-level validation. To operationalize this multi-dimensional evidence, the following section introduces a five-stage deployment methodology directly grounded in the limitations, opportunities and recurring design patterns identified throughout this review.

6. A Proposed Methodology for Optimization and Deployment

This methodology consolidates the recurring constraints, optimization patterns and hardware limitations identified throughout Sections 3–5, transforming the evidence into a practical and reproducible workflow for embedded deployment.

The comprehensive review presented in Sections 4 and 5 highlights the vast and complex landscape of optimization techniques, architectures and platforms available for deploying neural networks on embedded systems. To navigate this complexity, developers need a structured approach. Based on our analysis, we propose the following five-stage methodology designed to guide the process from initial requirements to final deployment (see Figure 3).

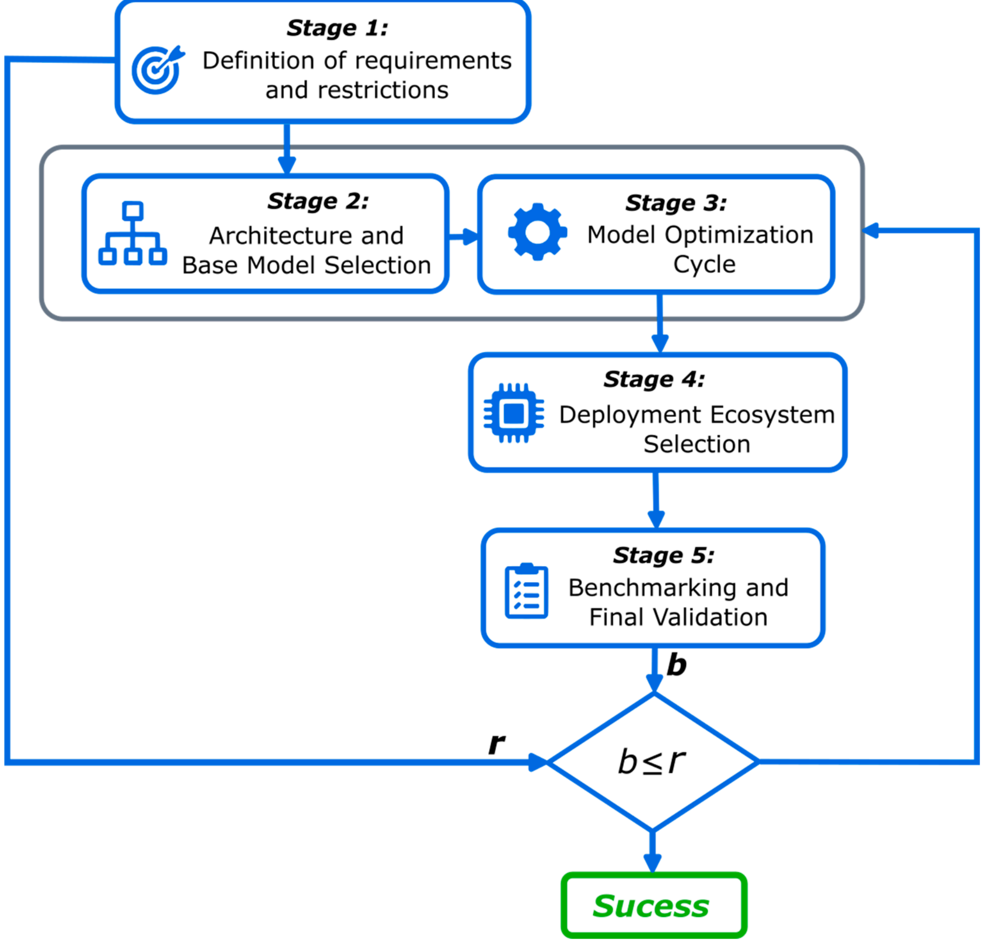

Figure 3. Proposed methodology for optimizing and deploying neural networks on embedded systems.

- Stage 1: Definition of Requirements and Constraints

This stage commences with the precise definition of the main task, as this will dictate the choice of the neural network architecture. In the domain of computer vision, tasks such as classification, object detection and image segmentation are diverse, for which CNNs are the predominant architecture. In contrast, sequential data processing for time-series prediction or audio recognition benefits from RNNs.

Subsequently, the quantitative constraints that represent the hard operational limits of the embedded system must be established. This involves defining a maximum latency, a threshold for inference time that cannot be exceeded. This factor is critical in real-time applications like ADAS or surveillance, where a delayed response can be unacceptable, as their efficiency and safety depend on significantly low latencies. Fixing this limit ensures that the system operates safely and effectively within its operational time window. A power budget must also be specified, detailing the maximum power consumption in W or mW. This restriction not only determines the solution's viability in battery-powered devices or those with limited thermal dissipation but is also a fundamental requirement that directly constrains the hardware selection. Furthermore, the memory footprint establishes both the maximum storage size of the model (measured in MB) and the permissible RAM usage during runtime. This is one of the most defining constraints of embedded systems, which by nature operate with limited memory resources.

Finally, it is fundamental to set the minimum accuracy target for the model to be functionally useful. This metric is task-dependent, such as the mAP for object detection. This objective must be balanced against the other constraints, as there is an inherent trade-off between a model's accuracy and its efficiency in a resource-constrained environment.

- Stage 2: Selection of Architecture and Base Model

Building upon the requirements defined in Stage 1, this phase involves selecting the most suitable architecture family and a specific base model. The choice is based on the task domain, justifying the use of each family by its fundamental characteristics.

For computer vision, CNNs are typically employed, given their capability to extract relevant features from images in tasks such as classification, segmentation, or object detection. The selection of a specific model within this family is based on the project's constraints: models like YOLO are chosen when a balance between speed and accuracy is sought, whereas architectures like MobileNet or SqueezeNet are preferred when efficiency is the top priority.

For sequential data processing, the choice lies with RNNs, due to their primary characteristic of modeling temporal dependencies. Within this family, the selection is narrowed down to variants like LSTM and GRU, or to alternatives such as Transformers if handling long-term dependencies and parallelism are determining factors. For scenarios with extreme constraints (TinyML), the selection is oriented towards compact architectures, explicitly designed to optimize resource usage on microcontrollers.

Once the most suitable architecture family has been identified, the strategy concludes with the selection of a pre-trained, lightweight variant as the base model (e.g., YOLO-tiny, MobileNetV2). This provides a solid and already-optimized starting point for the compression and fine-tuning cycle of the next stage.

As established in our review of architectures (Section 6.1), the choice of the base model depends on a trade-off between accuracy and efficiency. For applications where speed is critical, architectures like YOLO are preferable, whereas MobileNet or SqueezeNet are prioritized when power consumption is the main constraint, as demonstrated by the comparison in Table 7.

- Stage 3: Model Optimization Cycle

This stage consists of an iterative process aimed at reducing the computational complexity and size of the base model to meet the constraints defined in Stage 1, while simultaneously minimizing accuracy loss.

The optimization cycle follows a progressive sequence, beginning with techniques that offer the highest impact for the lowest complexity. The first step is quantization, a technique that reduces the numerical precision of the model's weights and activations, commonly by converting 32-bit floating-point values to 8-bit integers. This adaptation is often indispensable, as it aligns the model with the capabilities of embedded hardware, which is typically optimized for low-precision integer operations. Quantization is applied first because it is a straightforward method that significantly reduces model size and accelerates inference. If quantization alone is insufficient to meet the objectives, the process continues with pruning, a technique that removes non-essential connections or neurons to reduce model complexity. This process, however, typically requires re-training to recover the lost accuracy.

When even greater compression is needed, advanced optimization techniques are employed. One of these is knowledge distillation, where a smaller "student" model is trained to mimic the behavior of a larger, more accurate "teacher" model. Finally, extreme quantization, such as binarization (1-bit) or ternarization (2-bit), can be considered. These techniques offer maximum compression and are ideal if the target hardware can leverage bit-level operations, although their application is complex and requires specialized training strategies to manage the loss of accuracy.

- Stage 4: Selection of the Deployment Ecosystem

Once the model is optimized, this stage involves selecting the hardware and software combination on which it will be executed in the target environment.

The choice of the hardware platform is directly linked to the application's requirements, such as power consumption, cost and processing capability. For low-power and low-cost scenarios, typical of TinyML, MCUs such as the ARM Cortex-M series are the preferred option. For high-performance tasks, such as real-time computer vision, specialized accelerators like Edge TPUs, NPUs, or boards with integrated GPUs, such as the Nvidia Jetson family, are chosen. If the application demands maximum flexibility and the implementation of custom accelerators, FPGAs are the ideal alternative due to their reconfigurable nature and their potential for high energy efficiency.

The selection of the software framework must be aligned with the chosen hardware platform and the model's origin. Ecosystems such as TFLite are used for mobile and embedded devices, especially in conjunction with Edge TPUs. Analogously, PyTorch Mobile is used for models developed in PyTorch. To ensure interoperability between different frameworks and hardware, ONNX Runtime is a robust solution that operates on the standard ONNX format. Additionally, for models that have undergone very low-bit quantization, it is advisable to consider specialized runtimes like DeepliteRT, which are optimized to efficiently execute these types of operations.

- Stage 5: Benchmarking and Final Validation

The final stage of the methodology consists of deploying the optimized model onto the target hardware and performing a rigorous validation to ensure all predefined requirements are met. This stage is not merely a final check, but the trigger for the methodology's iterative feedback loop, which is crucial for achieving an optimal solution.

The process begins with the deployment of the optimized model from *Stage 3* using the software framework selected in *Stage 4*. Once deployed on the target hardware, a series of benchmarks is executed to measure the model's real-world performance. Key metrics

such as latency (ms), throughput (FPS), power consumption (W) and task-specific accuracy (e.g., mAP) are collected. This collection of measured performance metrics constitutes the benchmark vector, which we denote as b.

These results are then systematically compared against the collection of goals and constraints defined in Stage 1, which we will call the requirements vector r. The core of the validation is to satisfy the condition where every measured benchmark in b meets its corresponding requirement in r.

Based on this validation, the methodology dictates one of the following paths, as illustrated in the feedback loop of Figure 3. If all benchmarked metrics in b successfully meet the requirements defined in r, the deployment is considered a success and the process concludes with the resulting model validated for the specific application. If, however, one or more benchmarks fail to meet the requirements, the methodology requires an iteration and refinement by returning to a previous stage. The choice of which stage to return to depends on the nature and magnitude of the failure. The most common feedback loop is a return to Stage 3, chosen when performance metrics are close to the target but require further tuning. For instance, if the accuracy is slightly below the target, one might revisit the pruning strategy or apply QAT. If latency or power consumption are too high, more aggressive optimization techniques could be explored.

A more fundamental loop is taken back to Stage 2 when there is a significant gap between the benchmarks and the requirements, suggesting that the chosen base model is fundamentally too complex or unsuitable for the task. In this case, no amount of optimization in Stage 3 will be sufficient and a lighter or different base architecture must be selected. In some cases, the validation process may reveal that the initial requirements were unrealistic or mutually exclusive (e.g., demanding extremely low latency and extremely high accuracy with a very limited power budget), which triggers a return to Stage 1 to re-evaluation and adjustment of the project's foundational goals to align with what is technically feasible.

This iterative process ensures that the final deployed model is not just a theoretical construct, but a solution robustly validated against the practical constraints and performance goals of the real-world application.

6.1. Case Study: Real-Time Object Detection in Edge AI

This case study illustrates the application of the five-stage methodology to a common Edge AI scenario: the deployment of a real-time vision system for person and object detection with defined performance requirements.

- Stage 1: Definition of Requirements and Constraints (r)

The precise definition of operational limits is the fundamental step. For this monitoring system, the requirements are established as follows:

- ○ Primary Task: Detection of people and objects in a controlled environment.
- ○ Quantitative Constraints (r):

Maximum Latency (L_{max}): The minimum requirement is to process between (1000 ms/5 = 200 ms) and 10 FPS (1000 ms/10 = 100 ms), (end-to-end) for smooth monitoring. This is the dominant constraint.

- ○ Power Budget (P_{max}): <5 Watts. Consumption must be low despite being connected to the grid, to avoid the need for active heat dissipation (fans), which increases reliability and reduces cost.
- ○ Maximum Memory (M_{max}): <512 MB (execution RAM). The operating system and other processes consume resources, leaving this strict limit for model operation.

- ○ Minimum Accuracy (A_{max}): >90% mAP. Detection must be highly reliable, especially since the environment is controlled, which raises the required accuracy threshold.

- Stage 2: Selection of Architecture and Base Model

The second stage establishes the starting point of the design by selecting the architecture and pre-trained base model that minimizes computational cost without compromising the required accuracy.

- ○ Neural Network Architecture

The Primary Task is the real-time detection and localization of people and objects. This task intrinsically demands CNN architectures optimized for Object Detection, as they must manage both class classification and position regression (bounding box). Consequently, the YOLO family of architectures is chosen, recognized for its efficiency and speed in real-time prediction.

- ○ Base Model Selection

Deployment at the Edge AI requires resolving the fundamental conflict between accuracy ($A_{max} \geq 90\%$ mAP) and efficiency ($L_{max} \leq 100$ ms). Given that Maximum Latency (L_{max}) is the dominant constraint, the base model must be intrinsically lightweight to meet the time and power requirements ($P_{max} < 5$ W). The YOLOv8n variant is chosen as the smallest and fastest in its family, explicitly designed for edge deployments. The suffix '*n*' (nano) indicates the version with the minimum parameters and computational complexity (FLOPs), providing the best speed-accuracy trade-off. This choice maximizes the probability of achieving the latency target. Furthermore, the YOLOv8 architecture ensures that, through optimization, the model retains sufficient representational capacity to exceed the minimum required accuracy ($A_{max} \geq 90\%$ mAP).

- Stage 3: Model Optimization Cycle

This stage initiates the optimization cycle to reduce the computational complexity of the base model (YOLOv8n) and adjust it to the constraints of Stage 1, prioritizing the minimization of accuracy loss.

Since the base model in full precision (FP32) results in an unacceptable L_{max} when executed solely on the CPU, acceleration via a specialized processing unit (GPU or NPU) is mandatory. Consequently, quantization is established as the dominant optimization technique. This conversion to low-precision formats (INT) is a functional requirement, as hardware accelerators are intrinsically optimized to operate with integers. Applying quantization is essential to achieve the required P_{max} and inference speed.

However, after the initial quantization is applied, the model will be validated in Stage 5. Only if the accuracy drops below the threshold ($A_{max} \geq 90\%$ mAP), will more complex optimization techniques, such as QAT or pruning, be introduced in a subsequent cycle iteration.

- Stage 4: Selection of the Deployment Ecosystem

This stage finalizes the choice of the Deployment Ecosystem by selecting the optimal combination of hardware and software to execute the optimized INT8 model, in accordance with the performance and power constraints of Stage 1.

- ○ Hardware Platform Selection

The NXP i.MX 8M Plus board is selected. This platform is ideal for edge inference, as its heterogeneous computing architecture combines multiple processing units. Specifically, it integrates an ARM Cortex-A53 CPU for general-purpose tasks and post-processing, a GPU, and a 2.3 TOPS NPU specially designed to accelerate quantized Machine Learning models. This directly meets the mandatory accelerator requirement established in Stage 3.

◦ Software Ecosystem Selection and Configuration

The NXP hardware dictates the ecosystem. TensorFlow Lite (TFLite) is used as the runtime. However, the .tflite model is not used directly; it must be compiled offline using the NXP eIQ Toolkit, which converts the optimized model to efficiently utilize the NPU's delegate.

- Stage 5: Benchmarking and Final Validation

The final stage consists of deploying the optimized model and rigorously validating its performance on the target hardware, comparing the measured results (b) with the initial requirements (r).

◦ Deployment and Performance Measurement: The optimized model (YOLOv8n-INT8), compiled with the chosen ecosystem's toolkit (Stage 4), is deployed on the i.MX 8M Plus board. The system's end-to-end performance is measured, which includes pre-processing, accelerated inference on the NPU, and post-processing executed on the CPU. The results that meet the success condition ($b \leq r$) are summarized in Table 18, demonstrating the complete validation of the methodological design.

Table 18. Results with Success Condition.

Metric	**Requirements (r)**	**Measurement (b) on NPU**	**Condition ($b \leq r$)**	**Result**
Latency	$\leq$Between 100 ms and 200 ms	132 ms (7.5 FPS)	$132 \leq$ 100–120	Success
Accuracy (mAP)	$\geq$90%	91.5%	$91.5 \geq 90$	Success
Power (Active)	<5 W	4.1 W	$4.1 < 5$	Success

The results presented in Table 18 correspond to the optimized deployment scenario on the NPU, which meets the success condition. The complete analysis, contrasting the performance obtained on the CPU and the GPU against the NPU, is presented in Table 19.

Table 19. Comprehensive Final Results.

Metric	**CPU**	**NPU**	**GPU**
Latency	980 ms	132 ms	975 ms
Accuracy (mAP)	94%	91.5%	92%
Power (Active)	5.1 W	4.1 W	5 W

6.2. Formal Analysis of the Feedback Loop and the Trade-Off Space

The core of the proposed methodology lies in Stage 5 (Benchmarking and Validation), which functions as a quantitative comparator that activates the feedback loops shown in Figure 3. This process is formalized by defining two key vectors:

Requirements Vector (r): Defined in Stage 1, it represents the operational limits of the system in each dimension.

$$r = \{L_{max}, P_{max}, M_{max}, A_{max}\}$$

Benchmark Vector (b): Measured in Stage 5, this represents the actual performance of the optimized model on the target hardware.

$$b = \{L_{med}, P_{med}, M_{med}, A_{med}\}$$

The Success Condition requires that every measured component in b complies with the constraint defined in r. Formally,

$$\text{SUCCESS} = (L_{med} \leq L_{max}) \wedge (P_{med} \leq P_{max}) \wedge (A_{med} \leq A_{max})$$

If this condition fails, the Feedback Loop is immediately activated, mandating a return to Stage 3 (Optimization) to apply additional techniques (such as structured pruning) or resolve an identified bottleneck.

For the applied case study, the navigation of the Trade-off Space—defined by the inverse relationships between latency, power, and accuracy—can be illustrated. Table 19 and Figure 4 show the evaluated configurations of the YOLOv8n model on the i.MX 8M Plus platform.

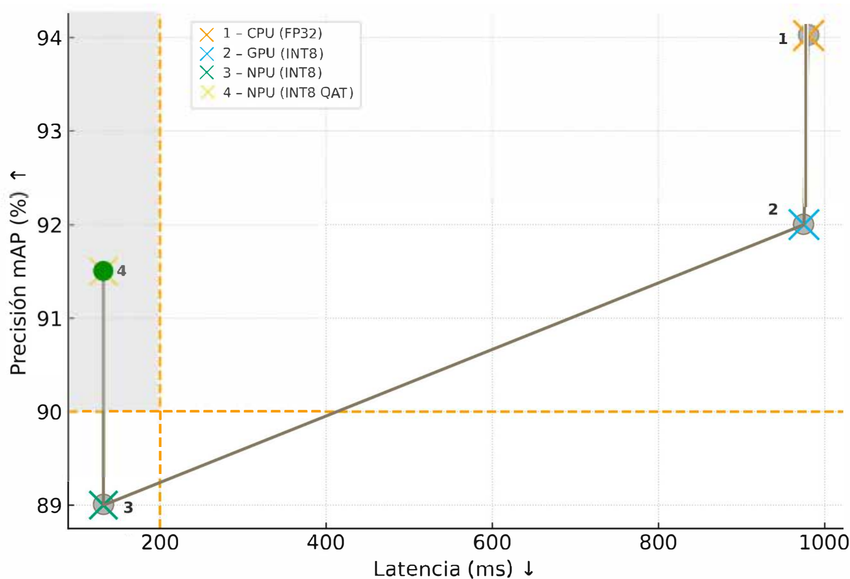

Figure 4. Trade-off Space Analysis.

1. CPU (FP32): This exhibits the highest accuracy (94% mAP) and highest consumption (5.1 W) but results in an unviable latency (980 ms), failing to meet L_{max}
2. GPU (FP32): Similar to the CPU, execution on the GPU also presents excessive latency (975 ms). This demonstrates that, for large models or non-optimized architectures, the runtime overhead and data movement nullify the parallel processing capability of the GPU, making it inefficient for the L_{max} requirement.
3. NPU (Initial INT8): The initial quantization to INT8 on the NPU achieves acceptable latency (132 ms) and efficient consumption (4.1 W). However, the measured accuracy of 89% mAP does not meet the requirement ($A_{max} \geq 90\%$ mAP).

This failure of the Success Condition immediately activates the Feedback Loop, mandating a return to Stage 3. In this iteration, QAT was applied to resolve the accuracy drop.

4. NPU (INT8 QAT—Final): The optimized configuration using QAT achieved the best multi-objective balance. The final operating point reached 132 ms latency, efficient consumption of 4.1 W, and a measured accuracy of 91.5% mAP (see Table 19).

The final configuration on the NPU (INT8 QAT) demonstrates the optimal convergence of the methodology, achieving a solution that is simultaneously feasible (meets all r requirements) and efficient in the design trade-off (see Figure 4). This operating point represents the best multi-objective balance and confirms that the optimization cycle robustly guides the design process toward the deployment of a viable solution in the edge

environment. Although this work focuses on a single detailed case study, the proposed five-stage methodology is general and can be applied to a wide range of edge deployment scenarios, including industrial inspection, embedded robotics, and autonomous mobile systems. Overall, the workflow provides a practical and empirically validated process that structures the entire decision-making pipeline—from requirement definition to model selection, optimization, hardware alignment, and final deployment.

7. Discussion and Future Work

Having reviewed the state-of-the-art techniques, architectures and platforms, this section discusses their broader implications. We will first explore the key real-world applications where these technologies are making a significant impact. Following this, we will analyze the emerging trends and future opportunities that are shaping the next generation of intelligent embedded systems. Finally, we will address the persistent open challenges—both technical and ethical—that must be overcome to realize the full potential of this field.

7.1. Applications of Neural Networks in Embedded Systems

Neural networks have revolutionized the field of embedded systems, allowing the implementation of advanced solutions in devices with limited resources.

CNNs have transformed artificial vision in embedded systems, enabling applications that require accurate and real-time analysis. Here are some examples:

- Object detection: In embedded systems, CNNs are used for object detection in real-time applications, such as autonomous driving systems and advanced driver-assistance systems (ADAS), to analyze the environment, identify objects like pedestrians, vehicles and traffic signs, and make decisions to ensure safe navigation [81].
- Image Classification and Segmentation: For example, deployed in medical devices to classify medical images or segment tumors accurately and efficiently [82].

Environmental sound recognition is an emerging application that combines spectral analysis techniques and neural networks to classify and detect acoustic events in real time. In embedded systems, a model based on CNNs uses spectrograms as input to classify sounds. This model, implemented in hardware such as FPGAs with tools such as High-Level Synthesis for Machine Learning (hls4ml), achieves significantly reduced inference times without compromising accuracy. This makes it ideal for embedded systems applications such as traffic monitoring and security systems, where real-time sound analysis is crucial [83].

For time series prediction, RNNs are employed to optimize resources in microcontroller-based home systems, enabling efficient and energy-saving deployment. This demonstrates the feasibility of implementing complex neural networks for predictive tasks even on resource-constrained devices [84].

Another significant application is vision in adverse conditions. Conditions such as fog, rain, or poor lighting pose a significant challenge for computer vision systems. However, recent advances, such as the Upgraded-YOLO model based on YOLOv5, have significantly improved performance [85]. This model uses data augmentation and architectural modifications to enhance the detection of small objects in low-visibility conditions, targeting edge devices with restricted resources.

7.2. Emerging Trends and Future Opportunities

The future of ML in embedded systems is being shaped by several innovative technologies. TinyML is a key emerging field focused on implementing models on low-power devices like microcontrollers and sensors. This approach enables local data processing,

which reduces latency, enhances privacy and eliminates the need for constant Internet connectivity [86]. This has revolutionized domains like healthcare and industrial automation and plays a crucial role in critical environments by enabling Out of Distribution (OOD) detection systems to identify anomalies in real time [87]. TinyML is already a practical reality for specific tasks such as keyword spotting (KWS) or simple sensor anomaly detection. However, its application to more complex vision tasks remains an active research challenge, with short-term/emerging feasibility, dependent on advances in the co-design of ultra-lightweight models and more powerful microcontrollers.

Another emerging trend is sub-4-bit quantization (4-, 3- and 2-bit), which promises substantial efficiency gains for ultra-constrained devices. However, its practical adoption is currently limited by several factors: significant accuracy degradation when reducing precision below 8 bits, the need for advanced QAT combined with knowledge distillation, and the lack of broad hardware support for sub-4-bit operators [88]. As a result, despite its strong theoretical efficiency, sub-4-bit quantization remains an emerging short- to medium-term trend with notable training and tooling barriers before widespread deployment becomes feasible [89].

Despite these advances in model-level efficiency, the practical adoption of such techniques in real systems depends not only on the model itself but also on the level of support provided by the surrounding hardware and software ecosystem, where significant limitations still persist.

The expansion of hardware has created a compatibility gap: software frameworks and ML compilers, such as Apache TVM or Glow, still struggle to efficiently optimize a model for the broad diversity of hardware accelerators (NPUs, FPGAs, MCUs). This limited maturity of the toolchain makes deployment a costly and manual process.

Furthermore, the lack of standardized and representative benchmarks remains a fundamental problem, as discussed in Section 3.3. Without a standard evaluation methodology, it is almost unmanageable to fairly compare solutions, which leads to over-optimization for specific tasks and hinders reproducibility. For trends such as on-device learning to mature, the research community must prioritize the development of more robust compilation frameworks and benchmark datasets that reflect real-world complexity.

The development of autonomous systems is also being driven by machine learning, which enables real-time decision-making and improves operational efficiency. Recent architectures based on edge computing optimize collaboration between devices, reducing reliance on cloud services and enhancing the privacy of locally processed data. This is transforming industrial and logistical processes through applications like predictive maintenance and autonomous energy management [90].

A related trend is Data Democratization and Data Fabric. Data democratization aims to make data more accessible within organizations, fostering collaboration in the development of ML solutions [91]. Data fabric supports this by integrating data from multiple sources (edge, cloud, on-premises) to provide unified access and reduce fragmentation [92], which is essential for applications in robotics and logistics.

These trends are more organizational and infrastructure-related than purely technical ones for the edge. Their adoption is ongoing, but the complete and secure integration of data from the edge to the cloud remains complex, with feasibility dependent on the maturity of IoT platforms and data management policies, presenting a variable timeframe [93].

Looking forward, On-Device Learning will allow embedded devices to learn and adapt in real-time, improving personalization by eliminating the need to send data to the cloud for training. This enhances privacy and allows devices to adapt to changes in the environment or user preferences [94]. Although promising for privacy and personalization, on-device learning faces significant barriers due to the high memory and computation

requirements for training (backpropagation). Current solutions are often limited to simple fine-tuning. Its commercial-scale feasibility is considered medium-term/in development, pending radically more memory-efficient optimization algorithms.

Neuromorphic Computing, with brain-inspired architectures like Spiking Neural Networks (SNNs), promises solutions with greater energy efficiency capable of handling complex tasks by emulating the parallel processing of the human brain [95]. This approach offers great impact potential, especially with SNNs. However, its adoption requires a paradigm shift in both hardware (specialized neuromorphic chips, still not massively available) and algorithms (SNN models and training methods). Its generalized integration is considered long-term/exploratory, contingent on the hardware and software ecosystem maturing and demonstrating clear advantages over conventional approaches in real-world applications.

Finally, the integration with the Internet of Things (IoT) will continue to transform key industries. This combination will enable networks of smart devices that share information and learn collaboratively, further improving efficiency and productivity across different sectors [14].

7.3. Open Challenges and Future Research Agenda

Despite promising trends, significant challenges must be addressed to ensure the successful integration of ML in embedded systems.

(a) Technical Challenges

The integration of neural networks in embedded systems poses a series of technical challenges [45]:

- Hardware limitation: It is essential to design models that can operate within the constraints of memory, power and energy consumption.
- Compatibility and Adaptability: Existing frameworks are often not optimized for embedded systems, requiring the development of specialized tools.
- Model optimization: Techniques such as quantization, pruning and knowledge distillation have proven to be effective in reducing model complexity without significantly compromising model accuracy.
- Robustness and Reliability: Systems must be able to operate under adverse conditions and in the face of malicious attacks.

(b) Ethical and Social Challenges

- Bias and Fairness: AI models can reflect and amplify biases present in their training data, leading to discriminatory outcomes.
- Privacy and Security: Protecting user privacy is a critical challenge when implementing AI in embedded systems. Models must also be secure against malicious attacks.
- Labor Impact: The automation of tasks through AI could have a significant impact on employment, requiring careful planning for workforce transitions.

A Future Research Agenda

Based on the technical and ethical challenges described, we synthesize the most critical areas requiring future research in the following agenda:

- Edge-Nativa Security and Robustness: Security challenges extend beyond data privacy. Urgent research is needed on the robustness of edge models against adversarial attacks specifically designed to exploit the weaknesses of quantized and pruned models, which may be more fragile than their 32-bit counterparts.
- Energy-Aware Neural Architecture Search (NAS): NAS methods, such as APQ, have proven effective for co-optimizing architecture, pruning, and quantization. However, these methods often optimize for latency or accuracy. The pending challenge is to

integrate power consumption (a pillar of edge computing) directly into the search cost function, creating a true "low-power" NAS that discovers architectures efficient in TOPS/W, not just fast ones.

- Holistic Metrics and Benchmarking: As highlighted in Section 3.3, the field lacks standardized evaluation. Efforts like MLPerf Tiny are a first step, but a research agenda is needed to create holistic benchmarks that evaluate performance under multiple simultaneous constraints (accuracy, latency, power, and memory). This is vital for objectively validating co-design methodologies, such as the one proposed in Section 6.
- Automation of Hardware–Software Co-Design: Overcoming hardware limitations and software fragmentation (such as the diversity of toolchains like TensorFlow Lite, eIQ Toolkit, or OpenVINO) requires automated co-design. Future research should focus on compilation frameworks that not only optimize the model for fixed hardware but also allow for flexible co-design. In this approach, the model architecture (like convolution kernels) and the accelerator micro-architecture (like buffer size) are jointly optimized. This research direction represents a deeper and more automated integration of the Model Selection (Stage 2) and Hardware/Ecosystem Selection (Stage 4) components of the proposed five-stage methodology.
- Memory-Efficient On-Device Learning: The trend of on-device learning offers great promise for adaptability and privacy. However, the main technical challenge is performing backpropagation and weight updating on devices with extremely limited RAM (in the KB range). New optimization algorithms are needed to train models with a near-zero memory footprint. This challenge directly aligns with the proposed methodology.

8. Conclusions

The integration of neural networks into embedded systems represents a transformative paradigm, driving the development of intelligent, real-time solutions for a host of critical and everyday applications. This systematic review has confirmed that successful deployment in resource-constrained environments is not dependent on a single factor, but rather on a holistic approach. The key lies in careful co-design and balance between model optimization techniques such as quantization and pruning, the selection of efficient specialized architectures like CNNs and compact networks and the choice of an appropriate hardware platform, from low-power MCUs to specialized accelerators like NPUs and Edge TPUs.

To bridge the gap between theory and practice, this paper's primary contribution is a structured five-stage methodology. This framework provides developers with a practical, iterative guide to navigate the complexities of model selection, optimization and validation, ensuring that the final solution meets demanding real-world constraints.

While emerging trends like TinyML and neuromorphic computing promise an even more powerful and efficient future, significant persistent challenges in energy efficiency, model robustness and security must continue to be addressed. Continued research in these areas will be essential to unlock the full potential of intelligent systems at the edge.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/electronics14244877/s1, File S1: PRISMA 2020 Checklist (PDF).

Author Contributions: Conceptualization, Á.G. and D.A.; methodology, R.C.-C.; validation, R.C.-C.; formal analysis, R.C.-C. and Á.G.; investigation, R.C.-C., D.A. and Á.G.; resources, Á.G. and D.A.; data curation, R.C.-C.; writing—original draft preparation, R.C.-C.; writing—review and editing,

Á.G.; visualization, R.C.-C.; supervision, Á.G.; project administration, D.A.; funding acquisition, Á.G. and D.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by grant number IND2023/TIC-27863 of the Comunidad de Madrid under the 2023 Ayudas para la Realización de Doctorados Industriales.

Data Availability Statement: No new data were created or analyzed in this study.

Conflicts of Interest: Authors Ruth Cordova-Cardenas and Daniel Amor were employed by the company RBZ Robot Design. The remaining author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Zheng, Z.; Li, Y.; Chen, J.; Zhou, P.; Chen, X.; Liu, Y. Threshold Neuron: A Brain-Inspired Artificial Neuron for Efficient On-Device Inference. *arXiv* **2025**, arXiv:2412.13902.
2. Mishra, A.; Cha, J.; Park, H.; Kim, S. (Eds.) *Artificial Intelligence and Hardware Accelerators*; Springer International Publishing: Cham, Switzerland, 2023; ISBN 978-3-031-22169-9.
3. Zhu, S.; Yu, T.; Xu, T.; Chen, H.; Dustdar, S.; Gigan, S.; Gunduz, D.; Hossain, E.; Jin, Y.; Lin, F.; et al. Intelligent Computing: The Latest Advances, Challenges, and Future. *Intell. Comput.* **2023**, *2*, 6. [CrossRef]
4. Tang, Q.; Yu, F.R.; Xie, R.; Boukerche, A.; Huang, T.; Liu, Y. Internet of Intelligence: A Survey on the Enabling Technologies, Applications, and Challenges. *IEEE Commun. Surv. Tutorials* **2022**, *24*, 1394–1434. [CrossRef]
5. Gorospe, J.; Mulero, R.; Arbelaitz, O.; Muguerza, J.; Antón, M.Á. A Generalization Performance Study Using Deep Learning Networks in Embedded Systems. *Sensors* **2021**, *21*, 1031. [CrossRef]
6. Chen, Y.; Zheng, B.; Zhang, Z.; Wang, Q.; Shen, C.; Zhang, Q. Deep Learning on Mobile and Embedded Devices: State-of-the-Art, Challenges, and Future Directions. *ACM Comput. Surv.* **2021**, *53*, 1–37. [CrossRef]
7. Zheng, H.; Duan, J.; Dong, Y.; Liu, Y. Real-Time Fire Detection Algorithms Running on Small Embedded Devices Based on MobileNetV3 and YOLOv4. *Fire Ecol.* **2023**, *19*, 31. [CrossRef]
8. Page, M.J.; McKenzie, J.E.; Bossuyt, P.M.; Boutron, I.; Hoffmann, T.C.; Mulrow, C.D.; Shamseer, L.; Tetzlaff, J.M.; Akl, E.A.; Brennan, S.E.; et al. The PRISMA 2020 Statement: An Updated Guideline for Reporting Systematic Reviews. *Syst. Rev.* **2021**, *10*, 89. [CrossRef]
9. Gawlikowski, J.; Tassi, C.R.N.; Ali, M.; Lee, J.; Humt, M.; Feng, J.; Kruspe, A.; Triebel, R.; Jung, P.; Roscher, R.; et al. A Survey of Uncertainty in Deep Neural Networks. *Artif. Intell. Rev.* **2023**, *56*, 1513–1589. [CrossRef]
10. Padilla, D.; Rashwan, H.A.; Puig, D.S. On Determining Suitable Embedded Devices for Deep Learning Models. In *Frontiers in Artificial Intelligence and Applications*; Villaret, M., Alsinet, T., Fernández, C., Valls, A., Eds.; IOS Press: Amsterdam, The Netherlands, 2021; ISBN 978-1-64368-210-5.
11. Biglari, A.; Tang, W. A Review of Embedded Machine Learning Based on Hardware, Application, and Sensing Scheme. *Sensors* **2023**, *23*, 2131. [CrossRef]
12. Roth, W.; Schindler, G.; Klein, B.; Peharz, R.; Tschiatschek, S.; Fröning, H.; Pernkopf, F.; Ghahramani, Z. Resource-Efficient Neural Networks for Embedded Systems. *J. Mach. Learn. Res.* **2024**, *25*, 1–51.
13. Liang, H.; Fan, Z.; Sarkar, R.; Jiang, Z.; Chen, T.; Zou, K.; Cheng, Y.; Hao, C.; Wang, Z. M3ViT: Mixture-of-Experts Vision Transformer for Efficient Multi-Task Learning with Model-Accelerator Co-Design. *Adv. Neural. Inf. Process. Syst.* **2022**, *35*, 28441–28457.
14. Abo, M.D. An Efficiency Comparison of NPU, CPU, and GPU When Executing an Object Detection Model YOLOv5. Bachelor's Thesis, School of Electrical Engineering and Computer Science (EECS), University Park, PA, USA, 2024.
15. Hot Tech Vision and Analysis (HTVA). *AI Accelerators for Machine Vision Competitive Analysis and Review: Developer Experience And Performance Evaluation*; Hot Tech Vision and Analysis: Chepachet, RI, USA, 2025; p. 19.
16. Banbury, C.; Reddi, V.J.; Torelli, P.; Holleman, J.; Jeffries, N.; Kiraly, C.; Montino, P.; Kanter, D.; Ahmed, S.; Pau, D.; et al. MLPerf Tiny Benchmark. *arXiv* **2021**, arXiv:2106.07597.
17. Baller, S.P.; Jindal, A.; Chadha, M.; Gerndt, M. DeepEdgeBench: Benchmarking Deep Neural Networks on Edge Devices. In Proceedings of the 2021 IEEE International Conference on Cloud Engineering (IC2E), Virtual, 4–8 October 2021.
18. Cantero, D.; Esnaola-Gonzalez, I.; Miguel-Alonso, J.; Jauregi, E. Benchmarking Object Detection Deep Learning Models in Embedded Devices. *Sensors* **2022**, *22*, 4205. [CrossRef]
19. Wang, T.; Wang, K.; Cai, H.; Lin, J.; Liu, Z.; Wang, H.; Lin, Y.; Han, S. APQ: Joint Search for Network Architecture, Pruning and Quantization Policy. In Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 14–19 June 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 2075–2084.

20. Pham, P.; Abraham, J.A.; Chung, J. Training Multi-Bit Quantized and Binarized Networks with a Learnable Symmetric Quantizer. *IEEE Access* **2021**, *9*, 47194–47203. [CrossRef]
21. Gou, J.; Yu, B.; Maybank, S.J.; Tao, D. Knowledge Distillation: A Survey. *Int. J. Comput. Vis.* **2021**, *129*, 1789–1819. [CrossRef]
22. Sandra, C.P.; Pedro, V.-L.; Antonio, T.D.; Leonel; Rocío, C.S. *Developing Linear Algebra Codes on Modern Processors: Emerging Research and Opportunities: Emerging Research and Opportunities*; IGI Global: Hershey, PA, USA, 2022; ISBN 978-1-7998-7084-5.
23. Luo, X.; Wang, H.; Wu, D.; Chen, C.; Deng, M.; Huang, J.; Hua, X.-S. A Survey on Deep Hashing Methods. *ACM Trans. Knowl. Discov. Data* **2023**, *17*, 1–50. [CrossRef]
24. Hawks, B.; Duarte, J.; Fraser, N.J.; Pappalardo, A.; Tran, N.; Umuroglu, Y. Ps and Qs: Quantization-Aware Pruning for Efficient Low Latency Neural Network Inference. *Front. Artif. Intell.* **2021**, *4*, 676564. [CrossRef] [PubMed]
25. Fahim, F.; Hawks, B.; Herwig, C.; Hirschauer, J.; Jindariani, S.; Tran, N.; Carloni, L.P.; Guglielmo, G.D.; Harris, P.; Krupa, J.; et al. Hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices. *arXiv* **2021**, arXiv:2103.05579.
26. An, S.; Shin, J.; Kim, J. Quantization-Aware Training With Dynamic and Static Pruning. *IEEE Access* **2025**, *13*, 57476–57484. [CrossRef]
27. Xu, J.; Tan, X.; Luo, R.; Song, K.; Li, J.; Qin, T.; Liu, T.-Y. NAS-BERT: Task-Agnostic and Adaptive-Size BERT Compression with Neural Architecture Search. In Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining, Singapore, 14–18 August 2021; pp. 1933–1943.
28. Akkad, G.; Mansour, A.; Inaty, E. Embedded Deep Learning Accelerators: A Survey on Recent Advances. *IEEE Trans. Artif. Intell.* **2024**, *5*, 1954–1972. [CrossRef]
29. XLA. Optimización del Compilador Para el Aprendizaje Automático. Available online: https://www.tensorflow.org/xla?hl=es-419 (accessed on 7 February 2025).
30. Morera, L.M.M.; Por, T.; Cabrera, J.J.H.; Vieira, A.H. Implementación de APIs Comerciales de OpenAI Mediante Modelos Open Source. Bachelor's Thesis, Universidad de Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, Spain, 2024.
31. Park, Y.; Budhathoki, K.; Chen, L.; Kübler, J.M.; Huang, J.; Kleindessner, M.; Huan, J.; Cevher, V.; Wang, Y.; Karypis, G. Inference Optimization of Foundation Models on AI Accelerators. In Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Barcelona, Spain, 25–29 August 2024; ACM: New York, NY, USA, 2024; pp. 6605–6615.
32. Lepri, F.; Khaddam-Aljameh, R.; Ielmini, D.; Indiveri, G.; Eleftheriou, E. In-Memory Computing for Machine Learning and Deep Learning. *IEEE J. Explor. Solid-State Comput. Devices Circuits* **2023**, *9*, 1–17. [CrossRef]
33. Marwedel, P. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*; Embedded Systems; Springer International Publishing: Cham, Switzerland, 2021; ISBN 978-3-030-60909-2.
34. Alzubaidi, L.; Zhang, J.; Humaidi, A.J.; Al-Dujaili, A.; Duan, Y.; Al-Shamma, O.; Santamaría, J.; Fadhel, M.A.; Al-Amidie, M.; Farhan, L. Review of Deep Learning: Concepts, CNN Architectures, Challenges, Applications, Future Directions. *J. Big. Data* **2021**, *8*, 53. [CrossRef] [PubMed]
35. Terven, J.; Córdova-Esparza, D.-M.; Romero-González, J.-A. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Make* **2023**, *5*, 1680–1716. [CrossRef]
36. Techzizou YOLOv4 VS YOLOv4-Tiny. *Analytics Vidhya*. 2024. Available online: https://medium.com/analytics-vidhya/yolov4-vs-yolov4-tiny-97932b6ec8ec (accessed on 16 September 2025).
37. Shuvo, M.M.H.; Islam, S.K.; Cheng, J.; Morshed, B.I. Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review. *Proc. IEEE* **2023**, *111*, 42–91. [CrossRef]
38. Chen, L.; Lin, S.; Lu, X.; Cao, D.; Wu, H.; Guo, C.; Liu, C.; Wang, F.-Y. Deep Neural Network Based Vehicle and Pedestrian Detection for Autonomous Driving: A Survey. *IEEE Trans. Intell. Transport. Syst.* **2021**, *22*, 3234–3246. [CrossRef]
39. Padilla, R.; Netto, S.L.; da Silva, E.A.B. A Survey on Performance Metrics for Object-Detection Algorithms. *Remote Sens.* **2021**, *13*, 555.
40. Alif, M.A.R. YOLOv11 for Vehicle Detection: Advancements, Performance, and Applications in Intelligent Transportation Systems. *arXiv* **2024**, arXiv:2410.22898. [CrossRef]
41. Jiang, P.; Ergu, D.; Liu, F.; Cai, Y.; Ma, B. A Review of Yolo Algorithm Developments. *Procedia Comput. Sci.* **2022**, *199*, 1066–1073. [CrossRef]
42. Mehta, S.; Rastegari, M. MobileViT: Light-Weight, General-Purpose, and Mobile-Friendly Vision Transformer. *arXiv* **2022**, arXiv:2110.02178.
43. Zhao, X.; Song, Y. Improved Ship Detection with YOLOv8 Enhanced with MobileViT and GSConv. *Electronics* **2023**, *12*, 4666. [CrossRef]
44. Maaz, M.; Shaker, A.; Cholakkal, H.; Khan, S.; Zamir, S.W.; Anwer, R.M.; Khan, F.S. EdgeNeXt: Efficiently Amalgamated CNN-Transformer Architecture for Mobile Vision Application. In Proceedings of the European conference on computer vision, Tel Aviv, Israel, 23–27 October 2022.

45. Rezk, N.M.; Purnaprajna, M.; Nordstrom, T.; Ul-Abdin, Z. Recurrent Neural Networks: An Embedded Computing Perspective. *IEEE Access* **2020**, *8*, 57967–57996. [CrossRef]
46. Mienye, I.D.; Swart, T.G.; Obaido, G. Recurrent Neural Networks: A Comprehensive Review of Architectures, Variants, and Applications. *Information* **2024**, *15*, 517. [CrossRef]
47. Bustán Andrade, P.A.; Carrión Zamora, P.E.; Chamba Lara, A.E.; Pazmiño Piedra, J.P. A comprehensive evaluation of ai techniques for air quality index prediction: RNNs and transformers. *Ingenius* **2025**, *33*, 60–75. [CrossRef]
48. Sanford, C.; Hsu, D.; Telgarsky, M. Representational Strengths and Limitations of Transformers. *arXiv* **2024**, arXiv:2402.09268.
49. Dosovitskiy, A.; Beyer, L.; Kolesnikov, A.; Weissenborn, D.; Zhai, X.; Unterthiner, T.; Dehghani, M.; Minderer, M.; Heigold, G.; Gelly, S.; et al. An Image Is Worth 16×16 Words: Transformers for Image Recognition at Scale. *Sensors* **2021**, *21*, 6895.
50. Panopoulos, I.; Nikolaidis, S.; Venieris, S.I.; Venieris, I.S. Exploring the Performance and Efficiency of Transformer Models for NLP on Mobile Devices. In Proceedings of the 2023 IEEE Symposium on Computers and Communications (ISCC), Gammarth, Tunisia, 9–12 July 2023.
51. Lalapura, V.; Joseph, A.; Satheesh, H. Recurrent Neural Networks for Edge Intelligence: A Survey. *ACM Comput. Surv.* **2021**, *54*, 1–38. [CrossRef]
52. Jayanth, R.; Gupta, N.; Prasanna, V. Benchmarking Edge AI Platforms for High-Performance ML Inference. In Proceedings of the 2024 IEEE High Performance Extreme Computing Conference (HPEC), Wakefield, MA, USA, 23–27 September 2024.
53. Zhang, P.; Zeng, G.; Wang, T.; Lu, W. TinyLlama: An Open-Source Small Language Model. *arXiv* **2024**, arXiv:2401.02385.
54. Wieder, O.; Kohlbacher, S.; Kuenemann, M.; Garon, A.; Ducrot, P.; Seidel, T.; Langer, T. A Compact Review of Molecular Property Prediction with Graph Neural Networks. *Drug Discov. Today Technol.* **2020**, *37*, 1–12. [CrossRef] [PubMed]
55. Zhang, L.; Bao, C.; Ma, K. Self-Distillation: Towards Efficient and Compact Neural Networks. *IEEE Trans. Pattern Anal. Mach. Intell.* **2021**, *44*, 4388–4403. [CrossRef] [PubMed]
56. Yun, S.; Park, J.; Lee, K.; Shin, J. Regularizing Class-Wise Predictions via Self-Knowledge Distillation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 14–19 June 2020; pp. 13873–13882. [CrossRef]
57. Gao, G.; Liu, Z.; Zhang, G.; Li, J.; Qin, A. DANet: Semi-supervised differentiated auxiliaries guided network for video action recognition. *Neural Netw.* **2023**, *158*, 121–131. [CrossRef] [PubMed]
58. Zhang, Z.; Feng, F.; Huang, T. FNNS: An Effective Feedforward Neural Network Scheme with Random Weights for Processing Large-Scale Datasets. *Appl. Sci.* **2022**, *12*, 12478. [CrossRef]
59. Novickis, R.; Justs, D.J.; Ozols, K.; Greitāns, M. An Approach of Feed-Forward Neural Network Throughput-Optimized Implementation in FPGA. *Electronics* **2020**, *9*, 2193. [CrossRef]
60. Liu, X.; Xu, W.; Wang, Q.; Zhang, M. Energy-Efficient Computing Acceleration of Unmanned Aerial Vehicles Based on a CPU/FPGA/NPU Heterogeneous System. *IEEE Internet Things J.* **2024**, *11*, 27126–27138. [CrossRef]
61. Sankaran, A.; Mastropietro, O.; Saboori, E.; Idris, Y.; Sawyer, D.; AskariHemmat, M.; Boukli Hacene, G. Deeplite NeutrinoTM: A BlackBox Framework for Constrained Deep Learning Model Optimization. *AAAI* **2021**, *35*, 15166–15174. [CrossRef]
62. Learn the Architecture–Optimizing C Code with Neon Intrinsics. Available online: https://developer.arm.com/documentation/102467/0201/Why-use-Neon-intrinsics-?lang=en (accessed on 28 February 2025).
63. TensorFlow Lite. Available online: https://www.tensorflow.org/lite/guide?hl=es-419 (accessed on 3 December 2024).
64. ONNX. Runtime | Home. Available online: https://onnxruntime.ai/ (accessed on 3 December 2024).
65. Kwon, Y.; Cha, J.H.; Lee, J.; Yu, M.; Park, J.; Lee, J. ACLTuner: A Profiling-Driven Fast Tuning to Optimized Deep Learning Inference. In Proceedings of the Machine Learning for Systems Workshop at NeurIPS, New Orleans, MI, USA, 28 November–9 December 2022.
66. Manor, E.; Greenberg, S. Custom Hardware Inference Accelerator for TensorFlow Lite for Microcontrollers. *IEEE Access* **2022**, *10*, 73484–73493. [CrossRef]
67. Wulfert, L.; Kühnel, J.; Krupp, L.; Viga, J.; Wiede, C.; Gembaczka, P.; Grabmaier, A. AIfES: A Next-Generation Edge AI Framework. *IEEE Trans. Pattern Anal. Mach. Intell.* **2024**, *46*, 4519–4533. [CrossRef]
68. CMSIS-NN. CMSIS NN Software Library. Available online: https://arm-software.github.io/CMSIS-NN/latest/ (accessed on 3 December 2024).
69. PyTorch. Available online: https://pytorch.org/ (accessed on 3 December 2024).
70. Hamdi, M.A.; Daghero, F.; Sarda, G.M.; Delm, J.V.; Symons, A.; Benini, L.; Verhelst, M.; Pagliari, D.J.; Burrello, A. MATCH: Model-Aware TVM-Based Compilation for Heterogeneous Edge Devices. In *Transactions on Computer-Aided Design of Integrated Circuits and Systems*; IEEE: Piscataway, NJ, USA, 2024.
71. Immonen, R.; Hämäläinen, T. Tiny Machine Learning for Resource-Constrained Microcontrollers. *J. Sens.* **2022**, *2022*, 1–11. [CrossRef]
72. Machupalli, R.; Hossain, M.; Mandal, M. Review of ASIC Accelerators for Deep Neural Networks. *Microprocess. Microsyst.* **2022**, *89*, 104441. [CrossRef]

73. Prabu, T.; Srinivasan, K. Design and Implementation of High-Performance FPGA Accelerator for Non-Separable Discrete Fourier Transform Optimizing Real-Time Image and Video Processing. *J. Nanoelectron. Optoelectron.* **2024**, *19*, 843–856. [CrossRef]
74. Messaoud, S.; Bouaafia, S.; Maraoui, A.; Ammari, A.C.; Khriji, L.; Machhout, M. Deep Convolutional Neural Networks-Based Hardware–Software on-Chip System for Computer Vision Application. *Comput. Electr. Eng.* **2022**, *98*, 107671. [CrossRef]
75. Yu, J.; De Antonio, A.; Villalba-Mora, E. Deep Learning (CNN, RNN) Applications for Smart Homes: A Systematic Review. *Computers* **2022**, *11*, 26. [CrossRef]
76. Ali, M.L.; Zhang, Z. The YOLO Framework: A Comprehensive Review of Evolution, Applications, and Benchmarks in Object Detection. *Computers* **2024**, *13*, 336. [CrossRef]
77. Novac, P.-E.; Boukli Hacene, G.; Pegatoquet, A.; Miramond, B.; Gripon, V. Quantization and Deployment of Deep Neural Networks on Microcontrollers. *Sensors* **2021**, *21*, 2984. [CrossRef] [PubMed]
78. Berthelier, A.; Chateau, T.; Duffner, S.; Garcia, C.; Blanc, C. Deep Model Compression and Architecture Optimization for Embedded Systems: A Survey. *J. Sign Process. Syst.* **2021**, *93*, 863–878. [CrossRef]
79. Dhilleswararao, P.; Boppu, S.; Manikandan, M.S.; Cenkeramaddi, L.R. Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey. *IEEE Access* **2022**, *10*, 131788–131828. [CrossRef]
80. Alam, S.; Yakopcic, C.; Wu, Q.; Barnell, M.; Khan, S.; Taha, T.M. Survey of Deep Learning Accelerators for Edge and Emerging Computing. *Electronics* **2024**, *13*, 2988. [CrossRef]
81. Nidamanuri, J.; Nibhanupudi, C.; Assfalg, R.; Venkataraman, H. A Progressive Review: Emerging Technologies for ADAS Driven Solutions. *IEEE Trans. Intell. Veh.* **2022**, *7*, 326–341. [CrossRef]
82. Sarmiento-Ramos, J.L. Aplicaciones de las redes neuronales y el deep learning a la ingeniería biomédica. *Rev. UIS Ing.* **2020**, *19*, 1–18. [CrossRef]
83. Vandendriessche, J.; Wouters, N.; Da Silva, B.; Lamrini, M.; Chkouri, M.Y.; Touhafi, A. Environmental Sound Recognition on Embedded Systems: From FPGAs to TPUs. *Electronics* **2021**, *10*, 2622. [CrossRef]
84. Vanting, N.B.; Ma, Z.; Jørgensen, B.N. A Scoping Review of Deep Neural Networks for Electric Load Forecasting. *Energy Inf.* **2021**, *4*, 49. [CrossRef]
85. Delleji, T.; Slimeni, F.; Fekih, H.; Jarray, A.; Boughanmi, W.; Kallel, A.; Chtourou, Z. An Upgraded-YOLO with Object Augmentation: Mini-UAV Detection Under Low-Visibility Conditions by Improving Deep Neural Networks. *Oper. Res. Forum* **2022**, *3*, 60. [CrossRef]
86. Capogrosso, L.; Cunico, F.; Cheng, D.S.; Fummi, F.; Cristani, M. A Machine Learning-Oriented Survey on Tiny Machine Learning. *IEEE Access* **2024**, *12*, 23406–23426. [CrossRef]
87. Saraan, A.; Hussain, N.; Zahara, S.M. Tiny Machine Learning (TinyML) Systems. Preprint, ResearchGate, December 2024. Available online: https://www.researchgate.net/publication/386579238 (accessed on 4 December 2025).
88. Park, E.; Yoo, S. PROFIT: A Novel Training Method for Sub-4-Bit MobileNet Models. In Proceedings of the European Conference on Computer Vision, Glasgow, UK, 26–28 August 2020.
89. Liu, H.-I.; Galindo, M.; Xie, H.; Wong, L.-K.; Shuai, H.-H.; Li, Y.-H.; Cheng, W.-H. Lightweight Deep Learning for Resource-Constrained Environments. *ACM Comput. Surv.* **2024**, *56*, 1–42. [CrossRef]
90. Delgado, J.R. Sistema Autónomo de Generación de Energía Renovable. In *Memorias de las XVII Jornadas de Conferencias de Ingeniería Electrónica*; Universidad Politécnica de Cataluña: Terrassa, Spain, 2011.
91. Dinten, R.; Zorrilla, M. *Laredo: Democratización de Análisis de Flujos de Datos Para el Mantenimiento Predictivo*; Universidad de Cantabria: Cantabria, Spain, 2023.
92. Rieyan, S.A.; News, M.R.K.; Rahman, A.B.M.M.; Khan, S.A.; Zaarif, S.T.J.; Alam, M.G.R.; Hassan, M.M.; Ianni, M.; Fortino, G. An Advanced Data Fabric Architecture Leveraging Homomorphic Encryption and Federated Learning. *Inf. Fusion* **2024**, *102*, 102004. [CrossRef]
93. Kuznetsov, M.; Novikova, E.; Kotenko, I.; Doynikova, E. Privacy Policies of IoT Devices: Collection and Analysis. *Sensors* **2022**, *22*, 1838. [CrossRef] [PubMed]
94. Ureña, A.C.; González Calero, P.A. Aprendizaje profundo en IoT: Redes neuronales convolucionales con imágenes aplicadas a un vehículo autónomo. Master's Thesis, Universidad Complutense de Madrid, Madrid, Spain, 2021.
95. García, J.G.; Vilda, C.C.; Alonso, D.P. *Avances Algorítmicos Aplicados al Procesamiento de Información en el Campo de la Visión Artificial Basada en Eventos Para Sistemas Bioinspirados*; Universidad Rey Juan Carlos: Madrid, Spain, 2023.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.