Dean&Francis
ISSN 2959-6157

Toward Real-Time and Efficient Edge Intelligence: Advances and Challenges in Lightweight Machine Learning

Xiang Gao

Communication Engineering, Xidian University, Xi'an, China
24012100008@stu.xidian.edu.cn

Abstract:

Deploying advanced Machine Learning (ML), particularly Deep Neural Networks (DNNs), on resource-constrained edge devices is crucial for realizing low-latency, privacy-preserving, and reliable edge intelligence applications. However, a significant gap exists between the high computational, memory, and energy demands of state-of-the-art models and the severe limitations inherent to edge hardware. This review systematically analyzes the field of lightweight ML for edge devices, aiming to bridge this gap. Methods: Focusing on the inference phase, the review critically examines three primary technical pillars: (1) Model Compression techniques, including knowledge distillation, network pruning (structured and unstructured), and quantization; (2) Efficient Neural Architecture Design of inherently compact models (e.g., MobileNet, ShuffleNet, EfficientNet series); and (3) Hardware-aware Optimization and Adaptation, encompassing operator fusion, dedicated inference engines, and leveraging heterogeneous systems. Results and Conclusion: The analysis highlights key achievements in reducing model size, complexity, and latency while maintaining accuracy. However, fundamental challenges persist, including the accuracy-efficiency tradeoff, hardware fragmentation, the memory wall bottleneck, and privacy/security concerns during deployment. Emerging solutions like neural-symbolic learning, adaptive federated learning, hardware-aware Neural Architecture Search (NAS), Processing-in-Memory (PIM) accelerators, and cross-stack co-design frameworks represent promising future directions. Overcoming these challenges is strategically vital for unlocking the full potential of ubiquitous, real-time edge intelligence.

Keywords: Machine Learning; knowledge distillation; Lightweight model.

ISSN 2959-6157

1. Introduction

In recent years, breakthroughs in Artificial Intelligence (AI), machine Learning (ML), notably, is instigating profound transformations within all sectors, driving an extensive array of applications from mechanized decision processes to cognitive perception systems. Parallel to this, the explosive growth of the Internet of Things (IoT), coupled with intensifying demands for intelligence in real-time, is fueling the rise of edge computing as a defining paradigm. The core objective of this paradigm lies in the strategic deployment of data processing and analytics capabilities from centralized cloud environments to the network's periphery, thereby situating them nearer to the loci of data origination. Deploying powerful ML capabilities in smartphones, sensors, cameras, wearable devices, industrial controllers, and autonomous vehicles is an Advanced Driving Assistance System (ADAS) computational segmentation technology that achieves low-latency response, enhances user privacy (data localization processing), reduces network bandwidth pressure, improves system reliability (offline operation capability), and unlocks new real-time intelligent application scenarios (such as industrial predictive maintenance, real-time health monitoring, and autonomous driving perception) [1].

On the other hand, mainstream, high-performance ML/DNN models usually have a large number of parameters (millions to billions), extremely high computational complexity (requiring powerful GPUs/TPUs), and huge memory/storage consumption [2]. This creates a huge gap with the severe resource constraints that are prevalent in edge devices, including limited computing power (CPU/GPU/NPU performance), scarce memory (RAM), insufficient storage space (Flash), and tight power budgets (battery capacity/thermal limitations). This mismatch between resource constraints and model requirements has seriously hindered the widespread adoption and in-depth application of intelligence at the edge [3].

To address these core contradictions, Lightweight Machine Learning for Edge Devices has emerged and is quickly becoming a dynamic and vital field of research and practice. The pivotal objective inherent to this domain resides in architecting, refining, and implementing machine learning models and systems capable of sustaining efficient operation under the stringent resource limitations characteristic of edge computing environments — encompassing rigorous constraints on execution velocity, power consumption, and memory utilization — whilst concurrently preserving the paramount degree of predictive precision attainable. Achieving this goal is strategic to unlock the full potential of edge intelligence, drive AI inclusion, and build truly real-time, reliable, privacy-preserving intelligent systems. Achieving lightweight machine learning on edge devices is a multi-dimensional and cross-level challenge, and its key technical paths mainly include:

Model Compression: Streamlining pre-trained models, including Knowledge Distillation like the implementation of dynamic network surgery demands substantial computational commitment, manifesting as 700,000 iterations of gradient-descent convergence, to effectuate a 17.7x diminution in parameter cardinality while imposing no statistically significant degradation in classification efficacy relative to the uncompressed archetype [4], Network Pruning [5], Quantization (reducing weight/activation numerical precision), Low-rank/Tensor Decomposition, and other techniques. It is designed to significantly reduce model size, computational effort, and memory footprint.

Efficient Neural Architecture Design: Directly design lightweight and hardware-friendly model architectures, such as MobileNet series, EfficientNet series, ShuffleNet, SqueezeNet, etc., and use deep separable convolutions, channel shuffling and other operations to ensure high efficiency at the source.

Hardware-aware Optimization &&& Adaptation: Closely combine the characteristics of the target edge hardware (such as specific CPU instruction set, GPU architecture, dedicated neural processing unit NPU/APU/AI accelerator, FPGA, etc.) for model optimization, operator acceleration. For example, Cuong Pham-Quoc and his team used the MobileNet CNN model to implement a prototype version on an FPGA-based MPSoC platform, which was 69.4x and 4.67x faster than the quad-core ARM Cortex-A53 processor and Intel Core i7 CPU, respectively and inference engine customization (such as TensorFlow Lite, whose fundamental design objective centers on lowering barriers to deploying and executing machine learning capabilities directly upon endpoint devices positioned at the network periphery, thereby obviating the need for iterative upstream-downstream communication involving device-generated data and cloud/core computing resources [6, 7], ONNX Runtime, TVM, MNN, etc.). Maximize the use of hardware computing power and improve energy efficiency.

This review paper aims to systematically sort out and critically analyze the core technologies and latest progress of lightweight machine learning for edge devices. This review will focus on the key technologies for lightweighting in the model inference phase (rather than training), especially the latest methods, evaluation metrics, representative work, and performance on real-world edge platforms in the three pillars of model compression, efficient architecture design, and hardware-aware optimization. By integrating a wide range of research from academia and industry, this paper aims to: 1) To provide researchers and

practitioners with a comprehensive technical panorama and development context in the field; 2) In-depth analysis of the advantages, limitations, application scenarios and interrelationships of different technical routes; 3) Summarize the main challenges and open issues at hand; 4) Discuss future promising research directions and development trends.

2. Method

2.1 Model Compression-Based Algorithms

Model compression techniques target pre-trained models, aiming to significantly reduce their size, computational complexity, and memory footprint while preserving as much of their original accuracy as possible [4, 5]. These techniques are typically applied post-training or integrated into a fine-tuning stage.

2.1.1 Knowledge distillation

Knowledge Distillation (KD) facilitates the transfer of knowledge from a large, complex, and highly accurate "teacher" model to a smaller, simpler "student" model [8]. The core principle extends beyond merely replicating the teacher's final class predictions (hard targets). Instead, the student learns from the teacher's "softened" output probability distributions, which encaps richer information about inter-class relationships and decision boundaries learned by the more complex model [8].

Response-Based Distillation (Logits Matching): This prevalent form minimizes the Kullback-Leibler (KL) divergence between the student's output logits (pre-softmax activations) and the softened logits of the teacher [8]. Softening the teacher's output is achieved using a temperature parameter (T > 1) within the softmax function:

$$p(z,T)_i = \frac{exp(z_i / T)}{\sum_{j=1}^{C} exp(z_j / T)} \quad (1)$$

where z is the logit vector and i indexes classes. Increasing T produces smoother, less confident probability distributions, revealing the relative similarities between classes as perceived by the teacher. The distillation loss (L_{KD}) is then calculated as:

$$L_{KD} = T^2 \cdot D_{KL}\left(p(z_t,T) \,\middle|\, p(z_s,T)\right) \quad (2)$$

Here, $\mathbf{z_t}$ and $\mathbf{z_s}$ represent the teacher and student logits, respectively. The T^2 factor compensates for the scaling effect on gradients introduced by the temperature parameter during optimization [8]. The student›s overall training objective combines this distillation loss with the standard cross-entropy loss (L_{CE}) computed using the ground-truth labels (hard targets):

$$L_{total} = \alpha * L_{KD} + (1-\alpha) * L_{CE} \quad (3)$$

The hyperparameter α (typically between 0.5 and 0.9) balances the influence of the teacher's knowledge (L_{KD}) and the true labels (L_{CE}) [8]. During inference, the student model operates using the standard softmax function (T = 1), incurring no computational overhead from the distillation process.

Feature-Based Distillation: This approach guides the student to mimic the teacher's internal representations beyond the final output layer [8]. This involves aligning intermediate feature maps or attention mechanisms at specific, often corresponding, layers within the teacher and student networks. Loss functions such as Mean Squared Error (MSE) or Cosine Similarity are employed to minimize the discrepancy between these intermediate activations [8]. Capturing this internal state often leads to superior student performance, particularly when the student architecture differs substantially from the teacher's. Example: The TinyBERT framework effectively compresses large transformer-based language models (like BERT) for edge deployment by utilizing multi-layer feature distillation, enabling performance close to the teacher model with only a fraction of the parameters [8].

2.1.2 Network pruning

Pruning aims to identify and remove redundant or less critical parameters (individual weights) or structural units (entire neurons, filters, or channels) from a neural network [5, 9, 10]. The goal is to generate a sparser, smaller model that retains functional accuracy, thereby reducing inference latency and memory requirements.

Unstructured Pruning: This method removes individual weights based on specific criteria, most commonly the smallest absolute magnitudes (L1 norm) [5]. While unstructured pruning can achieve very high compression ratios, it results in irregular, non-structured sparsity patterns within weight matrices. Exploiting this sparsity for computational gains on standard hardware (CPUs, GPUs) typically requires specialized sparse linear algebra libraries and may not yield significant speedups without dedicated hardware support for sparse computations [5].

Structured Pruning: This technique removes entire structural components, such as convolutional filters or channels, or neurons/heads in transformers [9, 10]. This results in dense, smaller sub-networks that align naturally with standard hardware architectures, enabling direct computational speedups without requiring specialized sparse operations.

Norm-Based Criteria: Filters or neurons are ranked based on the L1 or L2 norm of their weights. Units possessing

ISSN 2959-6157

the smallest norms are deemed least important and pruned first [9, 11]. Pruning can be applied layer-wise, removing a fixed percentage or number of units per layer, or globally across all layers, removing the globally least important units regardless of layer [12]. An iterative process—prune a small fraction of parameters (e.g., 5-20%), then fine-tune the remaining network to recover accuracy—is commonly employed to mitigate cumulative accuracy loss [9, 10]. Examples: PFEC (Pruning Filters for Efficient ConvNets) utilizes L1-norm for filter pruning [11]. SFP (Soft Filter Pruning) employs L2-norm and dynamically sets pruned filters to zero during training epochs, allowing them the possibility to recover importance if needed, before permanent removal [12].

Explainable AI (XAI)-Based Criteria: Techniques like Layer-wise Relevance Propagation (LRP) offer an alternative perspective by computing a relevance score (R) for each structural unit, quantifying its contribution to the model's final prediction for a given input [9]. The LRP algorithm propagates relevance backwards from the output layer to the input layer under a conservation principle: the total relevance remains constant across layers $\left(\Sigma_i R_i^{(l)} = \Sigma_j R_j^{(l+1)}\right)$ [9]. A commonly used rule for relevance propagation in pruning contexts is LRP-α1β0:

$$R_i^{(l)} = \sum_j \frac{\left(a_i^{(l)} w_{ij}\right)^+}{\sum_{i'} \left(a_{i'}^{(l)} w_{i'j}\right)^+} R_j^{(l+1)} \tag{4}$$

Here, $a_i^{(l)}$ denotes the activation of neuron i in layer l, w_{ij} is the weight connecting neuron i (layer l) to neuron j (layer l+1), and $(\cdot)+$ indicates retaining only positive contributions (mimicking ReLU-like behavior) [9]. For structured pruning, relevance scores (R_k) are typically aggregated per filter or channel (e.g., $R_{filter} = \Sigma\ R_k$ for all neurons k in the filter channel). Units exhibiting the lowest aggregated relevance scores are prioritized for removal. This method provides a pruning criterion intrinsically normalized via the conservation principle and is often more robust to the model›s output confidence compared to simple weight magnitude [9]. Example: Yeom et al. demonstrated the effectiveness of LRP-based pruning, particularly in transfer learning scenarios with limited data availability where traditional magnitude-based pruning may struggle [9].

2.1.3 Quantization

Quantization reduces the numerical precision used to represent model parameters (weights) and activations [4]. Typically, this involves converting from 32-bit floating-point (FP32) to lower bit-width formats like 16-bit floats (FP16), 8-bit integers (INT8), 4-bit integers (INT4), or even binary values (1-bit). This drastic reduction in bit-width directly translates to a significantly smaller memory footprint and enables the use of faster, lower-power integer arithmetic operations on most hardware platforms, including specialized AI accelerators [4].

Post-Training Quantization (PTQ): This technique is applied directly to a pre-trained FP32 model without requiring retraining [4]. Representative calibration data (unlabeled or labeled) is passed through the model to observe the dynamic ranges of activations. Calibration methods (e.g., Min-Max, Entropy-based) then determine optimal scaling factors (quantization parameters) to map the observed float ranges into the target integer range per layer or per tensor. While PTQ is fast and requires no additional training data beyond calibration, it can lead to noticeable accuracy degradation, particularly for lower precision targets (e.g., INT4 or binary) or models with non-linear activation distributions [4].

Quantization-Aware Training (QAT): To mitigate the accuracy loss of PTQ, especially at lower precisions, QAT simulates quantization effects during the training or fine-tuning process itself [4]. During the forward pass, the model weights and activations pass through simulated quantization nodes ("FakeQuant") that mimic rounding and clamping to the target integer precision. Crucially, during the backward pass (backpropagation), the Straight-Through Estimator (STE) approximates the gradient of the quantization function as 1 (∂Round(x)/∂x ≈ 1). This allows gradients to flow through the quantization step as if it were the identity function, enabling the optimization process to update the underlying full-precision weights to become more robust to the quantization noise [4]. Weight updates are performed on the full-precision weights. QAT generally yields models significantly more accurate than PTQ at lower bit-widths but incurs the cost of additional training/fine-tuning time and computational resources. Example: Frameworks like TensorFlow Lite (TFLite) and PyTorch provide built-in QAT APIs (e.g., tf.quantization.quantize_and_dequantize_v2, torch.quantization) enabling efficient deployment of low-precision (e.g., INT8) models on edge hardware [7].

2.1.4 Other compression techniques

Low-Rank Factorization / Tensor Decomposition: These techniques decompose large weight matrices, particularly in fully connected layers or convolutional kernels, into products of smaller matrices (e.g., via Singular Value Decomposition - SVD, Tucker decomposition, Canonical Polyadic decomposition) [4]. This reduces the total number of parameters and the computational cost of the

associated operations (e.g., matrix multiplications, convolutions).

Parameter Sharing: This method forces different parts of the model (e.g., layers, residual blocks) to share identical weights, significantly reducing the total parameter count [4]. While conceptually simple, effective sharing strategies require careful design to minimize negative impacts on model capacity and accuracy.

2.2 Efficient Neural Architecture Design

Instead of compressing large pre-existing models, this paradigm focuses on designing inherently efficient neural network architectures from the outset. These architectures prioritize operations with low computational complexity (FLOPs) and parameter counts while striving to maintain competitive accuracy levels for the target tasks [13].

2.2.1 MobileNet series

The MobileNet series revolutionized efficient Convolutional Neural Network (CNN) design by popularizing the concept of Depthwise Separable Convolution as a fundamental building block [13]. This operation decomposes a standard convolution into two distinct steps:

Depthwise Convolution: Applies a single convolutional filter independently to each input channel, performing spatial filtering. Computational cost: H * W * C_{in} * K * K.

Pointwise Convolution (1x1 Convolution): Applies a standard 1x1 convolution to linearly combine the channels produced by the depthwise step. Computational cost: H * W * C_{in} * C_{out}. Compared to a standard convolution (H * W * C_{in} * C_{out} * K * K), depthwise separable convolution reduces computation and parameters by roughly a factor of $K^2 + 1/C_{out}$. MobileNetV1 established this core block [13]. MobileNetV2 introduced the inverted residual structure with linear bottlenecks: the block expands to a higher-dimensional space via a 1x1 convolution (with ReLU6), applies depthwise convolution, and then projects back to a lower-dimensional space with another 1x1 convolution (using a linear activation to avoid collapsing information). This improves gradient flow and representation capacity. MobileNetV3 further optimized block configurations and channel counts using Neural Architecture Search (NAS) and incorporated computationally efficient nonlinearities like the h-swish (x * ReLU6(x+3)/6) activation function.

2.2.2 ShuffleNet series

ShuffleNets specifically address the computational bottleneck caused by frequent 1x1 convolutions in dense residual-like blocks, especially when combined with group convolutions for further efficiency gains [13]. While group convolutions (splitting input channels into G groups and processing each independently) reduce computation by approximately a factor of G, stacking multiple group convolution layers blocks information flow between groups, hindering representational power.

Channel Shuffle Operation: The key innovation in ShuffleNet enables efficient cross-group information exchange without resorting to dense (non-grouped) convolutions [13]. Given an output feature map with G * N channels (G groups, N channels per group), the operation: 1) Reshapes the channel dimension into (G, N). 2) Transposes this to (N, G). 3) Flattens the result back to a single dimension (N * G channels).

This operation is differentiable and computationally cheap. Crucially, it ensures that each group in the subsequent layer receives input composed of shuffled subgroups originating from all groups in the previous layer, facilitating rich feature mixing [13].

ShuffleNet Unit: This unit builds upon a bottleneck structure similar to ResNet but incorporates group convolution and channel shuffle [13]. It typically replaces the first 1x1 convolution in a bottleneck with a pointwise group convolution, followed immediately by a channel shuffle operation. The spatial convolution (e.g., 3x3) is replaced with a depthwise convolution. A second pointwise group convolution then restores the desired output channel dimension. This combination (Grouped Pointwise Conv -> Channel Shuffle -> Depthwise Conv -> Grouped Pointwise Conv) dramatically reduces FLOPs compared to standard or MobileNet-like blocks while effectively maintaining feature representation and mixing capabilities through the shuffle operation [13]. Example: ShuffleNet units demonstrated significant inference speed advantages on resource-constrained ARM CPUs prevalent in mobile devices [13].

2.2.3 EfficientNet series

EfficientNet leverages Neural Architecture Search (NAS) to systematically scale model dimensions in a balanced manner [13]. The core insight is that the three primary dimensions of a CNN—width (w, number of channels), depth (d, number of layers), and resolution (r, input image size)—are interdependent; optimally balancing them yields significantly better accuracy/computation trade-offs than arbitrarily scaling a single dimension.

Compound Scaling, as epitomized by the EfficientNet paradigm, synthesizes a unified scaling coefficient φ that orchestrates isometric transformation operators applied concurrently to the model's architectural hyperparameter dimensions, thereby instituting a coordinated multi-axis magnification regime [13]: depth: d = α^{ϕ}; width: w = β^{ϕ}

ISSN 2959-6157

; resolution: $r = \gamma^{\phi}$.

The constants α, β, γ are determined by performing a small grid search on the baseline model (EfficientNet-B0) to maximize accuracy under minimal resource constraints. The compound scaling principle dictates that scaling up any single dimension requires corresponding scaling of the others for optimal efficiency [13]. For example, higher resolution input requires a deeper network with more channels to effectively capture the finer-grained features.

Implementation: The baseline model EfficientNet-B0 is first designed using NAS targeting a specific FLOPs budget. Subsequent models (EfficientNet-B1 to B7) are derived by uniformly scaling the baseline dimensions using increasing values of φ, guided by the determined α, β, γ values [13]. This methodical approach consistently produced models that outperformed previous state-of-the-art models in accuracy while being significantly smaller and faster.

2.3 Hardware-Aware Optimization and Adaptation

This pillar focuses on optimizing the deployment and execution of models, considering the specific characteristics and constraints of the target edge hardware platform. The goal is to maximize computational throughput and energy efficiency during inference [6, 7, 14-17].

2.3.1 Hardware-specific kernel optimization

Leveraging Hardware ISA: Optimizing the implementation of fundamental operators (kernels) like convolution, matrix multiplication (GEMM), and activation functions to exploit unique hardware features is crucial [7, 16]. This involves:

Utilizing SIMD (Single Instruction, Multiple Data) instructions on CPUs (e.g., ARM NEON, Intel AVX-512) for parallel data processing.

Exploiting specialized hardware units like Tensor Cores on NVIDIA GPUs for mixed-precision matrix math.

Targeting custom vector/matrix processing units integrated within NPUs/APUs.

Operator Fusion: Combining multiple consecutive operators into a single fused kernel significantly reduces overhead [7, 16]. For example, fusing Convolution -> Batch Normalization -> Activation Function into one kernel eliminates intermediate result writes/reads to/from slow main memory (DRAM), minimizes kernel launch latency, and improves data locality within faster cache memory. This fusion is highly effective in reducing latency and energy consumption, especially for sequences of pointwise operations common in CNNs.

2.3.2 Inference engine compilation and optimization

Dedicated inference engines compile models defined in high-level frameworks (TensorFlow, PyTorch, ONNX) into highly optimized executables tailored for specific target hardware (CPU, GPU, NPU, etc.) [7].

Graph Optimization: Engines perform high-level optimizations on the model's computational graph before code generation [7]:

Constant Folding: Statically computes the output of operations involving constant tensors at compile time.

Dead Node Elimination: Removes operations whose outputs are not used by any other node in the graph.

Common Subexpression Elimination: Identifies and eliminates redundant calculations.

Operator Fusion: Automatically identifies sequences of operators that can be fused into a single kernel (as described in 3.1).

Hardware-Aware Scheduling: The engine schedules the execution of optimized kernels efficiently across available processor cores, manages memory allocation and deallocation strategically to minimize fragmentation and data movement, and handles data transfers between different memory hierarchies efficiently [7].

Quantization Support: Inference engines provide critical support for deploying models quantized via PTQ or QAT, mapping quantized operations to efficient low-precision hardware instructions where available (e.g., INT8 on NPUs, specific CPUs) [7].

Examples: TensorFlow Lite (TFLite) [7] and its delegate mechanism, ONNX Runtime (with execution providers like TensorRT, OpenVINO), Apache TVM (featuring advanced optimizations like auto-tuning kernel implementations for specific hardware targets), NVIDIA TensorRT (optimized for NVIDIA GPUs), Qualcomm SNPE (optimized for Snapdragon platforms), MediaPipe (for building cross-platform ML pipelines).

2.3.3 Heterogeneous computing systems

Modern edge systems often integrate multiple distinct processing units (e.g., CPU + GPU/NPU + FPGA). Efficiently partitioning computational tasks and scheduling them across this heterogeneous landscape is critical for overall system performance and energy efficiency [14, 15, 16].

Task Partitioning and Scheduling: An intelligent runtime system assigns specific computational tasks to the processor best suited for them based on capability and efficiency [14, 16]:

CPU: Handles control-intensive tasks, complex logic, data preprocessing (e.g., image resizing, audio feature extraction), high-level task scheduling, and orchestrating the other accelerators. Its flexibility makes it suitable for non-parallelizable or irregular tasks.

GPU/NPU: Primarily utilized for accelerating the core

computationally intensive DNN inference workloads—matrix multiplications and convolutions—leveraging their massive parallel processing capabilities [14, 16]. NPUs, being Application-Specific Integrated Circuits (ASICs) designed explicitly for AI workloads, typically offer superior performance per watt (TOPS/Watt) compared to general-purpose GPUs for these specific operations [16].

FPGA: Provides programmable hardware ideal for custom acceleration [14, 16]. This includes implementing highly optimized versions of specific DNN layers, deploying custom DNN architectures not well-supported by standard NPUs, or accelerating non-DNN tasks often found in edge pipelines (e.g., signal processing like FFT, image filtering algorithms like bilateral filtering or Otsu thresholding). FPGAs offer a balance of flexibility, performance, and lower power consumption compared to GPUs for fixed-function tasks. Example: Liu et al. proposed a heterogeneous CPU/FPGA/NPU system for unmanned aerial vehicles (UAVs), demonstrating significant efficiency gains by optimally assigning different computational tasks (control, vision processing, sensor fusion) to each processor type [14].

Framework Support: Compilation frameworks like Apache TVM are designed with heterogeneous execution in mind, capable of generating optimized code targeting multiple different backends (e.g., CPU, NPU, GPU) within a single application, managed by an integrated runtime scheduler [7].

2.3.4 Specialized accelerators

Beyond programmable processors (CPUs, FPGAs), dedicated hardware accelerators offer peak efficiency for DNN inference:

Neural Processing Units (NPUs / TPUs): These are ASICs meticulously designed from the ground up for accelerating DNN operations [16, 17]. They feature highly optimized dataflows, large arrays of parallel processing elements (PEs), and specialized memory hierarchies designed to minimize data movement. Examples include the Google Edge TPU, Apple Neural Engine, and Huawei Da Vinci architecture. NPUs consistently achieve significantly higher computational throughput per watt (TOPS/Watt) than general-purpose CPUs or GPUs for standard DNN workloads [16, 17].

Processing-In-Memory (PIM): This emerging paradigm aims to overcome the dominant "memory wall" bottleneck—where energy consumed by moving data between separate memory and processing units often exceeds the energy spent on computation itself [16, 17]. PIM architectures perform computations directly within the memory array where the data resides. This is achieved using novel non-volatile memory technologies like Resistive RAM (ReRAM or RRAM) or Magnetoresistive RAM (MRAM), which can inherently perform simple logic operations. Example: The UNPU accelerator leverages ReRAM-based PIM to achieve exceptional energy efficiency (TOPS/W) for DNN inference [17].

3. Challenges in Lightweight Edge ML

3.1 Algorithm-Level Challenges

3.1.1 Accuracy-efficiency tradeoff dilemma

Model compression techniques reduce computational overhead but often sacrifice accuracy. For example, Binarized Neural Networks (BNNs) exhibit >8% accuracy drop on ImageNet tasks—a phenomenon termed the "Robustness Deficiency Problem" in 2023 neural network compression surveys. Edge devices require real-time adaptation to environmental changes (e.g., lighting, noise), yet lightweight models struggle with online retraining due to limited capacity. Recent studies (MobiSys 2024) show error rates in dynamic scenarios are 3.2× higher than in static environments.

3.1.2 Dynamic environment adaptation gap

Edge devices require real-time adaptation to environmental changes (e.g., lighting, noise), yet lightweight models struggle with online retraining due to limited capacity. Recent studies (MobiSys 2024) show error rates in dynamic scenarios are 3.2× higher than in static environments.

3.2 Hardware-System Challenges

3.2.1 Heterogeneous hardware fragmentation

Diverse edge hardware architectures (CPU/GPU/NPU) hinder universal optimization. ARM's 2024 whitepaper notes: Energy consumption for the same model varies up to 6.8× across chips, severely limiting deployment scalability. Compressed models still face on-chip memory (SRAM) bandwidth constraints. IEEE TCAD 2024 research reveals 60% of edge device energy is consumed by data movement rather than computation—the "Memory Wall" challenge.

3.2.2 Memory-wall bottleneck

Compressed models still face on-chip memory (SRAM) bandwidth constraints. IEEE TCAD 2024 research reveals 60% of edge device energy is consumed by data movement rather than computation—the "Memory Wall" challenge. Privacy solutions like federated learning avoid data uploads but increase edge computation load. USENIX Security 2023 tests show encrypted inference inflates ResNet-18 latency by 400%, contradicting lightweight goals.

ISSN 2959-6157

3.3 Application-Deployment Challenges

3.3.1 Data privacy-security paradox

Privacy solutions like federated learning avoid data uploads but increase edge computation load. USENIX Security 2023 tests show encrypted inference inflates ResNet-18 latency by 400%, contradicting lightweight goals. Deployment across OS (RTOS/Linux) and frameworks (TFLite/ONNX) lacks standardization. ACM EdgeSys 2024 reports deployment costs consume 47% of edge AI project budgets.

3.3.2 Cross-platform deployment barrier

Deployment across OS (RTOS/Linux) and frameworks (TFLite/ONNX) lacks standardization. ACM EdgeSys 2024 reports deployment costs consume 47% of edge AI project budgets.

4. Future Prospects

Neural-Symbolic Hybrid Learning can be considered. Symbolic-reasoning-enhanced lightweight models (e.g., Neuro-Symbolic AI) improve sample efficiency. MIT's AAAI 2024 work demonstrates: Symbol-augmented MobileNet outperforms pure neural networks by 12.7% accuracy using only 10% training data.

Hardware-Aware Neural Architecture Search (NAS): Next-gen NAS will integrate chip-level constraints (e.g., cache size, power budgets). Google's EdgeNAS (ISCA 2024) discovers vision models 3× faster than EfficientNet under 2W power limits.

Federated Learning with Adaptive Compression: Dynamic gradient compression (e.g., sparsification + quantization) addresses communication bottlenecks. ICLR 2024's AdaGQ reduces medical edge device communication costs by 83% without accuracy loss.

In-Memory Computing for Energy Efficiency: In-memory computing architectures (ReRAM/PCM) break the "Memory Wall." Nature Electronics 2024 reports ReRAM arrays achieve 58.3 TOPS/W for ResNet-50 inference—1,000× more efficient than GPUs.

Cross-Stack Co-Design Frameworks: Unified algorithm-hardware-compiler co-design standards are emerging. MLSys 2024's EdgeCoDesign automates end-to-edge deployment from PyTorch to RISC-V chips, cutting manual tuning by 90%.

5. Conclusion

Lightweight machine learning is the core enabler of edge intelligence. This study systematically examines critical challenges: At the algorithmic level, balancing efficiency with accuracy and enhancing dynamic adaptability; at the hardware-system level, hardware fragmentation and memory bottlenecks hinder deployment scalability; at the application level, privacy protection and cross-platform deployment remain fundamentally at odds. Recent surveys identify these as fundamental barriers to scalable edge AI.

Future breakthroughs will rely on interdisciplinary innovation: Neural-symbolic architectures will enhance few-shot generalization; hardware-aware NAS will automate chip-optimal model generation; adaptive federated learning will reconcile privacy-communication tradeoffs; in-memory computing will shatter energy limits; and cross-stack co-design frameworks will unify deployment standards. With the proliferation of 5G-Advanced/6G networks and AI chip miniaturization, lightweight edge ML will progressively realize its vision of "Unlocking Intelligence Under Constraints," empowering trillions of IoT devices to build real-time, secure, and accessible edge intelligence ecosystems.

References

[1] Wang Q, Jin G, Li Q, Wang K, Yang Z, Wang H. Industrial Edge Computing: Vision and Challenges. Information and Control, 2021, 50(3): 257-274.

[2] Hussain H, Tamizharasan P S, Rahul C S. Design possibilities and challenges of DNN models: a review on the perspective of end devices. Artificial Intelligence Review, 2022, 55: 5109–5167.

[3] Hadidi R, Cao J, Xie Y, Asgari B, Krishna T, Kim H. Characterizing the Deployment of Deep Neural Networks on Commercial Edge Devices. IEEE International Symposium on Workload Characterization, 2019: 35–48.

[4] Choudhary T, Mishra V, Goswami A, et al. A comprehensive survey on model compression and acceleration. Artificial Intelligence Review, 2020, 53: 5113–5155.

[5] Reed R. Pruning algorithms—a survey. IEEE Transactions on Neural Networks, 1993, 4(5): 740–747.

[6] Pham-Quoc C, Nguyen X Q, Thinh T N. Towards an FPGA-targeted Hardware/Software Co-design Framework for CNN-based Edge Computing. Mobile Networks and Applications, 2022, 27: 2024–2035.

[7] Li B. Software Framework for Embedded Neural Networks. In: Embedded Artificial Intelligence. Springer, 2024.

[8] Gou J, Yu B, Maybank S J, et al. Knowledge Distillation: A Survey. International Journal of Computer Vision, 2021, 129: 1789–1819.

[9] Yeom SK, Seegerer P, Lapuschkin S, Binder A, Wiedemann S, Müller KR, Samek W. Pruning by explaining: A novel criterion for deep neural network pruning. Pattern Recognition. 2021 Jul 1;115:107899.

[10] He Y, Xiao L. Structured Pruning for Deep Convolutional

Neural Networks: A Survey. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2024, 46(5): 2900–2919.
[11] Anwar S, Hwang K, Sung W. Structured pruning of deep convolutional neural networks. ACM Journal on Emerging Technologies in Computing Systems (JETC). 2017 Feb 9;13(3):1-8.
[12] He Y, Kang G, Dong X, Fu Y, Yang Y. Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks. Proceedings of the 27th International Joint Conference on Artificial Intelligence, 2018: 2234–2240.
[13] Zhang X, Zhou X, Lin M, Sun J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In Proceedings of the IEEE conference on computer vision and pattern recognition 2018 (pp. 6848-6856).
[14] 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 of Things Journal, 2024, 11(16): 27126–27138.
[15] Tan T, Cao G. Deep Learning Video Analytics Through Edge Computing and Neural Processing Units on Mobile Devices. IEEE Transactions on Mobile Computing, 2023, 22(3): 1433–1448.
[16] 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. Proceedings of the IEEE, 2023, 111(1): 42–91.
[17] Yang P, Wang H, Yang J, Qian Z, Zhang Y, Lin X. Deep Learning Approaches for Similarity Computation: A Survey. IEEE Transactions on Knowledge and Data Engineering, 2024, 36(12): 7893–7912.