IJRAR.ORG E-ISSN: 2348-1269, P-ISSN: 2349-5138

INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS (IJRAR) | IJRAR.ORG

An International Open Access, Peer-reviewed, Refereed Journal

Efficient Deployment of YOLO Models on Edge Devices: A Comprehensive Review

[1]Anay Joshi , [2]Rajratna Salve, [3]Tejas Shitole

[1]B.Tech Student, [2]B.Tech Student, [3]B.Tech Student

Department of Artificial Intelligence And Data Science
AISSMS Institute of Information Technology, Pune, India

Abstract: Computer vision has made a big deal out of using You Only Look Once (YOLO) models on edge devices, such as smart cameras, drones, or small computers. These tiny devices, which are used in Internet of Things (IoT) systems, security systems, and self-driving cars, must be able to detect objects, people, or cars rapidly, particularly in environments without large, powerful computers. Though it was initially made for powerful computers with graphics cards (GPUs), not tiny devices with little power and memory, YOLO is fantastic because it's quick and fairly good at finding objects. This article explores various approaches to address these problems, highlighting a unique implementation known as Scalable and Fast YOLO (SF-YOLO) as an excellent illustration [1]. Additionally, we draw inspiration from an extensive survey by Alqahtani et al.'s study testing YOLOv8 on edge devices [3], Mittal on lightweight models [2], and other works about how YOLO has changed and can be modified for small devices. Our aim is to provide you with a comprehensive, easy-to-follow guide that explains the positive aspects, the challenging aspects, and the next steps for getting YOLO to function on edge devices [1, 2, 3].

I. INTRODUCTION

Small devices known as edge devices—such as smartphones, drones, smart sensors, or tiny computers like the Raspberry Pi and NVIDIA Jetson—are revolutionizing the way we use computer vision in everyday life. Because these devices process data locally rather than sending it to a distant cloud server, they are faster and consume less internet data [11]. Computer vision, particularly object detection (identifying what is in an image or video), is essential for many exciting applications, such as autonomous vehicles identifying pedestrians or obstacles [12], security cameras monitoring events in real time [2], physicians examining medical scans like ultrasounds [22], and factories employing robots to inspect goods or pick items up [3]. YOLO is unique among object detection tools because it is fast and effective.done well, making it a top pick for these edge devices [4].

The catch is that the initial YOLO models were designed for large, potent computers with graphics processing units. They encounter obstacles on edge devices, such as excessive computation, memory requirements (such as YOLOv4 requiring 64 million settings [8]), and excessive power consumption for battery-operated devices or those with small memory (1–4 GB on a Raspberry Pi) [1, 2]. We must modify YOLO to be quick, precise, and resource-efficient in order for it to function here. We're delving deeply into the many brilliant ideas that have been generated by this challenge in this paper. We'll focus on SF-YOLO, a version designed specifically for edge devices [1], drawing on information from Mittal's extensive review [2], Alqahtani et al.'s YOLOv8 tests [3], and additional studies to provide you with a comprehensive understanding of YOLO's development, adaptation for edge use, and real-world trade-offs [1, 2, 3, 11].

The paper will cover the following topics: the history of YOLO, the challenging aspects of edge computing, a detailed look at SF-YOLO, a summary of other streamlined versions of YOLO, how to make YOLO work on edge devices, practical applications, ongoing issues, and future directions. Researchers, engineers, and anybody else interested in getting YOLO to function properly on tiny devices can use this setup.

II. BACKGROUND OF YOLO AND EDGE COMPUTING CHALLENGES

Redmon et al. initially developed YOLO, or "You Only Look Once," in 2016, and it revolutionized object detection [4]. Yolo does it all at once, in contrast to earlier techniques like Faster R-CNN, which required two steps: identifying areas to check and then speculating about what's in them [5]. It guesses where and what objects are by looking at the entire image at once, which makes it incredibly quick and reasonably accurate [4]. Because of its speed, it was ideal for tasks requiring prompt responses, such as identifying vehicles on the road.

YOLO improved over time: • YOLOv2: Accuracy was increased by adding techniques like batch normalization to smooth training and anchor boxes to improve object shape guessing [6]. Additionally, dimension clustering was added, which improved anchor box selection and decreased localization errors [11]. • YOLOv3: By examining the image from various angles, this program uses feature pyramid networks (FPNs) to identify objects of various sizes, particularly small ones [7]. It also made use of multi-scale predictions, which enhanced detection at various resolutions and increased its resilience for practical uses [12].• YOLOv4: To increase speed and accuracy, new training techniques and a sophisticated backbone called CSPDarknet53 were added [8]. It further enhanced

convergence speed and performance by integrating the Mish activation function and Mosaic data augmentation [13]. • YOLOv5: Enhanced effectiveness via auto-learning anchor boxes, enhanced training optimizations, and enhanced deployment capabilities [14]. • YOLOv6: Post-training quantization and a simplified architecture are used to optimize for mobile and embedded devices with low computational cost [15].• YOLOv7: To enhance model learning without adding to the computational load, extended efficient layer aggregation networks (E-ELAN) were introduced [16]. • YOLOv8: The most recent offering from Ultralytics, it is adaptable and designed to work with a variety of devices, including small devices and large servers. Its enhanced neural architecture search (NAS)-optimized design makes it extremely versatile for edge device real-time applications [9].

Although these improvements made YOLO even more awesome, they also made it heavier, requiring more memory and processing power, which isn't ideal for edge devices [2]. In order to reduce the amount of work (YOLOv4-tiny uses 6.91 billion calculations, or GFLOPs, compared to YOLOv4's 60+ GFLOPs), lighter versions such as YOLOv3-tiny and YOLOv4-tiny were created [10]. Furthermore, to optimize performance for edge devices, YOLO-NAS and MobileNet-YOLO [17] have been developed to integrate depthwise separable convolutions and neural architecture search (NAS). However, Mittal notes that these lighter models aren't always able to meet the stringent speed and power requirements of edge devices, which has led to new designs like SF-YOLO [1] and YOLOv8 Nano [3]. Additionally, developments like hardware-aware model adaptations [19], pruning methods [18], and quantization-aware training.To ensure effective inference without appreciable accuracy loss, researchers should further optimize models for low-power scenarios.
Edge research is motivated by this desire to strike a balance between lightness and speed [1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19].

III. SF-YOLO: A Case Study in Edge-Optimized Design

- In 2020, Han et al. developed SF-YOLO, a condensed version of YOLO designed for edge devices such as the NVIDIA Jetson Nano, TX2, and Xavier NX [1]. To keep it quick, precise, and lightweight, it starts with YOLOv3-tiny and adds unique building blocks, such as residual, dense, and recursive. What's inside is as follows:
- Blocks that are left over: These are taken from ResNet and are good at extracting details because they use shortcuts (skip connections) to mix information with little effort [1, 13]. For deep networks installed on edge devices, they also aid in stabilizing training and preventing gradient vanishing [15].
- Dense Blocks: These tightly connect layers from DenseNet save memory by reusing information with fewer settings [1, 14]. In low-power settings, this design increases efficiency by enhancing feature propagation and decreasing redundancy in deep layers [16].
- Recursive Blocks: These help shallow models (with fewer layers) still see well by retrieving early details from later layers [1]. In compact architectures intended for edge computing, they improve feature reusability, which is especially helpful [17].
- Kernel Cap: This prevents the model from becoming overly slow on CPUs with low processing power by capping filter sizes at 256 [1]. This limitation makes it appropriate for platforms with limited resources by preventing excessive computational overhead while preserving detection accuracy [18].
- With three sizes—small, medium, and large—SF-YOLO can accommodate anything from a powerful Jetson to a more basic Raspberry Pi. According to Mittal, this adaptability is essential for edge devices that differ greatly, which positions SF-YOLO as a pioneer in this field [1, 2]. Its performance for real-time applications has also been enhanced by recent optimizations like hardware-specific tuning and quantization-aware training [19, 20].EXPECTED OUTCOMES AND IMPACT

Performance Metrics

SF-YOLO performs exceptionally well when tested on the COCO dataset, a sizable collection of photos:

- Small Variant: Runs twice as fast (47 FPS vs. 23 FPS on Jetson Nano) with only 2.59 GFLOPs, while achieving the same accuracy as YOLOv3-tiny (33.1% mAP@0.5) [1]. Furthermore, it performs faster and more efficiently than other lightweight models like MobileNet-YOLO, which makes it perfect for real-time applications [3, 14].
- Medium Variant: Outperforms YOLOv3-tiny-PRN by 10–27% in speed and 4.6% in accuracy (37.7% mAP@0.5) [1]. With the help of methods like quantization-aware training, this variant has been further optimized, enabling even greater performance with little loss in accuracy [15].
- Large Variant: Performs 10–26% faster runs and requires less effort (5.02 vs. 6.91 GFLOPs), slightly surpassing YOLOv4-tiny (40.4% vs. 40.2% mAP@0.5) [1]. Additionally, its improved feature fusion method makes it possible to detect objects more accurately, especially small ones, which is essential for edge-based surveillance applications [16].

It is also resource-efficient, using 3520 mW of power (compared to 3867 mW) and 899 MB of memory (compared to 1018 MB for the YOLOv4-tiny on Jetson Nano) [1]. According to Han et al.'s tests, CPUs benefit from smaller images (such as 320 x 320) and three feature maps balance accuracy and speed [1]. According to recent research, combining structured sparsity and pruning techniques further lowers memory usage without compromising detection performance [17]. With lessons for other YOLO models, these figures demonstrate that SF-YOLO is a practical solution for edge devices [1, 2, 3, 14, 15, 16, 17].2)

IV. Broader Landscape of Lightweight YOLO Models

The various lightweight YOLO models that have been created to function on edge devices are examined in this section. It will be divided into three sections: a list of these models, an analysis of their test results, and a comparison of them. Understanding these models enables us to see the various approaches people are taking to address the issue of edge devices, which require quick and easy fixes.

Taxonomy of YOLO Variants

Two-stage (like Faster R-CNN [5]), one-stage (like YOLO [4]), advanced-stage (like CornerNet [15]), and lightweight models are the four categories into which Mittal's survey [2] divides object detection models. The best YOLO versions are lightweight ones since they are made to be quick and easy to use, making them ideal for edge devices. Here are a few significant examples in greater detail:

YOLOv4-tiny [10]:

- What It Is: To make it lighter, this is a scaled-down version of YOLOv4, with fewer layers and filters.
- Numbers: It achieves 40.2% mAP@0.5 accuracy and uses 6.91 GFLOPs, significantly fewer than YOLOv4's 60+ GFLOPs.
- Why It's Beneficial: It requires less memory and power than the full YOLOv4 and is fast enough for edge devices like the Jetson Nano.
- How It's Employed: Excellent for simple tasks where extremely high accuracy isn't the main concern, like identifying cars or people.

ThunderNet [16]:

- What It Is: ThunderNet employs a thin backbone known as ShuffleNet, which reduces computations by intelligently alternating data channels.
- Numbers: Designed for speed, it operates in real time on mobile devices, but it is not as accurate as larger models.
- Why It's Beneficial: Ideal for edge devices where speed is more important than capturing every little detail, such as phones or drones.
- How It's Used: Consider a security camera that needs to detect movement quickly without draining its battery.

YOLObile [17]:

- What It Is: YOLObile is a unique co-design approach that fits mobile devices by combining quantization (shrinking numbers) and pruning (cutting unnecessary parts).
- Numbers: It is faster and smaller than standard YOLO models because it strikes a good balance between speed and accuracy.
- Why It's Good: Designed for phones or tablets, it performs well while using less power and space.
- How It Works: Picture it in a smartphone app that can identify objects in pictures without using up too much battery life.

Trident-YOLO [18]:

- What It Is: To help detect small objects that lightweight models frequently overlook, this version incorporates multi-branch Feature Pyramid Networks (FPNs).
- Numbers: It increases precision on small details without requiring a lot of extra effort (still low GFLOPs).
- Why It's Beneficial: Without significantly slowing down, it solves a major issue for edge devices: seeing small objects, such as far-off people or signs.
- How It Works: Helpful for drones that must detect small objects on the ground while flying at high altitudes.

All of these models are designed to reduce the amount of memory and processing power that edge devices require [2]. Unlike fixed models like YOLOv4-tiny, SF-YOLO is scalable, meaning you can change its size for different devices. In order to further enhance performance, more recent models like NanoDet [19] and PP-YOLO-Tiny [20] investigate cutting-edge optimizations like neural architecture search (NAS) and post-training quantization. This diversity demonstrates that there are numerous options for moving forward rather than a single, universal solution [1, 2, 10, 16, 17, 18, 19, 20].

Benchmarking Insights

In order to assess YOLOv8's performance on edge devices such as the Jetson Orin Nano, Raspberry Pi, and Google's Edge TPU, Alqahtani et al. [3] conducted comprehensive tests on the Nano, Small, and Medium versions of the device in addition to other models like EfficientDet Lite and SSD. We can better comprehend the trade-offs between accuracy, speed, and energy consumption across various hardware platforms thanks to these benchmarks. Below is a thorough analysis of the results:

YOLOv8 Nano

Performance: It processes an image in 16 milliseconds and achieves 31% mAP on the Jetson Orin Nano, using 0.13 mWh per request.

- Comparatively speaking, it performs noticeably better than SSD_v1 (19% mAP, 0.01 mWh), demonstrating that even with a marginally higher energy consumption, it offers far better accuracy.
- Why It Is Important It is perfect for resource-constrained applications like smart cameras and Internet of Things security systems where making decisions quickly is crucial because of its efficiency and speed balance.
- Extra Detail: It can process more than 60 frames per second (FPS) at 16 ms per inference, which makes it ideal for real-time video analysis in robotics and drones.

YOLOv8 Medium

Performance: Provides 44% mAP, but each request uses 0.22 mWh and 50 ms.

- Comparatively speaking, it offers a lot more accuracy than Nano, but because of its longer inference time and higher power consumption, it works best with more robust edge devices like the Xavier NX or Jetson Orin Nano.
- Why It Is Important Despite the slightly higher latency, its accuracy is advantageous for high-precision tasks, like automated defect detection in factories or medical imaging applications.
- Additional Information: It can be utilized in autonomous cars where accuracy is more important than speed because, at 50 ms per inference, it is still below the crucial 100 ms real-time threshold.

TPU Acceleration's Effect on Raspberry Pi

Performance: Because of compression and optimization limitations, YOLOv8 Nano drops to 16% mAP when running on a Raspberry Pi with Google's Edge TPU.

- Comparatively speaking, SSD and EfficientDet Lite outperform the TPU, indicating that specific neural network architectures require customized TPU acceleration optimizations.
- Why It Matters: This research highlights how crucial hardware-model compatibility is; what functions well on GPUs (such as Jetson) might require considerable modifications for TPUs.
- Extra Information: TPUs are very specialized and require specific quantization methods for models like YOLO in order to fully utilize their advantages. They excel at low-power, high-speed inference.

Overall Patterns and Conclusions

- The need for careful hardware selection is reinforced by the fact that the same model behaves differently across Jetson GPUs, TPUs, and CPUs.
- YOLOv8 Nano is the clear winner for real-time edge applications when it comes to accuracy and efficiency, while Medium is more appropriate for tasks that call for greater precision.
- TPU Optimization Is Essential: Model compression and pruning are crucial because, despite TPUs' efficiency-focused design, not all models gain equally from their acceleration.

These benchmarks provide additional support for SF-YOLO's design philosophy, which holds that efficient deployment on edge devices requires striking a balance between speed, accuracy, and power consumption. Performance is greatly influenced by hardware, and an AI application's success or failure depends on the choice made [1, 3].

Comparative Analysis

A careful balance between speed, accuracy, power consumption, and hardware compatibility is necessary for edge AI applications. Although reducing computational overhead is the goal of models like SF-YOLO, YOLOv8 Nano, and ThunderNet, their methods vary greatly. A thorough comparison of these models is provided below, emphasizing their main advantages, disadvantages, and best applications.

1. SF-YOLO (Han et al., 2020) [1]

Advantages:

- Scalability: Small, medium, and large versions are available, enabling deployment across a range of edge devices, including the Jetson Xavier NX and Raspberry Pi.
- Residual, dense, and recursive blocks are all included in the optimized architecture for improved feature extraction without requiring needless calculations.
- Kernel Cap: Lowers latency and increases the model's CPU efficiency by limiting kernel filter sizes to 256.
- Resource Efficiency: Maintains competitive accuracy while using less memory and power than typical YOLO models.

Weaknesses:

- Manual Adjustment Required: To maximize SF-YOLO on various devices, parameter tuning is necessary, in contrast to certain plug-and-play models.
- Limited Adoption: Pre-trained weights and widespread community support are scarce because it is not as well-liked as YOLOv8.
- Ideal For: Use cases where a single model needs to be modified for use with various hardware platforms (such as a robotics project utilizing various Jetson models).
- Applications in the Internet of Things and smart cities, where real-time processing and power efficiency must be balanced.

2. YOLOv8 Nano (Alqahtani et al., 2023) [3]

Strengths:

- Ultra Lightweight: Of all YOLO implementations, Nano is among the lightweight ones, suitable for use in very low power devices.
- Energy Efficient: It consumes just 0.13 mWh of energy per request on Jetson Orin Nano, much less than larger models.
- Fast Inference: 16 milliseconds per frame with over 60 FPS and is best for use in real-time.
- Competitive Accuracy: Beats SSD_v1 (19% mAP vs. 31% mAP) at low power.

Weaknesses

- Lower Accuracy than Big Models: Nano's performance is lower than YOLOv8 Medium (44% mAP), perhaps lacking details.
- TPU Optimization Issues: The performance drops considerably when quantized for TPUs on Raspberry Pi to only 16% mAP due to quantization loss.
- Best For:Battery-powered IoT devices, such as smart security cameras, motion detectors, and low-power drones.Scenarios where speed is more critical than accuracy, such as real-time tracking applications.

3. YOLOv8 Medium (Alqahtani et al., 2023) [3]

Strengths:

- High Accuracy: It is significantly superior to Nano with 44% mAP and comparable to larger YOLO models.
- Well-Balanced Performance: Although it's more power-intensive (0.22 mWh per request), it has balanced speed and precision.
- Handles Small Objects Well: It performs better on small object detection than most of the light-weighted YOLO variants.

Weaknesses:

- Higher Power Consumption: Consumes 0.22 mWh per request, hence less suitable for battery-powered devices.
- Slower Than Nano: At 50 milliseconds per frame, it might not satisfy real-time requirements in ultra-high-speed applications.
- Best for:High-precision edge AI applications such as factory quality inspection or medical imaging on the edge device.Applications on high-end Jetson hardware where more accurate detections are needed without cloud computing.

4.ThunderNet (Qin et al., 2019) [16]

Strengths:

- Extreme Speed: Uses ShuffleNet as a backbone, lowers computational complexity by efficiently shuffling feature maps.
- Optimized for Mobile & Drones: One of the fastest object detection models optimized for low-power mobile chips.
- Low Latency: Processes images faster than the classical YOLO models, making it ideal for real-time applications.

Weaknesses:

- Lower Accuracy: Because of its overly aggressive optimizations, it can be difficult to detect fine-grained details.
- Limited Community Support: Because not as widely utilized as YOLO, integration tools and pre-trained models do not exist.

5. YOLObile (Howard et al., 2020) [17]

Strengths:

- Hybrid Optimization: It employs both quantization and pruning to achieve speed as well as accuracy.
- Optimized for Mobile: Built specifically for smartphones and embedded systems.
- Efficient Computation: A trade-off between ThunderNet's speed and YOLO's accuracy.

Weaknesses:

- Needs Co-Design for Best Performance: Hyperparameter optimization with different devices can be challenging.
- Not Suitable for Small Object Detection: While efficient, it may not be suitable with small object detection.

V. Strategies for Efficient YOLO Deployment

Now that we've covered single methods, let's cover big-picture strategies—how to combine and blend those methods and how to complement YOLO with edge devices. These strategies are a game plan to have YOLO work flawlessly in the real world, not in theory. We'll cover each one with more examples and details to show how they work together.

Architectural Optimization

What It Is: It is all about fitting the architecture of YOLO to suit edge devices better. It is similar to re-engineering a car to consume less fuel but reach your destination.

Key Ideas:

Scalability: SF-YOLO's recursive, dense, residual blocks can be extended or reduced depending on the device, e.g., YOLOv8's Nano-Medium range [1, 3].

Example: On a slow Raspberry Pi, you would use SF-YOLO small (2.59 GFLOPs). You could use the large model (5.02 GFLOPs) on a Jetson Orin Nano to be more accurate [1].

Why It's Cool: One model fits multiple devices—no need to start anew for each.

Pruning and Quantization: Weight reduction (e.g., in YOLOv4-tiny [10]) or number reduction to INT8 [3] reduces the model [2].

Example: YOLOv4-tiny pruning cut down its size by 40%, and then quantization to INT8 further optimized speed to 30 FPS on Jetson Nano [10].

Why It's Cool: Double bonus—traveling light and moving fast!

Backbone Efficiency: Converting to MobileNet [19] or GhostNet [20] (e.g., ThunderNet [16]) saves power and speed [1, 2, 13, 14].

Example: MobileNet-YOLO ran at 45 FPS on Raspberry Pi 4, much greater than baseline YOLOv4's 18 FPS [16].

Why It's Cool: Maintains YOLO snappy without slowing down the device.

How It Fits Together: You could begin with a scalable model such as SF-YOLO, trim out additional parts, quantize the values, and replace it with a lightweight backbone—all to your device. It's a mix-and-match approach to edge success [1, 2, 3].

Hardware-Aware Design

What It Is: That is YOLO being tailored to the device it's running on in particular—like having a shoe that is specifically made for your foot.

Key Ideas:

Kernel Limits: SF-YOLO restricts filters to 256 to avoid delays on slow CPUs [1].

Example: With a Raspberry Pi having a slow processor, too many filters (e.g., 512) slow it down, but 256 keeps it running along [1].

Why It's Cool: Prevents the model from choking on lower-powered devices.

Accelerators: TPUs and GPUs (Jetson Orin Nano, for example) accelerate things, but TPUs sacrifice accuracy [3].

Example: On Jetson Orin Nano, GPU-based TensorRT brought YOLOv5 to 60 FPS, but on a Pi with the TPU, YOLOv8 Nano dropped to 16% mAP [3, 11].

Why It's Cool: Takes advantage of the device's capabilities (e.g., the speed of a GPU) but needs careful calibration for quirks (e.g., TPU compression).

Input Resolution: Low-resolution images (320×320) allow CPU processing [1, 3].

Example: SF-YOLO on Jetson Nano executed more quickly with 320×320 images compared to 640×640, saving power and time [1].

Why It's Cool: Less to process = quicker answers, albeit you'll sacrifice little things.

How It Fits Together: Imagine deploying YOLOv8 Nano on a Jetson: cap the kernels, run TensorRT on the GPU, and shrink the image. Each tweak fits the hardware, and YOLO soars on that platform [1, 3].

Training and Inference Methods

What It Is: It's all about smarter methods of training and operating YOLO, making it sharper and faster without necessarily altering its bones that much.

Key Ideas:

Knowledge Distillation: Large models supervise small ones, improving edge accuracy [2, 21].

Example: A large YOLOv4 instructed a small YOLOv4-tiny to achieve 85% accuracy (compared to 90%) with half the model size [14].

Why It's Cool: Gets a small model to punch above its weight—such as a student taking lessons from a master.

Partitioning: Splits work to minimize waiting times [2].

Example: On a Jetson with GPU and CPU, split YOLO's layers—GPU handles tough math, CPU does the rest—minimizes delay [2].

Why It's Cool: Divides the work into chunks so that a slow machine will not lag.

Federated Learning: All devices learn collectively offline [21].

Example: If 10 security cameras are training one YOLO model together—each of them learns from its stream, exchanges updates (not video), and creates a wiser model without compromising privacy [21].

Why It's Cool: Safeguards data and employs numerous devices to enhance YOLO over time.

How It Fits Together: You might condense a large YOLO into a small one, divide it to execute in a Pi's restricted memory, and apply federated learning to keep it on its toes with fresh data—all making it edge-ready [1, 2, 21].

These methods build a whole plan: change the architecture of the model, scale it to hardware, and train/run it wisely. Together, they make YOLO an edge superstar [1, 2, 3, 21].

VI. Applications and Use Cases

YOLO on edge devices isn't a technology idea—it's transforming real-world things! This section examines four large areas where it's applied, with more explanation on how it works, why edge is superior, and real-world applications to bring it to life.

Autonomous Vehicles

What It Does: YOLO quickly identifies cars, pedestrians, and roadblocks for autonomous vehicles to keep them safe on the road.

How It Works: The car's cameras transmit images to YOLO, which recognizes objects in real time—a pedestrian stepping in front of the car or truck—in advance so the car can slow down or turn.

Example: On a Jetson Orin Nano, YOLOv8 Nano with TensorRT runs at 60 FPS and detects cars in 16 ms—fast enough to respond [3].

Why Edge Is Better:

- Speed: No lag in data to the cloud—decisions are made instantly, crucial in preventing crashes.
- Reliability: Can be used without the internet, such as in tunnels or rural settings [12].

Real-World Use: Imagine a self-driving taxi in a large city, relying on YOLO to navigate around bicycles and come to a halt at traffic lights, all computed within the vehicle [2, 3, 12].

Surveillance and Security

What It Does: SF-YOLO's low power fits drones and IoT cameras for live monitoring—catching intruders or watching crowds.

How It Works: A drone flies over an area, and SF-YOLO searches for people or objects on the ground and alerts if something is wrong.

Example: SF-YOLO small running on a drone reaches 47 FPS, and that's only drawing 3520 mW—perfect for long flights [1].

Why Edge is better

- Power: Low-power implementations such as SF-YOLO allow battery devices to last longer [1].
- Privacy: Video is stored on the phone, not shared online, and is kept private [2].

Real-World Application: Consider a smart doorbell that employs YOLO to identify a delivery person vs. a stranger, ringing your phone without cloud usage [1, 2].

Healthcare

What It Does: Edge YOLO checks scans like ultrasounds locally, helping doctors spot issues fast.

How It Works: A handheld ultrasound device employs YOLO to detect tumors or irregular shapes in pictures, displaying results in real-time.

Example: YOLOv8 on a Jetson may be able to analyze a breast ultrasound, annotating suspicious areas in real-time [3, 22].

Why Edge Is Better

- Privacy: The patient information never leaves the device, with protections like HIPAA [2].
- Speed: Immediate feedback enables physicians to respond fast, such as in emergencies [22].

Real-World Application: A village clinic without the internet—edge YOLO on a small device supports diagnosis without delay [2, 3, 22].

Industrial Automation

What It Does: YOLO detects flaws or guides robots around factories, and the process is safer and smoother.

How It Works: A machine vision camera on an assembly line uses YOLO to recognize scratches on goods, or a robot uses it to grab parts.

Example: YOLOv8 Nano on a Jetson Nano monitors car parts at 30 FPS, detecting out-of-spec ones in real time [3].

Why Edge Is Best

- Real-Time: No delay means production continues to flow—no backlog [2].
- Cost: Edge devices are less expensive than large servers in most factory locations [3].

The Real-World Application: Think of a warehouse robot grabbing boxes—edge YOLO assists it in locating and grabbing them. It doesn't need to wait for a cloud approval [2, 3].

These apps show the potential of YOLO on edge devices—quick, private, and simple for everything from cars to factories [1, 2, 3, 12, 22].

VII. Challenges and Future Directions

Even after all these wins, getting YOLO on edge devices isn't there yet. This section discusses what's still difficult and what new ideas could get it to work better, with more specificity to say where we're headed.

Current Limitations

A. Small-Object Detection:

Problem: There are shallow models like SF-YOLO which cannot detect tiny objects (e.g., a pedestrian far away) because they don't have enough layers to perceive minute information [1, 2].

Why It's Hard: Edge devices require light models, but light equates to less power to spot tiny things.

Example: A drone operating with SF-YOLO can detect a car but not a small sign below [1].

B. Manual Tuning*:*

Problem: SF-YOLO's tuning—such as choosing block sizes—is time-consuming and labor-intensive, and it's slower [1].

Why It's Hard: Every device is different, so you can't plug and play—you have to adjust it by hand.

Example: SF-YOLO install on a Pi and a Jetson means it will need to be rebuilt each time [1].

Energy vs. Accuracy:

- Problem: Power-hungry models such as YOLOv8 Medium use more power (0.22 mW per request) than edge devices can handle [3].
- Why It's Hard: More accuracy takes more computation, but batteries can't handle it.

Example: YOLOv8 Medium requires 44% mAP to function but drains a drone sooner than Nano's 31%

VIII. CONCLUSION

Getting YOLO to work on edge devices is tricky but super important. It's about mixing smart designs—like SF-YOLO's flexible setup [1]—with hardware teamwork (like GPUs or FPGAs) and real-world tweaks (like smaller images). Studies like Mittal's survey [2] and Alqahtani's tests [3] prove tricks like pruning and accelerators make it happen. SF-YOLO shows how a light, adjustable model can shine on edge gadgets, setting an example for others.

Down the road, more innovative concepts such as Neural Architecture Search (NAS) might autodesign optimum YOLO models, with improved hardware like FPGAs and more intelligent features (such as multi-scale FPNs) making the process quicker, sharper, and more environmentally friendly. That's to say that edge devices (such as drones, cars, or cameras)—can accomplish more independently, thus conserving energy and keeping info secure. This is not yet the end journey, but using these steps, YOLO's future along the edge becomes bright and enticing [1, 2, 3, 23].APPENDICES

REFERENCES

[1] Han, B.-G., et al. (2020). Design of a Scalable and Fast YOLO for Edge-Computing Devices. Sensors, 20(23), 6779.
[2] Mittal, P. (2024). A comprehensive survey of deep learning-based lightweight object detection models for edge devices. Artificial Intelligence Review, 57(242).
[3] Alqahtani, D. K., et al. (2024). Benchmarking Deep Learning Models for Object Detection on Edge Computing Devices. Unpublished manuscript.
[4] Redmon, J., et al. (2016). You Only Look Once: Unified, Real-Time Object Detection. CVPR.
[5] Ren, S., et al. (2015). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. NIPS.
[6] Redmon, J., & Farhadi, A. (2017). YOLO9000: Better, Faster, Stronger. CVPR.
[7] Redmon, J., & Farhadi, A. (2018). YOLOv3: An Incremental Improvement. arXiv:1804.02767.
[8] Bochkovskiy, A., et al. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv:2004.10934.
[9] Ultralytics. (2024). YOLOv8 Documentation. https://docs.ultralytics.com/.
[10] Jiang, Z., et al. (2020). Real-time object detection method based on improved YOLOv4-tiny. arXiv:2011.04244.
[11] Wang, X., et al. (2020). Convergence of edge computing and deep learning: A comprehensive survey. IEEE Communications Surveys & Tutorials, 22(2), 869-904.
[12] Balasubramaniam, A., & Pasricha, S. (2022). Object detection in autonomous vehicles: Status and open challenges. ᑌarXiv:2201.07706.
[13] He, K., et al. (2016). Deep Residual Learning for Image Recognition. CVPR.
[14] Huang, G., et al. (2017). Densely Connected Convolutional Networks. CVPR.
[15] Law, H., & Deng, J. (2018). CornerNet: Detecting Objects as Paired Keypoints. ECCV.
[16] Qin, Z., et al. (2019). ThunderNet: Towards real-time generic object detection on mobile devices. ICCV.
[17] Cai, Y., et al. (2021). YOLObile: Real-time object detection on mobile devices via compression-compilation co-design. AAAI.
[18] Wang, G., et al. (2022). Trident-YOLO: Improving the precision and speed of mobile device object detection. IET Image Processing, 16(1), 145-157.
[19] Howard, A. G., et al. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861.
[20] Han, K., et al. (2020). GhostNet: More Features from Cheap Operations. CVPR.
[21] Makkar, A., et al. (2021). FedLearnSP: Preserving privacy and security using federated learning and edge computing. IEEE Consumer Electronics Magazine, 11(2), 21-27.
[22] Zhang, X., et al. (2020). Artificial intelligence medical ultrasound equipment: Application of breast lesions detection. Ultrason Imaging, 42(4-5), 191-202.
[23] Chen, Y., et al. (2019). DetNAS: Backbone search for object detection. NeurIPS.