YOLO-LitePi: A Lightweight Real-Time Traffic Sign Detection Pipeline Optimized for Raspberry Pi 5

Nguyen Quang Vinh[1], Nguyen Quoc Duy[1]
and Tin T. Tran[2(✉)][0000−0003−4252−6898]

[1] Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh City, Vietnam
{52200195,52200196}@student.tdtu.edu.vn
[2] Artificial Intelligence Laboratory, Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh City, Vietnam
trantrungtin@tdtu.edu.vn

Abstract. Real-time Traffic Sign Recognition (TSR) on Raspberry Pi 5 is constrained by limited compute budgets. We introduce **YOLO-LitePi**, an optimized detector that applies hardware-aware architectural scaling and deploys on the NCNN inference engine to reduce end-to-end latency. Using a two-stage pipeline—class-agnostic detection followed by a lightweight ShuffleNetV2 classifier—we evaluate on TT100K and VN-Signs. Compared with a YOLOv8n baseline under identical deployment settings, YOLO-LitePi achieves a 24.5%–74.4% improvement in throughput while still preserving strong recognition accuracy suitable for real-time TSR. On Raspberry Pi 5, the complete pipeline sustains 13.22–16.83 FPS with stable responsiveness across varied scenes, fully meeting the latency and reliability requirements of edge driver-assistance applications. We further detail engineering choices (ONNX export, post-processing, and CPU threading) that account for most of the latency improvement, highlighting that backend/runtime optimization can dominate architectural tweaks on resource-limited devices. The results suggest a practical recipe for deploying TSR on low-cost edge platforms without specialized accelerators.

Keywords: Traffic Sign Recognition · YOLO · Raspberry Pi 5 · NCNN · Edge Computing · Real-time Object Detection.

1 Introduction

Advanced driver-assistance systems (ADAS) and autonomous driving hinge on robust visual perception for safety and situational awareness. Among perception

✉ Corresponding author: Tin T. Tran, trantrungtin@tdtu.edu.vn

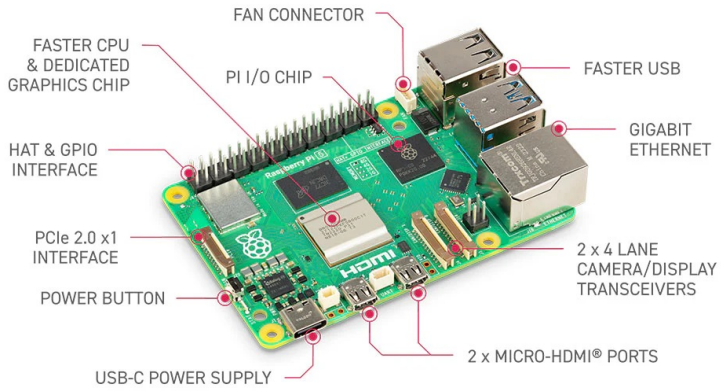

Fig. 1: Raspberry Pi 5 Model B (8 GB RAM). Its upgraded CPU, GPU, and I/O subsystem provide a stronger foundation for deep learning at the edge.

modules, traffic sign recognition (TSR) is fundamental: a typical pipeline comprises *detection* to localize candidate regions and *classification* to assign semantic labels. Although detectors such as YOLO, SSD, and Faster R-CNN attain high accuracy on GPU-equipped systems [1,8], their computational demands remain prohibitive for low-power embedded platforms.

The 2023 release of Raspberry Pi 5 significantly advances affordable edge computing with a faster Cortex-A76 CPU, a VideoCore VII GPU, and higher memory/I/O throughput, enabling on-device neural inference without cloud offloading. Prior work, however, has largely centered on Raspberry Pi 4 or NVIDIA Jetson [8,8], leaving open how accuracy–latency trade-offs evolve specifically on Pi 5 under realistic deployment constraints (runtime, threading, post-processing).

Although lightweight YOLO variants coupled with optimized inference engines (e.g., NCNN) have approached real-time on edge hardware [12], to our knowledge there has not been a *systematic* evaluation of a full two-stage TSR pipeline on Raspberry Pi 5. In particular, the combined impact of hardware-aware architectural scaling, backend/runtime choices, and system-level refinements on end-to-end latency remains underexplored.

To address this gap, we present a comprehensive benchmarking and optimization study of two-stage TSR on Raspberry Pi 5. Our contributions are:

- **YOLO-LitePi and comparative study.** We introduce **YOLO-LitePi**, a compact stage-1 detector for class-agnostic sign localization, and compare it with YOLO baselines (v5n, v8n, v11n) and lightweight classifiers (ResNet, MobileNet, EfficientNet, ShuffleNet) across PyTorch, ONNX Runtime, OpenVINO, and NCNN.
- **Multi-level optimization.** We systematize architectural scaling, backend/runtime acceleration (graph optimizations, NCNN threading), and algorith-

mic/system refinements (vectorized NMS, batched classification, CPU governor tuning, memory pre-allocation).
– **Real-time feasibility on Pi 5.** We break down end-to-end latency (capture, preprocessing, detection, classification, post-processing) and verify real-time throughput on Pi 5.

Under identical deployment settings, YOLO-LitePi yields a 24.5%–74.4% throughput gain over a YOLOv8n baseline with a controlled mAP@0.5 drop of 3.5–6.7 percentage points; the complete pipeline sustains 13.22–16.83 FPS on Raspberry Pi 5, demonstrating practical feasibility for edge driver-assistance scenarios.

2 Background

2.1 Traffic Sign Recognition Approaches

Early traffic sign recognition (TSR) systems relied on handcrafted visual features with classical machine learning. Color-based segmentation, edge detection, and HOG–SVM pipelines were computationally lightweight but highly sensitive to illumination, occlusion, and viewpoint changes, limiting real-world robustness [8]. Deep learning markedly improved TSR: two-stage detectors (e.g., Faster R-CNN [11]) and single-stage models such as SSD [5] and YOLO [10] achieve strong accuracy; the YOLO family (v3–v11) in particular offers a favorable speed–accuracy trade-off and is widely adopted in ADAS and intelligent transportation [1, 12]. In parallel with perception-focused TSR research, other ITS studies have examined temporal traffic behavior—for example, measuring similarity between vehicle speed records using Dynamic Time Warping (DTW) [14]—highlighting the broader landscape of data-driven intelligent transportation systems. However, most deep learning variants are tuned for high-end GPUs, making direct deployment on low-power edge devices challenging without additional simplification and runtime tuning.

2.2 Edge Deployment and Model Optimization

Embedded deployment has been explored on NVIDIA Jetson Nano/Xavier, Google Coral Edge TPU, and Raspberry Pi 4 [8,8]. Common acceleration strategies include pruning, mixed-precision quantization (FP16/INT8), and conversion to efficient inference stacks (ONNX Runtime, TensorRT, NCNN), often yielding 3–5× speedups with modest accuracy loss on supported accelerators [8]. Raspberry Pi 4, however, is limited by its 1.5 GHz Cortex-A72 CPU and modest GPU capabilities. By contrast, the 2023 Raspberry Pi 5 upgrades to a 2.4 GHz Cortex-A76 CPU and a substantially improved VideoCore VII GPU with Vulkan and OpenGL ES 3.1 support, enabling modern inference backends. Yet, to our knowledge, comprehensive benchmarking of full TSR pipelines on Pi 5 that exploits these capabilities for real-time operation remains limited in the literature.

2.3 Benchmarking and System-Level Optimization

While much prior work emphasizes model-level efficiency, system-level effects—image preprocessing overheads, non-maximum suppression (NMS) cost, memory-allocation patterns, and CPU scheduling—can materially impact end-to-end latency on constrained devices [12]. Only a few studies explicitly decompose TSR pipelines into their stages (capture, preprocessing, detection, classification, post-processing) or verify compliance with automotive-style real-time targets (typically 7–15 FPS and total latency < 100 ms) [1]. These observations motivate a holistic optimization framework that couples architectural scaling with back-end/runtime acceleration and hardware-aware system refinements. Building on this perspective, our work quantifies the accuracy–throughput–latency trade-offs of a two-stage TSR pipeline on Raspberry Pi 5 under realistic deployment constraints.

3 Methodology

3.1 System Overview

The proposed Traffic Sign Recognition (TSR) framework adopts a modular two-stage architecture optimized for real-time execution on the Raspberry Pi 5. The system is designed to balance accuracy, latency, and computational efficiency on low-power hardware. A lightweight YOLO-LitePi detector performs the computationally intensive localization step, while a CNN-based classifier handles the recognition of cropped ROIs. This division of tasks yields a superior speed–accuracy trade-off compared to single-stage models.

The complete pipeline comprises six interconnected modules that collectively enable end-to-end traffic-sign detection and recognition:

1. **Image acquisition** — Raw frames are captured using the Pi Camera V1.3 at 30 FPS under varying daylight and motion conditions.
2. **Preprocessing and resizing** — The input frames are converted to RGB, resized to the target resolution (640×640 px), and normalized for consistent illumination.
3. **Traffic sign detection** — A lightweight YOLO-based detector (base on YOLOv8n) [4] localizes candidate regions of interest (ROIs) in real time.
4. **ROI extraction** — Detected bounding boxes are validated based on dimensions, cropped, and batched for efficient downstream processing.
5. **Traffic sign classification** — Each ROI is classified using a backbone (ResNet18 [6], EfficientNet [13], MobileNet [12] or ShuffleNet [6]) to determine the corresponding sign category.
6. **Hierarchical class association** — Classification results are assigned to their spatially corresponding bounding boxes to refine predictions and filter false positives before logging results with timestamps.

Each module is implemented with explicit computational and memory constraints in mind, targeting an overall inference latency below 100ms and throughput exceeding 10 FPS on the Raspberry Pi 5. This modular design facilitates flexible benchmarking, allowing individual components to be profiled, replaced, or re-optimized independently.

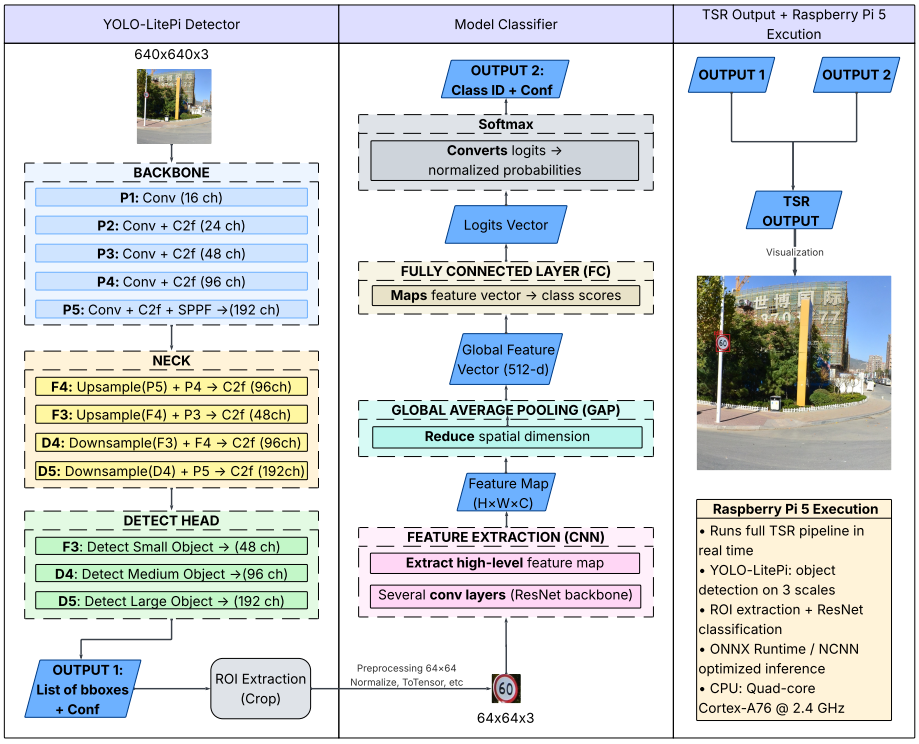

Fig. 2: Two-stage Traffic Sign Recognition pipeline optimized for Raspberry Pi 5. The system integrates lightweight detection and classification modules.

3.2 YOLO-LitePi Detector

The YOLO-LitePi detector is a custom lightweight variant derived from the Ultralytics YOLOv8n architecture [4], as shown in Fig. 2. To further reduce computational complexity for the Raspberry Pi 5, we applied a custom channel pruning strategy. Instead of the standard Nano width scaling (width_multiple=0.25 relative to the baseline), we reduced the channel count by an additional 25% across all backbone and neck layers. .Specifically, the channel dimensions were scaled down to 75% of the standard YOLOv8n configuration. This optimization, combined with a depth multiplier of 0.33, yields a highly compact

network with approximately 1.8 M parameters and 5.2 GFLOPS, representing a 43% reduction in parameter count compared to the standard YOLOv8n (3.2 M params).

Backbone. Given an input image $I \in \mathbb{R}^{640\times640\times3}$, the backbone extracts a pyramid $\{P_i\}_{i=1}^{5}$ via scaled Conv and C2f blocks (depth scaled by `depth_multiple`). The feature-map shapes (channels scaled by 0.25) are:

$$P_1 \in \mathbb{R}^{320\times320\times16}, \quad P_2 \in \mathbb{R}^{160\times160\times24}, \quad P_3 \in \mathbb{R}^{80\times80\times48},$$
$$P_4 \in \mathbb{R}^{40\times40\times96}, \quad P_5 \in \mathbb{R}^{20\times20\times192}.$$

The last stage P_5 is processed by an SPPF module (kernel 5) to enlarge the receptive field at low cost.

Neck. Let $\mathrm{Up}(\cdot)/\mathrm{Down}(\cdot)$ denote $2\times$ up/down-sampling, $[\cdot,\cdot]$ concatenation, and $\mathrm{C2f}(\cdot)$ a C2f block. The PAN-style fusion is:

$$\begin{aligned} F_4 &= \mathrm{C2f}\big([\mathrm{Up}(P_5),\, P_4]\big), \\ F_3 &= \mathrm{C2f}\big([\mathrm{Up}(F_4),\, P_3]\big), \\ D_4 &= \mathrm{C2f}\big([\mathrm{Down}(F_3),\, F_4]\big), \\ D_5 &= \mathrm{C2f}\big([\mathrm{Down}(D_4),\, P_5]\big), \end{aligned} \tag{1}$$

with $F_3 \in \mathbb{R}^{80\times80\times48}$, $D_4 \in \mathbb{R}^{40\times40\times96}$, and $D_5 \in \mathbb{R}^{20\times20\times192}$.

Detection head. The anchor-free head operates on $\{F_3, D_4, D_5\}$:

$$\begin{aligned} \mathrm{Detect}(F_3, D_4, D_5) &\to \{T_3, T_4, T_5\}, \\ T_3 \in \mathbb{R}^{80\times80\times(4+1)}, \quad T_4 \in \mathbb{R}^{40\times40\times(4+1)}, \quad T_5 \in \mathbb{R}^{20\times20\times(4+1)}. \end{aligned} \tag{2}$$

Unlike earlier YOLO variants, the objectness branch is removed; each output vector encodes $\{x, y, w, h, \mathrm{conf}\}$. Bounding boxes are predicted via DFL with reg_max $= 8$. Channel groups $[48, 96, 192]$ match the scaled feature widths supplied to the head.

Overall, YOLO-LitePi achieves substantial savings through channel scaling, reduced C2f depth, and lightweight PAN fusion while preserving multi-scale capacity for small/medium traffic-sign instances.

3.3 Multi-Level Optimization Strategy

Deploying deep learning on the Raspberry Pi 5 requires addressing significant bottlenecks in interpreter overhead and serialized operations. We design a hierarchical optimization strategy grounded in empirical profiling:

Model-Level Optimization Profiling reveals that detection accounts for over 60% of latency. To ensure tractability on the Pi 5, we adopt nano-scale architectures(YOLOv5n, YOLOv8n, and YOLOv11n) [4] which utilize efficient convolutional blocks (C3, C2f). For the second stage, we select lightweight ARM-optimized classifiers such as ResNet18, EfficientNet-B0 [13], MobileNetV2 [12], and ShuffleNetV2 [6] ensuring stable inference costs even in multi-sign scenarios.

Inference-Level Optimization To mitigate PyTorch runtime overhead, models are exported to high-performance backends like ONNX Runtime, OpenVINO and NCNN [2, 3, 9]. These engines employ operator fusion and graph rewriting, increasing throughput from 2.4 FPS (PyTorch FP32) to over 13 FPS. This confirms that backend optimization significantly outweighs incremental architectural differences.

Algorithmic Optimization To address Python-based post-processing bottlenecks [5], we replace standard NMS with vectorized NumPy operations, eliminating loops and reducing NMS latency by 25-40%. Additionally, memory allocation for preprocessing and ROI extraction is pre-calculated to minimize heap fragmentation and execution jitter.

System-Level Optimization To mitigate Python's GIL overhead, we adopt a sequential batch processing strategy where detected ROIs are aggregated and classified in batches of 8 to maximize vectorization. Additionally, the NCNN detector employs 4-thread intra-op parallelism to fully utilize the Raspberry Pi 5's quad-core Cortex-A76 architecture, ensuring optimal real-time throughput.

4 Experiments

We assess the proposed framework and optimizations described in Section 3.3. The study covers dataset preparation, experimental setup, and a comprehensive performance analysis of the pipeline on the Raspberry Pi 5.

4.1 Dataset

Experiments utilize the TT100K 2021 benchmark [15] and our proprietary VN-Signs dataset. While TT100K provides a large-scale baseline with 10,000 images under diverse conditions, VN-Signs [7] introduces localized Vietnamese road scenarios—characterized by distinct materials and complex urban backgrounds. Summary statistics for both datasets are provided in Tables 1.

4.2 Experimental Settings and Metrics

Training Configuration YOLO models were trained on an NVIDIA Tesla P100 (16 GB) using SGD with cosine annealing (lr = 0.01, momentum = 0.937,

Table 1: Dataset statistics comparison between TT100K and VN-Signs after preprocessing.

Metric	TT100K	VN-Signs
Images (Train/Test)	6,034 / 3,022	2,552 / 639
Objects (Train/Test)	16,749 / 8,274	6,689 / 1,645
Avg. boxes per image	2.78	2.6
Avg. box area ratio	0.0007	0.0038
Mean image width (px)	2,048	1,198
Mean image height (px)	2,048	681
Number of classes	91	49

decay = 0.0005). Training ran for 30 epochs with batch size 32 and 640×640 resolution, using early stopping (patience 10). Data augmentation included Mosaic, scaling, Copy–Paste, HSV jitter, and horizontal flips. Classification models were trained on cropped patches using cross-entropy loss with Adam/SGD and validation-based early stopping.

Deployment Environment After training, the detection and classification models are optimized and deployed on a **Raspberry Pi 5** running Debian Trixie (v13). The software stack includes Python 3.13, ONNX Runtime 1.23.2, OpenVINO 2025.3.0, NCNN 1.0.20250916, and OpenCV 4.12.0.88.

Evaluation Metrics

- **Detection metrics:** We adopt standard COCO-style metrics, including **Precision**, **Recall**, **F1-score**, **mAP@0.5**, and **mAP@0.5:0.95**, which jointly evaluate bounding-box correctness and overall detection quality across multiple IoU thresholds.
- **Classification metrics:** Cropped traffic-sign patches are evaluated using **Accuracy**, **Precision**, **Recall**, and **F1-score** (macro-averaged), along with the confusion matrix to analyze inter-class misclassification patterns.
- **Deployment metrics:** To assess real-time performance on embedded hardware, we measure **inference latency** (ms/frame), **throughput** (FPS), and **model footprint** (ONNX and quantized sizes) across all backends on the Raspberry Pi 5.

4.3 Detection Results

We evaluate the detection module in two complementary aspects: (1) training convergence on the TT100K, VN-Signs dataset, and (2) deployment performance on the Raspberry Pi 5, which is the primary target platform.

Training Behavior and Convergence Validation metrics after 30 epochs are reported in Table 2. The standard nano detectors (YOLOv5n, YOLOv8n, and YOLOv11n) achieve closely clustered mAP@0.5 scores (0.912–0.916) on TT100K, indicating that recent architectural revisions provide only marginal gains for this single-class task.

By comparison, the proposed **YOLO-LitePi** reflects a purposeful accuracy–efficiency trade-off. Although it shows a moderate reduction in mAP (∼5% on TT100K and ∼4% on VN-Signs) relative to YOLOv8n, it converges faster and reduces overall training time by 6.98% (TT100K) and 11.52% (VN-Signs). This balance positions YOLO-LitePi as an attractive option for resource-limited edge deployment.

Table 2: Validation performance after 30 epochs on TT100K and VN-Signs datasets.

Model	Precision	Recall	AP@.5	mAP.5-.95	Train
Dataset: TT100K					
YOLOv5n	0.9167	0.8382	0.9121	0.6575	89.40
YOLOv8n	**0.9218**	0.8387	0.9163	**0.6618**	88.97
YOLOv11n	0.9156	**0.8456**	**0.9164**	0.6613	90.64
YOLO-LitePi	0.8988	0.7937	0.8704	0.6034	**83.16**
Perf. Trade-off	-2.50%	-6.15%	-5.02%	-9.68%	+6.98%
Dataset: VN-Signs					
YOLOv5n	**0.9676**	0.9380	0.9803	0.7644	17.70
YOLOv8n	0.9616	**0.9453**	**0.9824**	**0.7673**	16.66
YOLOv11n	0.9527	0.9438	0.9763	0.7618	17.08
YOLO-LitePi	0.9313	0.8881	0.9437	0.6828	**15.66**
Perf. Trade-off	-3.75%	-6.05%	-3.95%	-12.23%	+11.52%

Inference Performance on Raspberry Pi 5 We evaluate the baseline performance of all detectors in a native PyTorch FP32 environment on the Raspberry Pi 5 (8 GB). The comparison includes standard YOLO nano variants, classical architectures (SSD-VGG16, Faster R-CNN), and the proposed YOLO-LitePi. Results are summarized in Table 3.

As observed, classical non-YOLO architectures are ill-suited for this edge deployment scenario. Faster R-CNN, while accurate, is computationally prohibitive (0.18 FPS), and SSD-VGG16 fails to achieve competitive precision on the challenging TT100K dataset. Among the standard YOLO variants (v5n, v8n, v11n), performance plateaus significantly; they exhibit negligible differences in accuracy (<1% variance) and remain clustered around a throughput of 2.3–2.5 FPS.

This indicates that on CPU-only execution, standard architectural updates yield limited latency improvements.

In contrast, **YOLO-LitePi** demonstrates a significant efficiency breakthrough. On the TT100K dataset, it yields a **25.81% increase in FPS** (3.12 vs. 2.48) with a controlled 5% reduction in mAP compared to the best baseline. On the VN-Signs dataset, the throughput doubles **(+100%) to nearly 5 FPS**. These PyTorch FP32 results confirm that while architectural pruning provides initial gains, substantial real-time performance requires the backend optimizations detailed in the subsequent section.

Table 3: Detection results using PyTorch FP32 on Raspberry Pi 5 (8 GB).

Model	Prec.	Recall	F1	AP@.5	mAP	FPS	Lat.
Dataset: TT100K							
SSD-VGG16	0.4807	0.3156	0.3810	0.3642	0.1894	1.15	869.6
Faster R-CNN-R50	0.7478	0.6588	0.7004	0.7332	0.4754	0.18	5555.6
YOLOv5n	0.9167	0.8382	0.8760	0.9121	0.6575	2.51	398.4
YOLOv8n	**0.9218**	0.8387	0.8783	0.9163	**0.6618**	2.33	429.2
YOLOv11n	0.9156	**0.8456**	**0.8792**	**0.9164**	0.6613	2.48	403.2
YOLO-LitePi	0.8989	0.7937	0.8430	0.8704	0.6034	**3.12**	320.5
Perf. Trade-off	-2.49%	-6.14%	-4.11%	-5.02%	-9.68%	+25.81%	–
Dataset: VN-Signs							
SSD-VGG16	0.8165	0.8085	0.8125	0.7871	0.5162	1.16	862.1
Faster R-CNN-R50	0.7647	0.6857	0.7230	0.7648	0.4778	0.18	5555.6
YOLOv5n	**0.9676**	0.9380	0.9526	0.9803	0.7644	2.49	401.6
YOLOv8n	0.9616	**0.9453**	**0.9534**	**0.9824**	**0.7673**	2.34	427.4
YOLOv11n	0.9527	0.9438	0.9483	0.9763	0.7618	2.48	403.2
YOLO-LitePi	0.9313	0.8881	0.9092	0.9438	0.6828	**4.98**	200.8
Perf. Trade-off	-3.75%	-6.05%	-4.64%	-3.94%	-12.23%	+100.0%	–

Prec.=Precision; *mAP*=mean AP@[0.5:0.95]; *Lat.*=Latency in *ms*.

Inference Backend Comparison To assess the effect of deployment frameworks, we benchmarked detectors on Raspberry Pi 5 (8 GB) using ONNX Runtime, OpenVINO and NCNN with identical input resolutions and preprocessing settings.

A clear efficiency hierarchy emerges. Moving from the native PyTorch baseline to ONNX Runtime immediately improved throughput to ∼6.5 FPS with no accuracy loss. The most substantial gains came from OpenVINO and NCNN, both reaching ∼13 FPS for standard models—nearly a 5x speedup over PyTorch—while preserving mAP@0.5 scores above 0.90.

Table 4: Backend comparison on Raspberry Pi 5 (8 GB), dataset TT100K.

Model	Backend	Precision	Recall	F1	AP@.5	FPS
YOLOv5n	ONNX	0.9162	0.8393	0.8761	0.9125	7.23
YOLOv8n	ONNX	**0.9212**	0.8366	0.8769	0.9148	6.74
YOLOv11n	ONNX	0.9153	0.8464	0.8795	**0.9168**	6.70
YOLO-LitePi	ONNX	0.9027	0.7904	0.8428	0.8700	**9.16**
Performance Trade-off		−2.01%	−6.61%	−4.17%	−5.11%	26.71%
YOLOv5n	OpenVINO	0.9121	0.8421	0.8757	0.9015	13.56
YOLOv8n	OpenVINO	0.9207	0.8389	0.8779	0.9078	13.26
YOLOv11n	OpenVINO	0.9159	**0.8554**	**0.8846**	0.9100	13.20
YOLO-LitePi	OpenVINO	0.9014	0.7968	0.8459	0.8751	**15.51**
Performance Trade-off		−2.10%	−6.85%	−4.38%	−3.83%	14.35%
YOLOv5n	NCNN	0.9087	0.8395	0.8727	0.9015	13.32
YOLOv8n	NCNN	0.9195	0.8374	0.8765	0.9072	13.40
YOLOv11n	NCNN	0.9122	0.8467	0.8782	0.9061	13.30
YOLO-LitePi	NCNN	0.8902	0.7993	0.8423	0.8750	**16.69**
Performance Trade-off		−3.18%	−5.60%	−4.09%	−3.55%	24.55%

The proposed **YOLO-LitePi** benefited the most from these optimizations. Using NCNN, it reached **16.69 FPS** on TT100K and **24.04 FPS** on VN-Signs, outperforming the strongest ONNX baseline by up to 94.56% with a controlled 6–7% accuracy trade-off. These results confirm that backend-level optimization is the primary driver of real-time performance on edge hardware, outweighing incremental architectural differences. While OpenVINO provides stable and predictable inference, NCNN offers the latency advantage necessary for ultra-fast embedded deployment.

4.4 Classification Results

This section evaluates lightweight CNN classifiers on the Raspberry Pi 5 CPU (PyTorch FP32) to establish the baseline computational cost of the recognition stage.

Performance Analysis and Model Selection Table 6 details the classification performance. All architectures achieve high top-1 accuracy (> 98%), confirming the quality of the generated sign patches.

Based on the empirical results, we select **ShuffleNetV2** as the primary classifier for the end-to-end pipeline. This decision is driven by three key factors aligning with our edge-computing objectives:

1. **Superior Throughput**: ShuffleNetV2 demonstrates dominant inference speed, achieving **405.4 FPS** on the TT100K dataset – approximately **2x faster**

Table 5: Backend comparison on Raspberry Pi 5 (8 GB), dataset VN-Signs.

Model	Backend	Precision	Recall	F1	AP	FPS
YOLOv5n	ONNX	**0.9721**	0.9362	**0.9538**	**0.9802**	7.35
YOLOv8n	ONNX	0.9592	0.9426	0.9508	0.9795	6.85
YOLOv11n	ONNX	0.9579	0.9401	0.9489	0.9773	6.84
YOLO-LitePi	ONNX	0.9354	0.8292	0.8791	0.9126	**14.30**
Perf. Trade-off		−3.77%	−12.01%	−7.83%	−6.90%	94.56%
YOLOv5n	OpenVINO	0.9563	0.9439	0.9501	0.9788	14.26
YOLOv8n	OpenVINO	0.9714	0.9278	0.9491	0.9789	13.81
YOLOv11n	OpenVINO	0.9493	0.9438	0.9465	0.9737	13.71
YOLO-LitePi	OpenVINO	0.9318	0.8309	0.8785	0.9131	**22.74**
Perf. Trade-off		−4.08%	−11.96%	−7.54%	−6.71%	59.45%
YOLOv5n	NCNN	0.9586	0.9424	0.9504	0.9786	13.78
YOLOv8n	NCNN	0.9684	0.9326	0.9502	0.9788	13.61
YOLOv11n	NCNN	0.9475	**0.9441**	0.9458	0.9737	13.72
YOLO-LitePi	NCNN	0.9297	0.8298	0.8769	0.9133	**24.04**
Perf. Trade-off		−4.00%	−12.10%	−7.74%	−6.69%	74.43%

than ResNet18 (203.1 FPS) and MobileNetV2 (174.3 FPS). This ultra-low latency ensures that the classification stage consumes negligible CPU time (< 2.5 ms per ROI), leaving maximum system resources available for the detection stage and other background tasks.

2. **Best Performance on Target Domain**: On the proprietary VN-Signs dataset, ShuffleNetV2 achieves the highest accuracy among all candidates (**0.9951**), outperforming even the heavier ResNet18 (0.9927). This indicates that the architecture's channel shuffle operation effectively captures the specific features of Vietnamese traffic signs.
3. **Efficiency-Accuracy Balance**: While ShuffleNetV2 incurs a marginal accuracy drop on TT100K compared to ResNet18 (0.9823 vs 0.9870, a difference of $< 0.5\%$), the massive gain in throughput justifies this trade-off for a real-time embedded application.

Table 6: Classification results on Raspberry Pi 5 CPU for TT100K and VN-Signs

Model	Accuracy	Precision	Recall	F1-score	FPS
Dataset: TT100K					
ResNet18	0.9870	**0.9817**	0.9637	0.9712	203.1
MobileNetV2	0.9883	0.9811	0.9680	**0.9735**	174.3

Continued on next page

Model	Accuracy	Precision	Recall	F1-score	FPS
EfficientNet-B0	**0.9889**	0.9762	**0.9721**	0.9724	151.6
ShuffleNetV2	0.9823	0.9670	0.9478	0.9532	**405.4**
Dataset: VN-Signs					
ResNet18	0.9927	0.9918	0.9839	0.9873	196.2
MobileNetV2	0.9933	0.9961	0.9864	0.9908	131.4
EfficientNet-B0	0.9939	0.9929	0.9892	0.9906	143.5
ShuffleNetV2	**0.9951**	**0.9966**	**0.9904**	**0.9932**	**279.2**

4.5 End-to-End Pipeline Performance

This subsection reports the end-to-end performance of the integrated two-stage TSR pipeline, combining the **YOLO-LitePi** detector with a ShuffleNetV2 classifier. The evaluation covers the complete processing sequence on the Raspberry Pi 5, including frame acquisition, preprocessing, detection, classification, and final decision generation.

To ensure stable execution and efficient CPU utilization, the system employs NCNN's multi-threaded backend (4 threads) for the detection stage and batched inference for classifying multiple Regions of Interest (ROIs) within each frame. This configuration provides deterministic latency while maintaining a streamlined sequential pipeline.

Across both benchmark datasets, the system delivers real-time throughput on the Raspberry Pi 5. The TT100K evaluation achieves **13.228 FPS** with a corresponding accuracy of 0.861 precision, 0.808 recall, and 0.825 F1-score. On the VN-Signs dataset, the pipeline reaches up to **16.832 FPS**, maintaining precision, recall, F1-score, and mAP@0.5 of 0.877, 0.745, 0.789, and 0.800, respectively. In all settings, the total per-frame latency remains **below 60 ms**, confirming the suitability of the proposed architecture for real-time embedded TSR deployment.

4.6 Discussion

The experiments confirm that the proposed multi-level optimization strategy effectively enables real-time TSR on the Raspberry Pi 5. The results show that backend-level optimization provides the largest performance gains. Replacing the native PyTorch runtime with ONNX Runtime, OpenVINO, or NCNN increased the throughput of standard YOLO models by up to 5x, indicating that compiler-level and backend-specific optimizations dominate over architectural differences among YOLOv5n, YOLOv8n, and YOLOv11n.

The YOLO-LitePi detector achieves its intended trade-off, outperforming all baseline detectors in raw throughput. Interestingly, we observe a notable variance in throughput between datasets: 16.69 FPS on TT100K versus 24.04 FPS on VN-Signs when using NCNN.Analysis of this discrepancy reveals that preprocessing overhead, rather than model inference complexity, is the primary

factor. Since the detector operates on a fixed input size (640×640), the convolutional computational cost remains constant regardless of object size or density. The throughput gap is attributable to the difference in native image resolutions. The TT100K dataset comprises high-resolution images (mean 2048×2048 px), which imposes a significantly higher CPU burden during the resizing and normalization steps compared to the VN-Signs dataset (mean 1198×681 px). This finding reinforces the importance of system-level optimization: on constrained edge devices like the Raspberry Pi 5, data preparation pipelines can become a latency bottleneck comparable to neural inference itself.

The end-to-end system satisfies real-time requirements, achieving more than 10 FPS with total latency below 80 ms when using NCNN (4 threads) and batched classification. Across all evaluations, the detection stage remains the primary latency bottleneck. These findings demonstrate that real-time TSR on low-cost edge hardware is feasible when optimization is applied jointly at the model, backend, and system levels.

5 Conclusion

This work presented a comprehensive benchmarking and optimization study of real-time Traffic Sign Recognition (TSR) on the Raspberry Pi 5. Through systematic evaluation of detection, classification, and end-to-end performance, we showed that real-time TSR on low-cost embedded hardware is achievable when optimization is applied holistically across the model, backend, and system levels.

The proposed **YOLO-LitePi** achieved its intended efficiency–accuracy balance: despite a modest $\sim$5% mAP drop relative to YOLOv8n on TT100K, it consistently delivered the fastest inference across all detectors. When accelerated with NCNN, YOLO-LitePi reached 16.69 FPS on TT100K and 24.04 FPS on VN-Signs, outperforming all baselines in raw throughput while maintaining acceptable accuracy. Backend selection proved to be the dominant factor in latency reduction, with ONNX Runtime, OpenVINO, and NCNN providing up to a 5x speedup over native PyTorch.

End-to-end evaluation further confirmed that the complete TSR pipeline—YOLO-LitePi paired with a lightweight CNN classifier—met real-time constraints, achieving over 10 FPS and sub-100 ms latency on Raspberry Pi 5. Detection remained the primary bottleneck, whereas classification contributed negligible delay due to its high throughput.

Overall, the results establish Raspberry Pi 5 as a practical platform for real-time TSR in resource-constrained deployments. Future work may explore multi-camera configurations, temporal smoothing, deeper quantization strategies, and hardware-accelerated inference paths to further enhance performance.

References

1. Álvaro Arcos-García, Álvarez García, J.A., Soria-Morillo, L.M.: Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and

stochastic optimisation methods. Neural Networks **99**, 158–165 (2018). https://doi.org/https://doi.org/10.1016/j.neunet.2018.01.005
2. Bai, J., et al.: Onnx runtime: A performance-oriented inference engine for deep learning models. Microsoft Research Technical Report (2019), https://onnxruntime.ai/docs/
3. Intel OpenVINO Toolkit. Intel Corporation (2020), https://docs.openvino.ai/
4. Jocher, G., Chaurasia, A., Qiu, J.: Yolo by ultralytics (Jan 2023), https://github.com/ultralytics/ultralytics
5. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y., Berg, A.C.: Ssd: Single shot multibox detector. In: European Conference on Computer Vision (ECCV). pp. 21–37 (2016). https://doi.org/10.1007/978-3-319-46448-0_2
6. Ma, N., Zhang, X., Zheng, H.T., Sun, J.: Shufflenet v2: Practical guidelines for efficient cnn architecture design. In: Ferrari, V., Hebert, M., Sminchisescu, C., Weiss, Y. (eds.) Computer Vision – ECCV 2018. pp. 122–138. Springer International Publishing, Cham (2018)
7. MaiTam, K.N., et al.: Vietnamese Traffic Signs. https://www.kaggle.com/datasets/maitam/vietnamese-traffic-signs/data (2024)
8. Maletzky, A., Hofer, N., Thumfart, S., Bruckmüller, K., Kasper, J.: Traffic sign detection and classification on the austrian highway traffic sign data set. Data **8**(1), 16 (2023). https://doi.org/10.3390/data8010016
9. Ni, H., ncnn contributors, T.: ncnn (Jun 2019), https://github.com/Tencent/ncnn
10. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 779–788 (2016). https://doi.org/10.1109/CVPR.2016.91
11. Ren, S., He, K., Girshick, R., Sun, J.: Faster r-cnn: Towards real-time object detection with region proposal networks. IEEE Transactions on Pattern Analysis and Machine Intelligence **39**(6), 1137–1149 (2017). https://doi.org/10.1109/TPAMI.2016.2577031
12. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: MobileNetV2: Inverted Residuals and Linear Bottlenecks . In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 4510–4520. IEEE Computer Society, Los Alamitos, CA, USA (Jun 2018). https://doi.org/10.1109/CVPR.2018.00474
13. Tan, M., Le, Q.V.: Efficientnet: Rethinking model scaling for convolutional neural networks. In: International Conference on Machine Learning (ICML). pp. 6105–6114 (2019). https://doi.org/10.48550/arXiv.1905.11946
14. Tin, T.T., Hien, N.T., Vinh, V.T.: Measuring similarity between vehicle speed records using dynamic time warping. In: 2015 Seventh International Conference on Knowledge and Systems Engineering (KSE). pp. 168–173 (2015). https://doi.org/10.1109/KSE.2015.69
15. Zhu, Z., Liang, D., Zhang, S., Huang, X., Li, B., Hu, S.: Traffic-sign detection and classification in the wild. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016). https://doi.org/10.1109/CVPR.2016.232