ACMA: An Adaptive Conditional Model Activation Framework for Efficient Real-Time Fire Detection on Edge Devices

Aymane El Mandili
College of Overseas Education, Nanjing University of Posts and Telecommunications, Nanjing, China

He Xu
School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China

ABSTRACT
Real-time fire detection on edge devices presents significant computational challenges. Existing solutions struggle to balance detection accuracy with efficiency in resource-constrained environments. This paper introduces Adaptive Conditional Model Activation (ACMA), a novel conditional execution framework that optimizes deep learning deployment through dynamic model gating. The proposed approach employs multi-color space analysis and scene-aware adaptive thresholding to selectively activate YOLO model only when preliminary fire indicators exceed dynamically calculated thresholds. Experimental results demonstrate that ACMA achieves 77% filtering accuracy with only a 3.2% system-level accuracy reduction compared to continuous YOLO. While CPU usage reduction appears modest (5%) on severely constrained hardware like Raspberry Pi 4B where baseline utilization is already saturated it enables a transformative throughput to increase from 0.14 to 38 FPS, a 270× improvement. On a desktop i5 CPU, ACMA reduces usage by 80% and increases FPS by 25 times.

Keywords
Fire detection, edge device, model optimization, raspberry pi, real-time detection.

1. INTRODUCTION
The escalating threat of wildfires globally, exacerbated by climate change, has intensified the need for rapid, dependable, and deployable early warning systems. Deep learning models, particularly object detectors like YOLO, have set new benchmarks for accuracy in visual fire detection. However, their prohibitive computational cost and energy consumption render them impractical for continuous real-time inference on resource-constrained edge devices, which are often the first and only line of defense in remote areas.

Existing approaches to this problem primarily focus on minimizing the deep learning model through techniques like model pruning, quantization, or the use of lightweight backbone networks. While these methods reduce the baseline cost of a single inference, they do not address the fundamental inefficiency of running a complex model on every single frame, the vast majority of which contain no fire. This represents a significant waste of computational resources. Other hybrid methods fuse traditional computer vision with deep learning but typically employ static, pre-defined thresholds that remain constant regardless of environmental changes. In contrast, the proposed adaptive thresholding mechanism introduces three key innovations:

(1) real-time scene context analysis that continuously monitors lighting conditions (dark/normal/bright) and adjusts sensitivity accordingly.

(2) temporal consistency modeling that uses recent detection history to modulate threshold sensitivity based on fire event persistence.

(3) saturation-aware adjustment that accounts for color intensity variations across different environments. This multi-faceted adaptation enables robust performance across diverse operational conditions where static thresholds inevitably fail.

(4) A Dual-Optimization Pipeline: A novel framework featuring both fast rejection for and static frame-skipping for temporal efficiency, dramatically reducing computational load on edge devices.

(5) An Adaptive Gating Mechanism: A dynamic thresholding algorithm that modulates the sensitivity of the fast rejection stage based on real-time scene analysis (e.g., brightness, saturation), ensuring robustness across varying environmental conditions.

(6) Configurable Temporal Persistence: A user-definable static frame-skipping parameter that allows balancing between computational savings and detection latency based on specific application needs.

This paper proposes a paradigm shift from "always-on" deep learning inference to conditional model execution. The proposed approach centers on fast rejection: making computationally inexpensive yet confident decisions that a frame contains no fire, thereby avoiding the cost of a full model inference. This work posits that for a sizable portion of input data; simple heuristics can provide certainty of the absence of fire. It is only in ambiguous cases where the pre-screening is uncertain that the powerful, but expensive, deep learning model is activated to make the final determination.

Furthermore, the temporal nature of fire events is leveraged to enhance computational efficiency. Unlike transient objects that may appear and disappear instantaneously, fire exhibits persistence in video sequences. Once a frame is confidently identified as containing fire through deep learning verification, The system employs a static frame skipping: the system bypasses deep learning analysis for a fixed number of subsequent frames (N), where N is a user-configurable parameter balancing detection latency and computational savings. This approach acknowledges that fire does not vanish abruptly between consecutive frames, allowing for predictable computational reduction without compromising safety.

To this end, the Adaptive Conditional Model Activation (ACMA) framework is proposed. ACMA leverages robust multi-color space analysis to perform an initial, efficient scan of each frame. An adaptive thresholding mechanism, responsive to scene context like lighting and temporal history, makes the critical gating decision. The system operates on two

optimization levels: 1) Spatial optimization through fast rejection of clear negative frames, and 2) Temporal optimization through static frame-skipping after positive detections. This dual approach ensures computational resources are spent judiciously, only when and where they are most needed.

The key contributions of this work are:

A Comprehensive Evaluation: An extensive experimental study on public fire detection benchmarks demonstrating that ACMA significantly reduces computational load (measured in FLOPs and inference time) while maintaining competitive detection accuracy compared to a baseline continuous YOLO.

Practical Deployment Analysis: Real-world validation on edge hardware, reporting metrics on FPS, CPU/Memory usage, and energy efficiency, providing a clear pathway for practical implementation.

2. RELATED WORK

The development of vision-based fire detection systems has evolved through several distinct phases, from traditional computer vision approaches to modern deep learning solutions and their optimization for edge deployment. This section reviews the relevant literature across these domains to contextualize the proposed Adaptive Conditional Model Activation (ACMA) framework.

2.1 Traditional Computer Vision Methods

Early fire detection research primarily focused on hand-crafted features derived from color spaces and temporal patterns. Celik [1] pioneered an efficient color-based fire detection method in YCbCr color space, establishing foundational work in pixel-level fire classification. This was extended by Gong et al. [2] who integrated multiple visual features including texture and motion characteristics for improved fire recognition. Later research by Ostroukh et al. [3] focused on early fire diagnostics through specialized color detection algorithms.

Although computationally efficient, these traditional methods face significant challenges with false positives from sunset scenes, artificial lighting, and other flame-colored objects. Mazur [4] systematically analyzed methods for reducing false alarms in video-based fire detection systems, highlighting the fundamental limitations of static thresholding approaches that lack adaptability to changing environmental conditions.

2.2 Deep Learning-Based Fire Detection

The advent of deep learning dramatically improved fire detection accuracy through automated feature learning. Initial approaches employed convolutional neural networks (CNNs), with Poojary et al. [5] conducting comparative studies of optimization techniques for fine-tuned CNN models. The field rapidly progressed to object detection architectures, particularly YOLO variants. Liang et al. [6] improved YOLOv5s with enhanced feature extraction modules, while Hoang et al. [7] applied Bayesian optimization for hyperparameter tuning in YOLO-based fire detection.

Recent advancements focus on architectural specialization for fire scenarios. He et al. [8] proposed DCGC-YOLO with dual-channel bottleneck structures, and Wang et al. [9] developed YOLO-LFD specifically optimized for forest fire detection with lightweight characteristics. These approaches achieve remarkable accuracy but maintain substantial computational requirements that challenge edge deployment.

2.3 Edge Computing Optimizations

Addressing computational constraints has driven research toward edge-optimized solutions. Mahmoudi et al. [10] demonstrated compressed deep learning models for real-time forest fire detection on edge devices, achieving significant speedup through model compression techniques. Similarly, Li et al. [11] integrated improved YOLOX with edge computing for tunnel fire smoke monitoring, employing knowledge distillation and quantization for embedded deployment.

Specialized lightweight architecture has emerged for this domain. Xiao et al. [12] developed EMG-YOLO specifically for embedded devices, incorporating efficient module design for multi-scale flame detection. Vazquez et al. [13] explored transfer learning enhanced models for wildfire detection through edge computing, though noting limitations in addressing key edge metrics like inference time and energy consumption.

2.4 Hybrid and Conditional Execution Approaches

Hybrid methods represent a promising direction combining computer vision efficiency with deep learning accuracy. Ryu and Kwak [14] implemented color-based pre-processing to detect candidate regions before deep learning analysis, demonstrating the potential of cascaded approaches but utilizing static thresholds. A significant advancement comes from Vincent et al. [15] who introduced early-exit strategies in compact models, enabling sample-wise dynamic inference and reducing latency by over 50% on edge devices.

Beyond single-device optimization, Gong et al. [16] proposed explainable semantic federated learning for industrial edge networks, addressing system-level challenges in distributed fire surveillance environments. These approaches collectively highlight the trend toward intelligent computation allocation rather than uniform processing.

2.5 Research Gap and Method Positioning

The proposed ACMA framework belongs to the category of conditional execution systems, distinct from detection models. Unlike YOLO-LFD [9] or EMG-YOLO [12] which are detection architectures, ACMA operates as an adaptive gating mechanism that can be coupled with any existing detector. Compared to hybrid methods like [14] which use static thresholds, ACMA introduces dynamic, scene-aware thresholding that responds to real-time lighting, saturation, and temporal context. Furthermore, ACMA incorporates temporal optimization through configurable frame skipping after positive detections, a feature absent in both pure detection models and static hybrid approaches. This positions ACMA as an efficiency-enhancing framework rather than a detection model, focusing on computational allocation optimization while maintaining compatibility with state-of-the-art detectors for final verification.

A fundamental distinction between conditional execution frameworks is the level at which optimization occurs. Table 1 classifies the core mechanisms of various fire detection methods, highlighting their adaptive capabilities. Prior approaches, such as static pre-filters [14] and even early-exit strategies within compact models [15], operate by modifying or accelerating the detection process itself. In contrast, the proposed ACMA framework introduces a system-level gating paradigm. Instead of making a single model more efficient, ACMA employs an adaptive pre-filter to make intelligent,

high-level decisions about when to activate a powerful but costly detector like YOLO. This shift from optimizing computation to managing activation is the key to its transformative efficiency gains on edge hardware.

Table 1: Comparing core mechanisms of different methods with ACMA

Method	Core Mechanism and Target	Adaptive
Baseline Yolo	"Always-on" inference. Runs full detector on every frame.	No
Static Pre-filter [15]	Fixed, color-based pre-processing before a detector.	No
Early-Exit [16]	Model-internal optimization. Adds early-exit branches to a compact CNN to skip later layers.	Partial
ACMA + YOLO (i5)	System-level gating. Lightweight pre-filter activates YOLO only when needed 14%	Yes
ACMA + YOLO (4Pi 4B)	Same as above, tested on severely constrained edge hardware.	Yes

3. METHODOLOGY

3.1 System Architecture

As shown in figure 1, the proposed model consists of four main components: multi-color space pre-filtering, adaptive threshold calculation, conditional activation gate and YOLO-based verification. The figure shows the main idea behind the solution, which is not based on fire detection, it's based on non-fire detection for fast rejection.

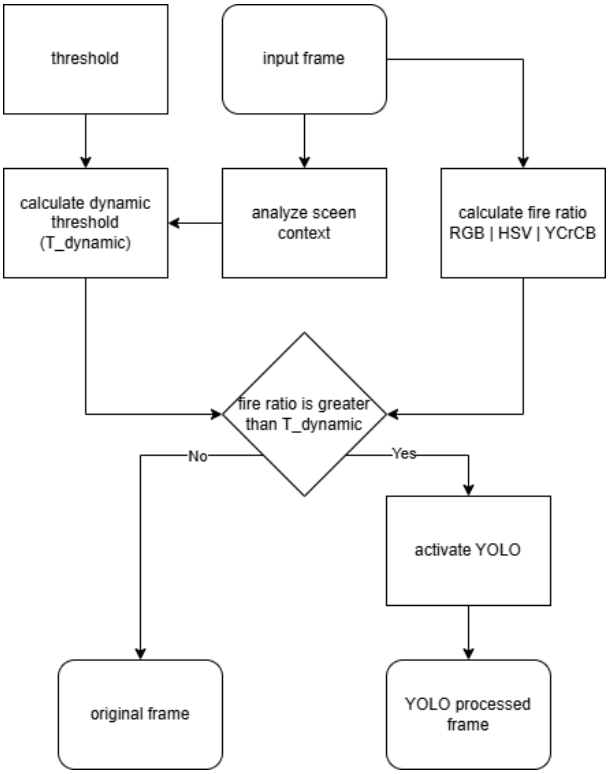

Figure 1: ACMA Architecture

3.2 Multi-Color Space Analysis

3.2.1 RGB Color Space Mask

The RGB color space provides intuitive color component separation, leveraging the characteristic red dominance in fire regions. The RGB fire mask is defined as:

$$M_{RGB}(x,y) = \begin{cases} 1;\ if \begin{cases} R(x,y) > G(x,y) \\ R(x,y) > B(x,y) \\ R(x,y) - G(x,y) > 30 \\ R(x,y) > 150 \\ G(x,y) > 50 \\ B(x,y) > 50 \\ R(x,y) > \ 1.2 \times G(x,y) \\ R(x,y) > 1.2 \times B(x,y) \end{cases} \\ 0;\ otherwise \end{cases}$$

where $R(x,y), G(x,y), B(x,y)$ represent the red, green, and blue channel intensities at pixel coordinate (x,y) respectively.

The fire ratio is calculated as:

$$Fire\ Ratio = \frac{\sum fire\ mask}{\sum pixel}$$

3.2.2 YCbCr Color Space Mask

The YCbCr color space separates luminance from chrominance, providing illumination invariance for fire detection. The YCbCr fire mask exploits the distinctive chrominance characteristics of fire.

$$M_{YCrCb}(x,y) = \begin{cases} 1;\ if\ \begin{cases} Cr(x,y) > Cb(x,y) \\ Cr(x,y) \geq 135 \\ Cb(x,y) \leq 120 \\ Y(x,y) \geq 80 \\ Cr(x,y) \geq 1.5 \times Cb(x,y) \end{cases} \\ 0;\ otherwise \end{cases}$$

where $Y(x,y)$ represents luminance, $Cr(x,y)$ represents red chrominance, and $Cb(x,y)$ represents blue chrominance component.

3.2.3 HSV Color Space Mask

The HSV color space separates color information from intensity, providing robustness to lighting variations. The HSV fire mask leverages the characteristic hue and saturation properties of flames.

$$M_{HSV}(x,y) = \begin{cases} 1;\ if\ \begin{cases} 0° \leq H(x,y) \leq 60° \\ S(x,y) \geq 0.5 \\ V(x,y) \geq 0.3 \\ S(x,y) \geq \dfrac{1 - V(x,y)}{2} \\ Cr(x,y) \geq 1.5 \times Cb(x,y) \end{cases} \\ 0;\ otherwise \end{cases}$$

where H(x,y)∈ [0^∘,360^∘] represents hue, S(x,y)∈ [0,1] represents saturation, and V(x,y)∈ [0,1] represents value (brightness). The hue range captures the characteristic red-to-yellow spectrum of flames, while the saturation-brightness correlation ensures vibrant fire-like colors.

3.3 Adaptive Thresholding

3.3.1 Brightness Adaptation Factor

$$\alpha_{brightness} = B_{min} + \frac{B_{span}}{1 + e^{-B_{steepness}\ (B_{norm} - B_{center})}} \quad (1)$$

Where Bnorm is normalized Average Brightness, Bmin is the Minimum brightness, Bspan Brightness Adaptation range, Bsteepness is the Transition steepness factor, Bcenter is Center point of adaptation.

The purpose of this equation is to control how much the threshold adapts to lighting conditions, so that it returns lower values in dark, higher values in bright conditions.

3.3.2 Saturation Amplification Factor

$$\alpha_{saturation} = (S_{base} + S_{slope} \times S_{norm}) \times (1 - B_{comp} \times B_{norm}) \quad (2)$$

Where Snorm is Normalized average saturation, Sbase is base saturation multiplier, Sslope is Saturation effect, Bcomp represents Brightness compensation factor.

This equation adjusts sensitivity based on color intensity and compensates for saturation effects in different lighting conditions.

3.3.3 Temporal Consistency Factor

let R_{avg} = mean(history[-N_{temp}])
let Trend = | Current ration – R_{avg} |

$$\alpha_{temporal} = \begin{cases} T_{persist\ base} + T_{persist\ slope} \times \left[1 - \min\left(\frac{R_{avg}}{T_{base}}, 1\right)\right]; if\ R_{\iota} \\ T_{spike\ penalty}; if\ Trend > S_{threshold} \\ 1; Otherwise \end{cases} \begin{pmatrix} \\ 1 \\ \end{pmatrix}$$

Where Ntemporal for temporal window size, Tpersist base for base persistence multiplier, Tpersist slope for persistence effect slope, Pthreshold for persistence threshold ratio, Tspike penalty for spike penalty multiplier, Sthreshold for spike detection threshold ratio and Tbase for base threshold.

3.3.4 Scene Stability Factor

$$let\ \alpha^2 = variance(history[-N_{stability}] + Current\ Ratio$$

$$let\ Norm_{variance} = \min\left(\frac{\alpha^2}{{T_{base}}^2 \times V_{scale}}, 1\right) \quad (2)$$

$$\alpha_{stability} = S_{base} + (S_{span} \times Norm_{variance})$$

Where $N_{stability}$ is the Stability window size, V_{scale} for Variance normalization scale, S_{base} for Base stability multiplier and S_{span} Stability effect span.

This equation measures scene consistency and it's more conservative for erratic inputs, more aggressive for stable scenes.

3.3.5 Dynamic Threshold

Using the factors from equation (1) (2) (3) (4) to calculate the dynamic threshold using the following equation:

$$T_{dynamic} = T_{base} \times \alpha_{brightness} \times \alpha_{saturation} \times \alpha_{temporal} \times \alpha_{stability} \quad (3)$$

$$T_{dynamic} = max\left(T_{min}, min\left(T_{dynamic}, T_{max}\right)\right)$$

3.4 Implementation

The proposed ACMA framework is implemented through two core algorithms: Dynamic Threshold Calculation for Adaptive Fire Detection and Adaptive Conditional Model Activation. The first algorithm computes a dynamic threshold by analyzing frame brightness and adjusting it using multiple factors brightness, saturation, temporal stability, and persistence ensuring adaptability to varying lighting conditions. This threshold is bounded within predefined limits to maintain robustness. The second algorithm applies a color-based mask to estimate fire presence and conditionally activates the YOLO model only when the computed fire ratio exceeds the dynamic threshold and activation constraints are satisfied. This selective execution strategy significantly reduces computational overhead while preserving acceptable detection accuracy. System metrics, including activation history and threshold values, are continuously updated to optimize performance. For all experiments, parameter values were configured as shown in Table 1, covering brightness, saturation, temporal, and stability factors, along with operational bounds. These settings enable the ACMA framework to dynamically balance efficiency and reliability in edge-based fire detection scenarios.

Algorithm 1: Dynamic Threshold Calculation for Adaptive Fire Detection

Input: *frame, current_fire_ratio, base_threshold, history*

Output: *dynamic_threshold*

$brightness \leftarrow \frac{1}{HW}\sum_{i=1}^{H}\sum_{j=1}^{W}\frac{I_R(i,j)+I_G(i,j)+I_B(i,j)}{3}$

light_condition ← CLASSIFY_LIGHT_CONDITION (brightness)

$\alpha_{brightness} \leftarrow$ *COMPUTER_BRIGHTNESS_FACTOR (brightness)*

$\alpha_{saturation} \leftarrow$ *COMPUTER_SATURATION_FACTOR (brightness)*

$\alpha_{temporal} \leftarrow$ *COMPUTER_TEMPORAL_FACTOR (brightness)*

$\alpha_{stability} \leftarrow$ *COMPUTER_STABILITY_FACTOR (brightness)*

$T_{dynamic} \leftarrow T_{base} \times \alpha_{brightness} \times \alpha_{saturation} \times \alpha_{temporal} \times \alpha_{stability}$

$T_{dynamic} \leftarrow max\ (T_{min}, min\ (T_{dynamic}, T_{max}))$

$H \leftarrow$ *UPDATE_TEMPORAL_HISTORY (H,* $r_{current}$*)*

Algorithm 2: Adaptive Conditional Model Activation

Input: frame, frame_count, base threshold, mask type, jump interval, activation history

Output: processed frame, metrics

$M_{fire} \leftarrow$ *APPLY_COLOR_MASK (frame, mask type)*

$P_{fire} \leftarrow$ *COUNT_NON_ZERO(*M_{fire}*)*

$R_{fire} \leftarrow \frac{P_{fire}}{H \times W}$

$T_{dynamic} \leftarrow$ *CALCULATE_DYNAMIC_THRESHOLD (frame,* R_{fire}*,* T_{base}*)*

Yolo_activated ← 0

If $R_{fire} > T_{dynamic}$ *then*

  If frame_count > N_{skip} *and NO_RECENT_ACTIVATION (A,* N_{skip}*) then*

    $R_{yolo} \leftarrow$ *YOLO_MODEL.PREDICT(I)*

    $I_{out} \leftarrow$ *ANNOTATE_RESULTS (frame,* R_{yolo}*)*

    Yolo_activated ← 1

  Else

    $I_{out} \leftarrow I$

  End if

Else

  $I_{out} \leftarrow I$

End if

A←UPDATE_ACTIVATION_HISTORY (A, yolo_activated)

metrics ← COLLECT_SYSTEM_METRICS ($T_{dynamic}$*, yolo_activated,* P_{fire}*)*

The complete configuration of adaptive factors and operational bounds used in all experiments is provided in Table 2. These fixed parameters ensure consistent evaluation and reproducibility across the baseline and ACMA enhanced pipelines.

Table 2: parameters used for this paper

Parameter Group	Variable	Value
Brightness	B_{min}	0.5
	B_{span}	1.5
	$B_{steepness}$	8
	B_{center}	0.3
Saturation	S_{base}	0.7
	S_{slope}	0.6
	B_{comp}	0.3
Temporal	$N_{temporal}$	10
	$T_{persist\ base}$	0.7
	$T_{persist\ slope}$	0.2
	$P_{threshold}$	0.5
	$T_{spike\ penalty}$	1.2
	$S_{threshold}$	0.2
Stability	$N_{stability}$	10
	V_{scale}	1.0
	S_{base}	0.8
	S_{span}	0.4
Bounds	T_{min}	0.001
	T_{max}	0.2

4. TESTING AND RESULTS

4.1 Testing environment

The experimental evaluation was conducted on two hardware platforms to assess performance under varying resource constraints: a desktop with an Intel i5 CPU (for development and benchmarking) and a Raspberry Pi 4B (for real-world edge deployment validation). The baseline for comparison was a continuous YOLO inference pipeline, where the detector processes every frame without gating. All tests used YOLOv8n implemented in PyTorch with OpenCV for frame processing, and system metrics (FPS, CPU, memory) were logged for both the baseline and the ACMA-enhanced pipeline.

The pseudo-code used for yolo testing in this experiment is as follows:

Algorithm 3: YOLO testing

Input: video path, model path

Output: metrics

video ← READ_VIDEO (video path)

model ← YOLO (model path)

While True do

    frame ← READ_FRAME (video)

    Result ← model(frame)

    Metrics ← CALCULATE_METRICS (fps, cpu, memory)

End

4.2 Scenario-Based Evaluation

The evaluation includes diverse environmental scenarios commonly encountered in real-world fire surveillance, including night-time illumination, smoke-dominant frames, and chromatically ambiguous lighting conditions. These scenarios stress-test the color-based gating mechanism and expose failure modes that are subsequently analyzed and discussed.

4.3 Performance Evaluation on Desktop Hardware

4.3.1 FPS

As demonstrated in Figures 2 and 3, ACMA YOLO achieved a remarkable performance improvement, processing approximately 60 FPS compared to the baseline YOLO's 2.4 FPS. This represents a 25× performance improvement while maintaining comparable accuracy.

The performance gains stem from two key mechanisms in the proposed approach:

1) Intelligent Frame Filtering: ACMA's color-based pre-filtering eliminates unnecessary YOLO processing on frames with low fire probability.

2) Temporal Frame Skipping: After positive fire detection, ACMA strategically skips subsequent frames where fire persistence is likely.

The performance graphs shown in Figures 2 and 3 clearly illustrate the dynamic FPS adaptation, particularly showing the frame rate surge during active skipping phases following fire detections. This demonstrates ACMA's ability to dynamically allocate computational resources based on scene content and detection confidence.

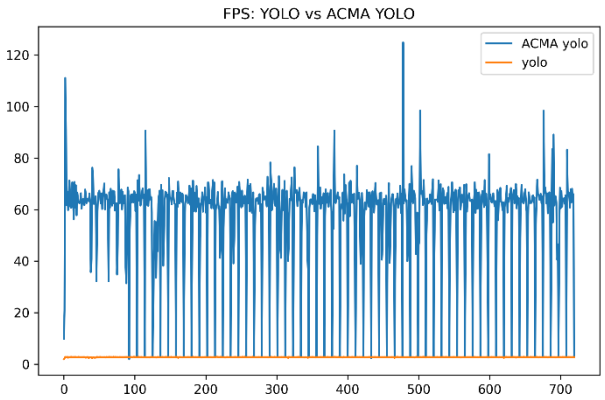

Figure 2: YOLO fps vs ACMA YOLO fps

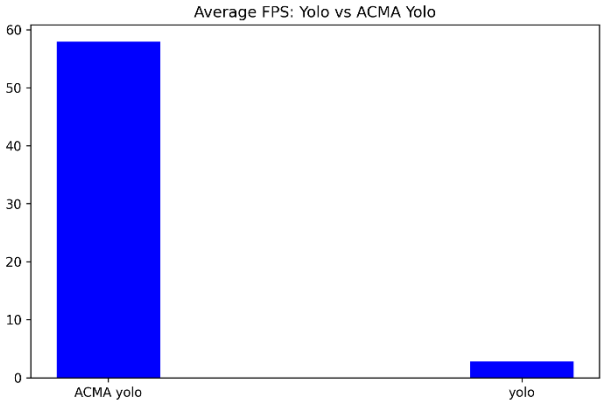

Figure 3: Average fps YOLO vs ACMA YOLO

4.3.2 CPU

The computational efficiency of the ACMA framework is further demonstrated through CPU utilization analysis, as presented in Figures 4 and 5. These results reveal a substantial reduction in processor load when comparing ACMA-enhanced YOLO against the baseline YOLO implementation.

The data indicates that ACMA achieves a significant decrease in CPU utilization, validating the computational efficiency of the filtering mechanism. This reduction in processing overhead directly stems from ACMA's selective execution strategy, where the computationally intensive YOLO model is invoked only when the pre-filtering stage identifies a high probability of fire presence.

The observed CPU usage patterns provide compelling evidence that the proposed solution effectively optimizes resource allocation while maintaining detection reliability. This efficiency gain is particularly valuable for deployment on resource-constrained edge devices, where sustained high CPU utilization would be impractical for continuous monitoring applications.

These results, combined with the frame rate improvements discussed previously, demonstrate that ACMA delivers substantial computational savings without compromising the core detection capabilities of the underlying YOLO model.

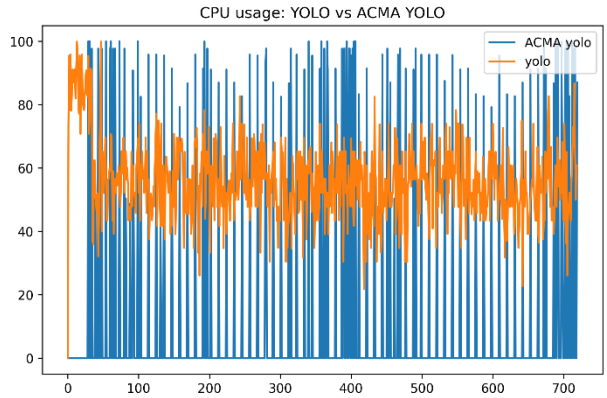

Figure 4: CPU usage YOLO vs ACMA YOLO

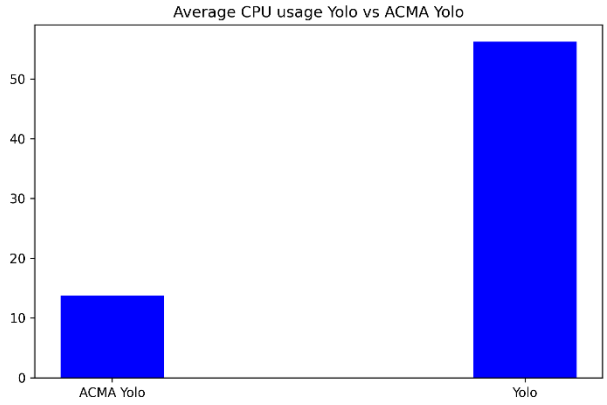

Figure 5: Mean CPU usage YOLO vs ACMA YOLO

4.3.3 Memory

The analysis of Figures 6 and 7 indicates that employing ACMA YOLO model results in an increase in memory consumption by approximately 80MB. This fluctuation is primarily attributable to the processes involved in deactivating and reactivating the YOLO model during operation. Such memory usage patterns are typical in models that require dynamic activation, and understanding this behavior is crucial for optimizing system performance and resource management in deployment scenarios.

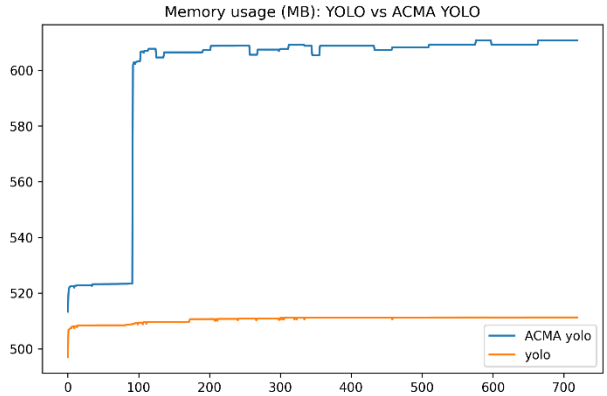

Figure 6: Memory usage (mb) YOLO vs ACMA YOLO

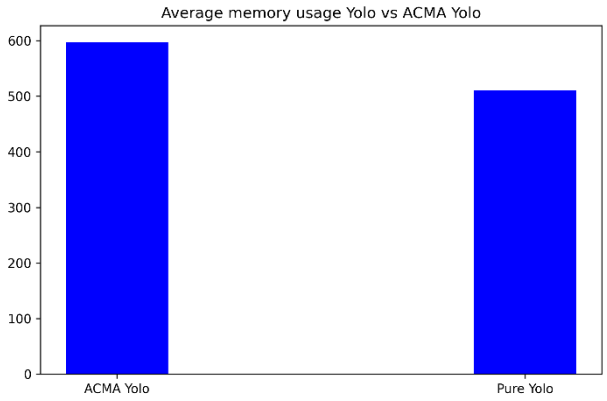

Figure 7: Average memory usage (mb) YOLO vs ACMA YOLO

4.4 Performance Evaluation on Raspberry Pi 4B

4.4.1 FPS

To evaluate the effectiveness of the ACMA framework, Performance tests conducted performance tests by deploying both the baseline YOLO model and the ACMA-enhanced YOLO model on a Raspberry Pi 4B. This setup was designed to compare their operational efficiency in a resource-constrained edge computing environment. Each model was tested under identical conditions to measure key performance metrics, including inference speed, accuracy, and resource utilization. The results shown in Figures 8 and 9 highlight how the ACMA framework improves model performance on low-power devices, demonstrating its suitability for edge applications that require real-time processing and low latency. This comparative analysis underscores the practical benefits of integrating ACMA with YOLO in real-world scenarios. Overall, the findings confirm that the ACMA-enhanced model achieves an excellent trade-off between accuracy and computational efficiency, enabling more effective AI solutions for distributed environments.

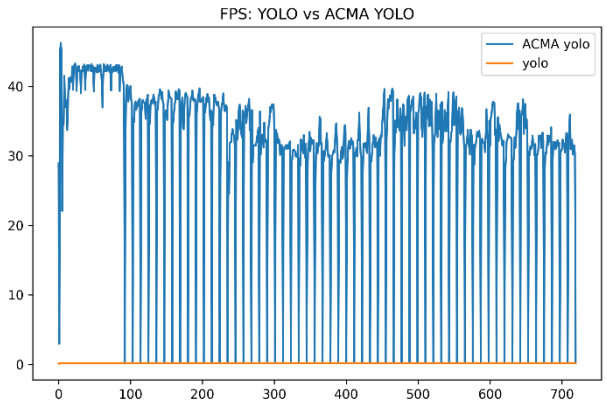

Figure 8: CPU usage YOLO vs ACMA YOLO

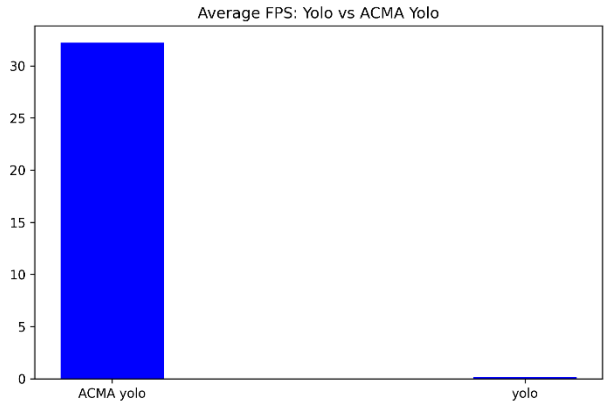

Figure 9: Average CPU usage YOLO vs ACMA YOLO

4.4.2 CPU

The ACMA YOLO model achieved an average CPU utilization of 95%, compared to 99.8% (~100%) for the baseline YOLO model. This represents a reduction of approximately 5%. While this decrease may appear modest, it is significant given that CPU usage cannot exceed 100%. The impact of this improvement was evident in processing time: the ACMA model completed all 729 frames within a few minutes, whereas the original YOLO model required nearly two hours to process the same workload. The difference in processing speed was remarkable in Figures 10 and 11. These findings highlight how even small reductions in resource consumption can lead to substantial performance gains in edge computing environments.

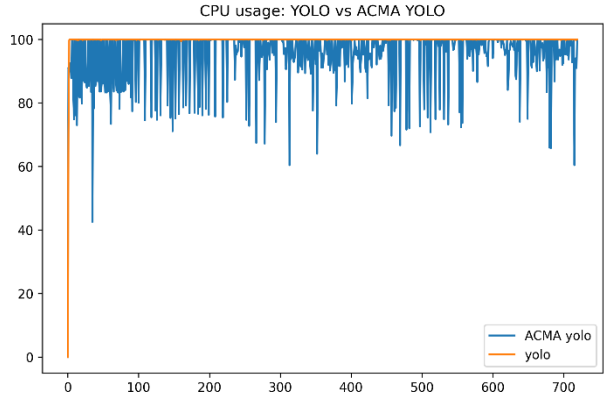

Figure 10: Fps YOLO vs ACMA YOLO

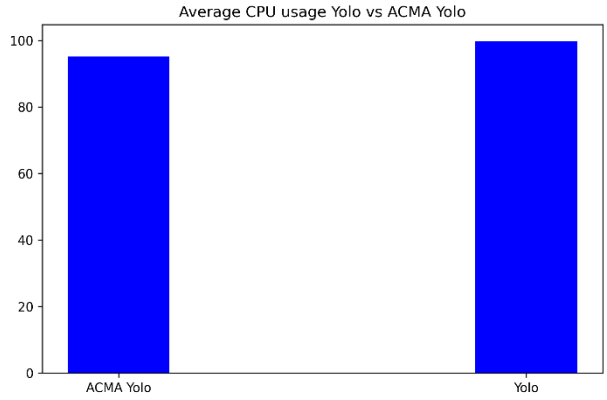

Figure 11: Average fps YOLO vs ACMA YOLO

While ACMA achieved a substantial 80% reduction in CPU utilization on an Intel i5 processor, a more modest reduction of approximately 5% was observed on the Raspberry Pi 4B. This apparent discrepancy arises from fundamental differences in hardware resource constraints and measurement semantics. On the i5 a multi-core desktop processor baseline YOLO inference operates below CPU saturation, allowing ACMA's selective execution to dramatically lower usage. In contrast, Raspberry Pi's resource-constrained environment causes baseline YOLO to persistently operate near full CPU saturation (~99.8% utilization), leaving minimal headroom for percentage-based reduction. However, this modest drop in CPU percentage belies the true efficiency gains: ACMA reduced per-frame processing time drastically, increasing throughput from 0.14 FPS to 38 FPS a 270× improvement. Thus, in severely constrained edge settings, traditional CPU utilization metrics can underestimate real performance improvements, which are more accurately reflected in throughput and latency measures.

4.4.3 Memory

The ACMA model maintained the same memory offset on the Raspberry Pi, with an increase of approximately 80 MB ±20MB in memory usage as shown in Figure 12 and 13

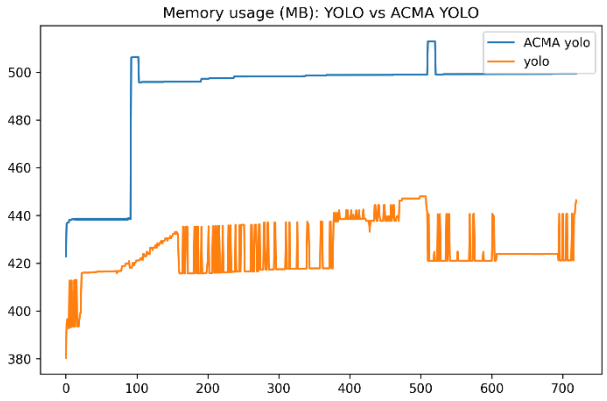

Figure 12: Memory usage YOLO vs ACMA YOLO

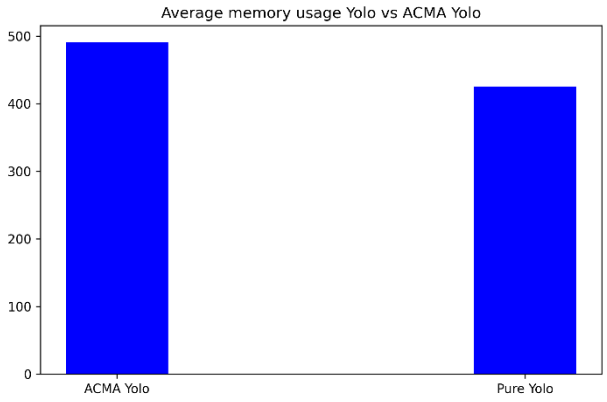

Figure 13: Average fps YOLO vs ACMA YOLO

4.5 Accuracy and Safet Testing

For evaluation purposes, the ACMA-gated YOLO pipeline was applied to the test dataset, with skipped frames archived separately. The baseline YOLO detector was subsequently executed on these frames, and fire detections were recorded as false rejections.

The evaluation methodology focuses on identifying frames where YOLO detected fire but were skipped by ACMA. Note that frames containing fire but not detected by YOLO are not counted as false rejections, as ACMA's purpose is to optimize YOLO execution rather than detect fire independently.

4.5.1 Calculation Methodology

The miss rate is determined through a specific calculation method, which involves analyzing the proportion of errors or misses relative to the total attempts or observations. This metric is crucial in various fields such as quality control, diagnostics, and performance evaluation, as it provides insight into the accuracy and reliability of a process or system. To compute the miss rate, the following formula is typically used:

$$\text{ACMA accuracy} = 1 - \frac{\text{number of false rejections}}{\text{total rejections}} \times 100\% \quad (6)$$

Understanding and accurately calculating the miss rate allows organizations and researchers to identify areas for improvement, optimize processes, and enhance overall effectiveness. It is essential to consider the context in which the miss rate is applied, as different fields may have specific definitions of what constitutes a 'miss' and how the total attempts are counted. Proper data collection and analysis are vital to ensure the validity of the calculated miss rate, which ultimately supports informed decision-making and strategic planning.

This metric provides a direct measure of the system's ability to avoid missing frames with fire, which is critical for safety and reliability.

4.5.2 Results and Significance

The safety testing yielded promising results, as summarized in Table 3. Analysis of the skipped frames revealed a false rejection rate of 23% among those frames. When considered over the entire video stream, this corresponds to a potential missed detection rate of 3.2% of all frames, calculated based on false rejections identified through post-hoc YOLO analysis. This metric, distinct from the model's direct accuracy reduction, represents an upper bound on the proportion of fire-containing frames that the ACMA gating mechanism may incorrectly bypass. These results confirm that the ACMA filtering strategy maintains a high safety margin, incorrectly skipping only a minimal fraction of frames that may contain fire.

Table 3: Testing results

Key	Value
Total frames	1300
Skipped frames	183
False Rejections	32
True Rejections	151
ACMA Accuracy	77%
Baseline YOLO accuracy	94%
ACMA+YOLO accuracy	90.8 %

4.6 Summary of Quantitative Results

The quantitative performance impact of each efficiency method is summarized in Table 4. The results demonstrate that the ACMA framework achieves a transformative reduction in detector activation rate to 14%, which directly enables its orders-of-magnitude improvements in throughput (FPS) across both desktop and severely constrained edge hardware.

Table 4: Overall performance summary

Platform	Method	Activation Rate	FPS	CPU Usage	Memory	Accuracy Drop
i5 CPU	YOLO	100%	2.4	100%	Baseline	0%
i5 CPU	ACMA + YOLO	14%	60	80%	+80 MB	3.2%
RPi 4B	YOLO	100%	0.14	100%	Baseline	0%
RPi 4B	ACMA + YOLO	14%	38	95%	+80 MB	3.2%

Table 2: Comparing other methods' results with ACMA

Method	Activation Rate	FPS Gain	CPU use	Accuracy
Baseline YOLO	100%	0	100%	Baseline Accuracy
Static Pre-filter	100%	Not Addressed	~100 %	+7%
Early-Exit	100%	~ 2x (>50% latency reduction)	~100 %	Ranges from -1% to -5%
ACMA+YOLO i5	14%	25x	80%	-3.2%
ACMA+YOLO RPi 4B	14%	270x	5%	-3.2%

5. DISCUSSION

The evaluated ACMA framework advances edge-based fire detection by significantly improving processing performance, achieving a 25-fold increase from 0.14 FPS to 38 FPS on devices like Raspberry Pi 4B, with minimal accuracy reduction. It employs a selective execution strategy activating the YOLO model only when preliminary indicators suggest fire, balancing efficiency and safety. The framework demonstrates a filtering accuracy of 77%, correctly identifying non-critical frames but with a 23% false rejection rate and 32 missed detections, mainly in low-contrast or rapidly changing lighting conditions. Environmental factors such as fog, haze, and artificial lighting can impact accuracy due to reliance on color features. Memory usage increases by approximately 80MB, which remains manageable for edge devices but offers room for optimization. Compared to static methods like quantization and pruning, ACMA's dynamic, content-aware approach provides superior performance. Its adaptability makes it suitable for real-world

applications with limited resources, though challenges remain in handling chromatic ambiguities and environmental variability (figures 14 and 13). Future improvements could include integrating additional features like thermal data or motion analysis to enhance robustness. Overall, the ACMA framework marks a significant step forward in deploying efficient, reliable fire detection systems at the edge, balancing high performance with safety considerations.

Figure 14: False rejected image in night view mode

Figure 15: False reject image due to smoke effect

6. CONCLUSION AND FUTURE WORK

This paper presented the Adaptive Conditional Model Activation (ACMA) framework, a system-level gating strategy designed to optimize deep learning–based fire detection on resource-constrained edge devices. By selectively activating a computationally expensive YOLO detector only when low-cost visual indicators exceed adaptive thresholds, ACMA significantly reduces redundant inference while maintaining safety-critical detection performance.

Experimental evaluations on both desktop and edge hardware demonstrate that ACMA reduces CPU utilization by up to 80% on non-saturated systems and achieves throughput improvements of up to 270× on severely constrained platforms such as Raspberry Pi 4B, with an accuracy degradation limited to 3.2%. These results confirm that conditional execution is a viable alternative to conventional model compression techniques for real-time edge intelligence.

Future work will focus on enhancing robustness under chromatically ambiguous conditions through multimodal sensing, including thermal and motion cues. Additionally, learning-based adaptive threshold optimization and integration with quantization and pruning techniques will be explored to further reduce memory overhead and improve scalability across heterogeneous edge platforms.

7. REFERENCES

[1] Celik, T.: Fast and efficient method for fire detection using image processing. ETRI J. 32(6), (2010).

[2] Celik, T.: Fast and efficient method for fire detection using image processing. ETRI J. 32(6), (2010).

[3] Ostroukh, E.N. et al.: Color detection algorithm for early fire diagnostics. J. Phys.: Conf. Ser. 2131, (2021).

[4] Mazur, M.: Analysis of methods for reducing the number of false alarms in video-based fire detection systems. J. Autom. Electr. Eng. 5(2), (2023).

[5] Poojary, R., Pai, A.: Comparative study of model optimization techniques in fine-tuned CNN models. In: Proc. International Conference on Electrical and Computing Technologies and Applications (2019).

[6] Liang, D. et al.: Fire and smoke detection method based on improved YOLOv5s. In: Proc. International Conference on Communication, Image and Signal Processing (2023).

[7] Hoang, V.H. et al.: Enhancing fire detection with YOLO models: a Bayesian hyperparameter tuning approach. Comput. Mater. Contin. 83(3), (2025).

[8] He, Y. et al.: DCGC-YOLO: the efficient dual-channel bottleneck structure YOLO detection algorithm for fire detection. IEEE Access 12, (2024).

[9] Wang, H. et al.: YOLO-LFD: a lightweight and fast model for forest fire detection. Comput. Mater. Contin. 82(2), (2025).

[10] Mahmoudi, S. et al.: Edge AI system for real-time and explainable forest fire detection using compressed deep learning models. In: Proc. 20th International Joint Conference on Computer Vision, Imaging and Computer Graphics (2025).

[11] Li, C. et al.: Intelligent monitoring of tunnel fire smoke based on improved YOLOX and edge computing. Appl. Sci. 15(4), (2025).

[12] Xiao, L. et al.: EMG-YOLO: an efficient fire detection model for embedded devices. Digit. Signal Process. 156, (2025).

[13] Vazquez, G. et al.: Detecting wildfire flame and smoke through edge computing using transfer learning enhanced deep learning models. arXiv preprint arXiv:2501.08639 (2025).

[14] Ryu, J., Kwak, D.: A method of detecting candidate regions and flames based on deep learning using color-based pre-processing. Fire 5(6), (2022).

[15] Vincent, G. et al.: Optimizing fire detection in edge devices: integrating early exits in compact models. In: AIAA Aviation Forum (2024).

[16] Gong, L. et al.: Explainable semantic federated learning enabled industrial edge network for fire surveillance. IEEE Trans. Ind. Inform. (2024).