CIT. Journal of Computing and Information Technology, Vol. 33, No. 3, September 2025, 157–181
doi: 10.20532/cit.2025.1005973

A Lightweight Real-time Fire Detection Framework for IoT Devices Utilizing Fine-tuned YOLOv10 and Accelerator Module

Trong Thua Huynh[1], De Thu Huynh[2], Du Thang Phu[1] and Anh Hao Nguyen[1]

[1]Posts and Telecommunications Institute of Technology, Ho Chi Minh City, Vietnam
[2]The Saigon International University, Ho Chi Minh City, Vietnam

This paper presents a novel real-time fire detection framework tailored for IoT devices by integrating the fine-tuned YOLOv10 model with the Accelerator module. Trained on the FireSmokeDataset (Roboflow) and an additional dataset we collected via Roboflow, the system covers fire, smoke, and distracting objects. Optimized for resource-constrained edge devices, the framework demonstrates exceptional performance, achieving high mean average precision (mAP) for fire and smoke detection, with metrics exceeding 84% and a maximum mAP50 of over 91%. We target deployments in residential homes, industrial facilities, and forest monitoring stations. A key contribution of the proposed framework is the construction of a diverse dataset encompassing fire, smoke, and distracting objects – an element often overlooked in existing fire detection datasets. Additionally, fine-tuning the YOLOv10 model components in conjunction with hardware acceleration ensures both prediction accuracy and improved inference response performance. Comprehensive evaluations confirm the system's robustness, scalability, and practicality under various operating conditions. Through experimental analysis, the YOLOv10-S (small) model stands out for its balance between efficiency and resource usage, making it a suitable choice for low-cost real-time applications with resource constraints. By utilizing the Coral Accelerator, the proposed framework reduces inference time by 58% compared to CPU-based implementations, achieving a latency of just 1.7 seconds per frame. The system's lightweight design ensures reliable deployment in remote areas with limited computational resources and unstable network connectivity, maintaining high accuracy while minimizing false alarms.

ACM CCS (2012) Classification: Computing methodologies → Artificial intelligence → Computer vision → Computer vision tasks → Scene anomaly detection

Keywords: accuracy, inference time, coral accelerator, edge computing, resource-constrained environments

1. Introduction

Fire-related incidents are a global challenge, causing annual economic losses estimated at billions of dollars and tragically claiming thousands of lives. For example, the National Fire Protection Association (NFPA) reported over 1.39 million fires in the United States in 2023 [1]. These fires resulted in an estimated 3,670 civilian fire deaths and 13,350 reported civilian fire injuries. The property damage caused by these fires was estimated at $23 billion. On average, a fire department responds to a fire somewhere in the US every 23 seconds. A home structure fire was reported every 95 seconds, a home fire death occurred every three hours, and a home fire injury occurred every 52 minutes. More than one-third of the fires (470,000—or 34 percent) occurred in or on structures. Most fire losses were caused by these structural fires, accounting for 3,070 civilian fire deaths (84 percent), 11,790 civilian fire injuries (88 percent), and $14.7 billion in direct property damage (83 percent). Similarly, according to the latest report [2] dated December 29, 2024, a total of 62,132 wildfires were recorded across the United States, burning a total area of 8,865,833.1 acres. The most severely affected states included California, with 8,307 fires burning 1,080,127.8 acres; Texas, with 4,598 fires affecting 1,313,832 acres; and Oregon, with 2,213 fires burning 1,799,831.7 acres. These statistics highlight the severity of wildfires across the United States in 2024 and underscore the urgent need for effective early fire detection systems.

Traditional fire detection methods, such as smoke detectors or conventional camera-based systems, often fail to respond quickly and accurately in critical situations. Recent studies have demonstrated that IoT-based devices, combined with artificial intelligence technologies, can enable real-time fire detection with higher efficiency. For instance, Magalhães *et al.* [3] explored the trade-off between latency and cost when deploying YOLO on Raspberry Pi and cloud servers. The study revealed that although Raspberry Pi-based configurations offer low costs, they fail to meet real-time requirements due to limitations in data processing. PG-YOLO [4] focused on optimizing YOLOv5 for edge devices by reducing the model size by nine times. While this solution improved computational efficiency, it achieved only average inference speed, which is insufficient for rapid response in fast-spreading fire scenarios. Ahmed Saleem Mahdi [5] developed a system using YOLOv5 for wildfire detection in edge computing environments, achieving 98% accuracy but facing challenges in deployment costs and a lack of diversity in the dataset, making it unsuitable for real-world scenarios. Similarly, the research by Talaat *et al.* [6] utilizing YOLOv8 achieved an accuracy rate of 97.1% but heavily relied on cloud processing, leading to latency issues when deployed in edge environments. Additionally, their dataset contained only fire and smoke images, lacking noisy images, and thus failed to represent real-world conditions comprehensively. These studies indicate that while significant progress has been made in fire detection, current solutions still face limitations in ensuring real-time performance in diverse and resource-constrained environments. This underscores the urgent need for more optimized solutions that ensure both high accuracy and real-time detection capabilities.

Recently, some studies have also proposed using Jetson Nano to accelerate wildfire detection systems [7], [8], [9], [10]. However, this device is relatively expensive and less flexible compared to combining IoT devices with a wider variety of models that are more affordable. Additionally, a simpler acceleration module enables easier connectivity and adjustments when configuration changes are needed for specific requirements.

The development of IoT and AI technologies offers transformative solutions to these challenges. Integrating intelligent algorithms with edge computing devices enables real-time processing and decision-making at the source, reducing dependency on centralized systems. In this context, we propose a novel real-time fire detection framework that combines the fine-tuned YOLOv10 model [11] with the Coral Accelerator [12], specifically designed for IoT applications. This approach addresses critical pain points such as latency and accuracy while maintaining a lightweight design suitable for deployment in constrained environments. By incorporating a diverse dataset of both fire and smoke images and fine-tuning the YOLOv10 small model (a fast yet robust model ensuring high prediction accuracy) supported by hardware acceleration devices, the proposed system ensures not only high accuracy but also the ability to deploy effectively in resource-constrained environments, such as remote areas or fire-prone hazardous zones. By leveraging the power of deep learning algorithms combined with advanced edge computing devices, our solution minimizes latency while enhancing fire detection capabilities in real-world conditions.

The remainder of the paper is organized as follows. In Section 2, we discuss modern fire detection methods concerning accuracy and real-time assurance. Section 3 presents the details of the architecture and system workflow. Section 4 presents the method of constructing the fire and smoke dataset and the development process. Section 5 presents the results and discussion. Finally, conclusions are provided in Section 6.

2. Related Works

Numerous fire detection and monitoring solutions have been extensively researched in recent years, with significant contributions focused on improving detection accuracy, computational efficiency, and deployability in real-world environments.

D. Mamadaliev *et al.* proposed an improved smoke and fire detection method based on the YOLOv8n model, incorporating significant architectural changes to enhance accuracy and efficiency [13]. The replacement of the CIoU loss function with WIoUv3 improved the model's focus on critical regions through a dynamic attention mechanism. Additionally, replacing the C2f module with residual blocks enhanced feature extraction capabilities, reduced training and inference time, and streamlined the overall process. The authors also proposed integrating GELAN blocks into the neck of YOLOv8n, further improving training efficiency. This method achieved outstanding results compared to other state-of-the-art algorithms, with a mean average precision mAP50 of 79.4% and improved performance metrics such as precision and recall. However, despite the performance improvements, the use of complex blocks and model transformations required higher computational resources, limiting their applicability in systems with constrained hardware. Moreover, the accuracy was affected by an insufficient training dataset that lacked diversity and did not represent various real-world conditions, such as diverse environments and different types of fire and smoke objects.

Ahmed Saleem Mahdi developed an early wildfire detection system using YOLOv5 in an edge computing environment [5]. While achieving a detection accuracy of 98%, their reliance on Jetson Nano devices limited scalability for cost-sensitive deployments. Talaat *et al.* proposed a Smart Fire Detection System (SFDS) leveraging YOLOv8 for real-time detection in smart cities [6]. Their method achieved a high accuracy rate of 97.1%, but it heavily relied on cloud-based processing, leading to significant latency in edge-based environments. Additionally, the datasets used in both studies lacked diversity, containing only fire and smoke images and omitting noisy images, which prevented them from fully representing real-world environments.

S. Saponara *et al.* proposed deploying YOLOv2 on Jetson Nano and Raspberry Pi cameras in [7]. This solution enables remote monitoring in smart infrastructure. The system provides real-time fire alerts and is suitable for applications in smart cities and transportation. However, while YOLOv2 is a robust model, it has been surpassed by newer versions such as YOLOv8 and YOLOv10 in terms of accuracy and speed. Using an older model reduces the system's ability to accurately detect fire and smoke objects, particularly under complex conditions such as low light or noisy environments. Chenglin Guo *et al.* proposed a system based on Nvidia Jetson Nano and YOLOv5s, accelerated using TensorRT and DeepStream for real-time fire detection [8]. The use of edge computing combined with the Azure IoT platform helped reduce latency and enhance detection efficiency. However, the lack of a diverse and comprehensive data source reduced performance in complex real-world scenarios, such as harsh weather conditions, crowded environments, or the presence of multiple unrelated heat sources. These limitations hinder its feasibility for widespread deployment. The study [9] proposes a real-time wildfire detection method using unmanned aerial vehicles (UAVs) equipped with Jetson Nano. Due to the computational and battery constraints of UAVs, this study employs TensorRT (NVIDIA's inference acceleration library) along with techniques such as Quantization Aware Training (QAT), Automatic Mixed Precision (AMP), and Post-Training Quantization (PTQ) to improve recognition speed. While optimization methods like QAT, AMP, and PTQ enhance inference speed, they also reduce model accuracy, especially in classifying fire images with complex noise. Additionally, Jetson Nano consumes a significant amount of power when running deep learning models, which shortens the UAV's flight time. Similarly, the study [10] presents a deep learning-based surveillance system for wildfire detection and monitoring using UAV. The system utilizes a UAV-mounted camera to capture images and applies deep learning algorithms for early fire detection. The YOLOv8 and YOLOv5 models are compared in terms of fire detection performance, while a CNN-RCNN network is developed to classify images as containing fire or not. This system is integrated with NVIDIA Jetson Nano hardware for real-time data processing and transmission of fire location information to a ground monitoring station, improving response time and timely intervention. However, the weight of the Jetson Nano reduces the UAV's flexibility and flight duration, limiting the system's operational range. Furthermore, the object detection accuracy of the YOLOv5 and YOLOv8 models reaches only 89%, which is lower than the image classification accuracy of 96%. This indicates that the ability to precisely identify the fire's location has not yet reached a high level.

Dong *et al.* introduced PG-YOLO [4], a compressed YOLOv5 variant specifically designed for edge devices, reducing the model size by 9 times while maintaining an accuracy of 93.4%. This approach highlights the potential of lightweight solutions but sacrifices some flexibility for general-purpose object detection. The study utilized the safety-helmet-wearing dataset, making it more suitable for helmet detection than fire detection. Liang *et al.* proposed the MEC-YOLO model [14], which combines a cloud-edge-end architecture with YOLO for vehicle detection tasks. This approach improved detection speed to 93% but primarily focused on vehicle applications rather than fire detection, which involves noisier data and presents greater challenges for achieving high accuracy.

In [15], Weichao *et al.* proposed an efficient and lightweight flame detection model, EFA-YOLO, featuring two key modules: EAConv (Efficient Attention Convolution) and EADown (Efficient Attention Downsampling) to enhance fire detection performance with superior inference speed. Compared to popular YOLO models such as YOLOv5 and YOLOv10, EFA-YOLO reduced model parameters by 94.6% and accelerated inference speed by 88 times, demonstrating significant potential for IoT applications. Firdaus *et al.* introduced a solution integrating YOLOv8 with IoT and multi-functional sensors such as DHT22 and MQ-2 for an early fire detection system [16]. The system achieved high accuracy (mAP50=0.97) and provided alerts via platforms like Blynk and Telegram, highlighting the effectiveness of combining AI and IoT for fire detection. However, this study used a dataset limited to flames and fire, lacking smoke and other noisy images, which restricts the practical applicability of the solution. In [17], St Banerjee *et al.* deployed YOLOv5 on unmanned ground vehicles (UGV) combined with AWS IoT Core for real-time fire detection and alerts. The system also integrated live video streaming and email notifications, showcasing a seamless blend of AI, IoT, and robotics. However, the proposal had limitations, including high prediction times due to YOLOv5's nature and reliance on third-party cloud platforms, which impacted real-time capabilities and increased deployment costs.

Table 1 presents a summary of the key characteristics contributed by recent publications to the research problem that we aim to address in this study.

3. System Architecture

The proposed real-time fire detection system, as shown in Figure 1, utilizes a fine-tuned YOLOv10 deep learning model and is deployed on a Raspberry Pi integrated with a Camera Module for real-time image capture and a Coral Accelerator to enhance the model's inference speed, ensuring fast and efficient data processing. Upon detecting fire-related signs such as flames or smoke, the Raspberry Pi sends the event to Firebase, including Realtime Database for event storage, Authentication for access control, and Storage for image management. Simultaneously, a notification is instantly sent to the mobile application via Firebase Cloud Messaging. Users can access detailed event information through the app, including captured images and timestamps. The core layer of the system includes:

- Raspberry Pi – Acts as the central processor, receiving data from the Camera Module, running the YOLO model, and sending alerts via Firebase. If necessary, it can also activate an alarm system.
- Camera Module – Captures real-time images and connects to the Raspberry Pi via CSI (Camera Serial Interface), ensuring fast and accurate data transmission.

Table 1. Comparative summary of existing fire detection methods.

Proposal	**Dataset**	**Model/ Method**	**mAP50 (%)**	**Inference time (ms)**	**Edge device/ Environment**
Magalhães *et al.* [3]	Custom (only fire, smoke)	YOLO	N/A	High	Raspberry Pi and Cloud
PG-YOLO [4]	safety-helmet-dataset	PG-YOLO (custom)	93.4	N/A	N/A (personalized)
Saleem M.A. [5]	Custom wildfire	YOLOv5	98.0	~100 ms	Jetson Nano
Talaat *et al.* [6]	flame + smoke	YOLOv8	97.1	High	Cloud-based
Saponara *et al.* [7]	N/A	YOLOv2	N/A	N/A	Jetson Nano/RPi
Chenglin *et al.* [8]	Custom (not details)	YOLOv5s + TensorRT + DeepStream	N/A	significantly reduced but not quantified	Jetson Nano (UAV)
Shamta and Demir [9]	Custom UAV wildfire images	YOLO + QAT/ AMP/PTQ	reduced accuracy	improved	Jetson Nano (UAV)
Mamadaliev *et al.* (ESFD-YOLOv8n) [13]	Custom (flame/smoke)	YOLOv8n	79.4	N/A	GPU
Liang & Zhang (MEC-YOLO) [14]	Vehicle detection dataset	YOLO (cloud-edge-end)	93	N/A	Cloud-Edge-End
Weichao *et al.* (EFA-YOLO) [15]	Custom (flame/smoke)	EFA-YOLO	N/A	faster than YOLOv5/ YOLOv10	IoT devices
Firdaus *et al.* [16]	Custom (only flame/fire)	YOLOv8 + DHT22 and MQ-2 sensors	97	N/A	IoT devices (Blynk, Telegram)
Banerjee *et al.* [17]	N/A	YOLOv5 + AWS IoT Core	N/A	High	Unmanned Ground Vehicle

- Coral Accelerator – A hardware accelerator connected via USB, assisting the Raspberry Pi in processing the YOLO model, reducing the load on the main processor, and speeding up fire detection.
- Fine-tuned YOLOv10 model – Detects fire-related signs such as flames and smoke, ensuring fast and accurate detection.

The remaining two layers of the system include:

- Frontend – A user-friendly interface that displays alert information, manages devices, and tracks events. The application enables remote monitoring and ensures timely responses.
- Middleware – A data storage and management infrastructure that continuously updates events and ensures seamless communication between the Raspberry Pi and the mobile application. Firebase also provides flexible scalability, ensuring the system operates stably over time.

Firebase realtime database and Cloud messaging are utilized as the backend for the following reasons:

1. Ease of integration: Firebase provides free SDKs that are easy to integrate with both Android and Raspberry Pi, reducing development time;
2. Low-latency notifications: Firebase Cloud Messaging supports sending push notifications to multiple mobile devices instantly without requiring a custom server setup;
3. Cost: Firebase Realtime database is free up to a certain read/write threshold.

However, the limitations include dependency on the Internet, lack of Quality-of-Service support in Firebase Realtime database compared to some advanced protocols, and the potential spike in Cloud messaging costs when scaling to hundreds of devices. In future work, we plan to explore MQTT (for environments with unstable Internet connectivity) or AWS IoT Core (for large-scale deployments with enterprise support). For now, Firebase meets our requirements for rapid development and prototyping.

The system workflow, as illustrated in Figure 2, outlines the operational steps in the real-time fire detection system. The process begins by updating the timer to adjust the intervals between tasks. The system then captures and processes images for fire detection. If a fire is detected, the system sends a notification and activates a timer to handle and upload the event. The workflow includes steps for managing the detection cycle, minimizing data flow when no fire is detected, and ensuring efficient event data transmission. Additionally, it involves handling transitions between detection and non-detection modes to maintain the system's operational efficiency.

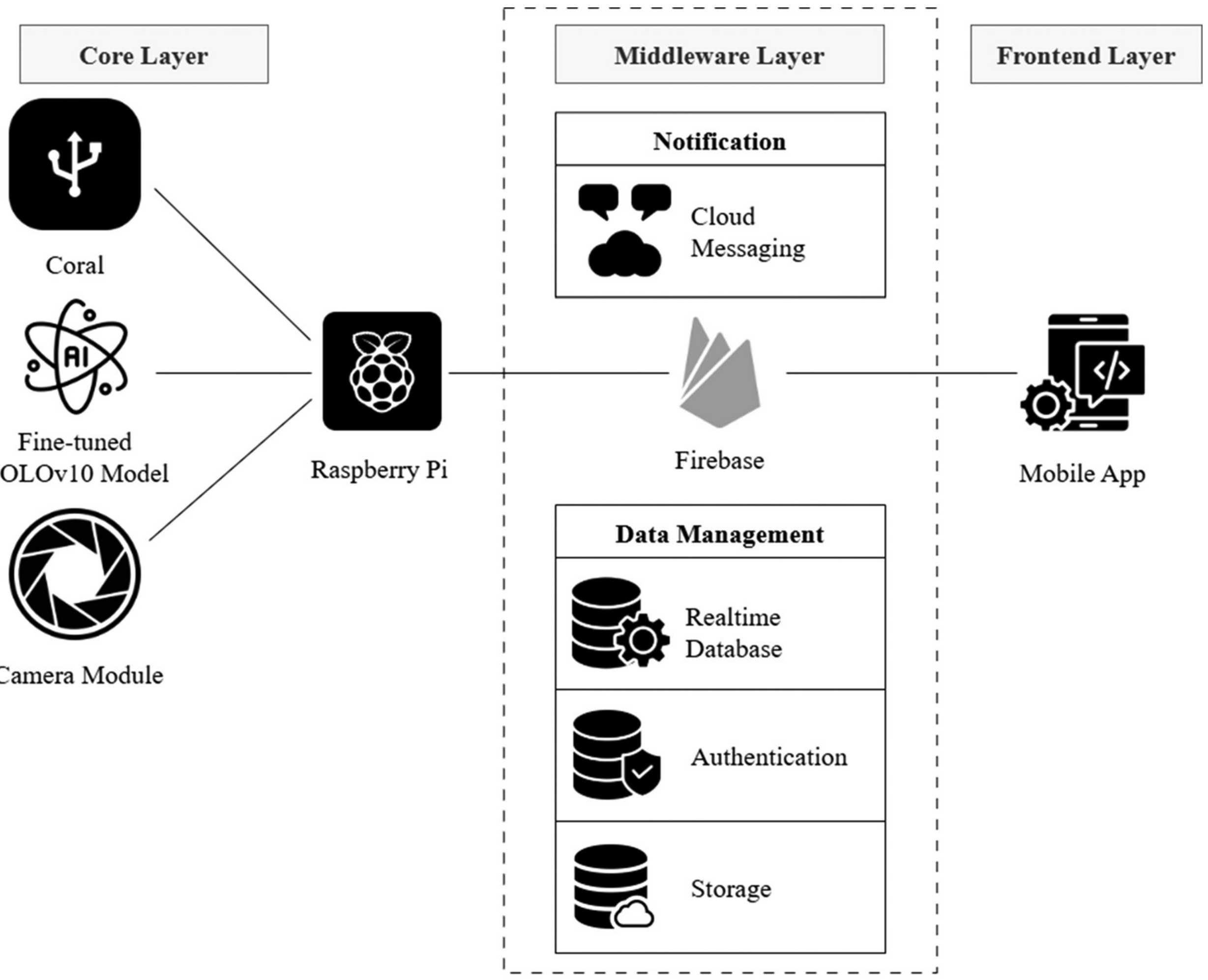


Figure 1. Real-time fire detection system.

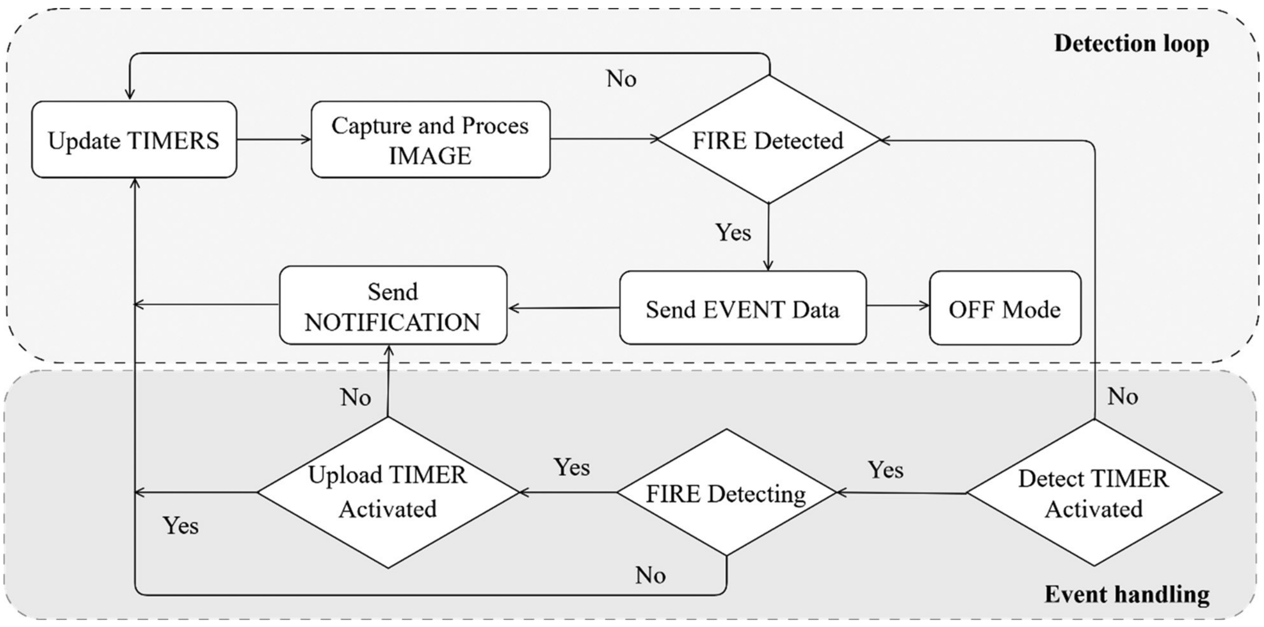


Figure 2. System operation workflow.

The workflow details are as follows:

1. Updating Timers (Update TIMERS): The system updates the timers to establish predefined intervals between tasks, ensuring continuous, smooth operations and efficient responses to real-time fire scenarios.
2. State Management: Local states (*e.g.*, Detect/Upload TIMER Activated, FIRE Detected/Detecting, Send LIMITED) are utilized to manage tasks effectively, ensuring execution only when necessary and preventing unnecessary continuous operations.
3. Non-Detection Mode (OFF Mode): The system transitions to a non-detection mode to reduce data flow and overall load, optimizing performance and conserving resources when no fire events are detected.
4. Activating the Camera Module: The system activates the Camera Module to continuously capture real-time data, ensuring timely image acquisition for accurate fire detection.
5. Processing Captured Images (Process IMAGE): Images captured by the camera are processed using YOLOv10 to detect fire-related signs such as flames or smoke, enabling swift and accurate identification of potential fire hazards.
6. Sending Notifications (Send NOTIFICATION): Notifications are sent to the mobile application interface, ensuring users are promptly informed and can take necessary actions.
7. Uploading Event Data (Send EVENT Data): Event data is sent to Firebase for secure storage and further analysis, ensuring reliable data management and supporting effective tracking of fire-related events.

4. Methodology

4.1. Dataset Preparation

For object detection tasks, the dataset plays a crucial role and must adhere to certain rules to meet the standards of the YOLO model framework. The training dataset is carefully constructed and curated to encompass a wide range of scenarios, including clear fire incidents, smoke without apparent flames, and confounding factors such as strong lighting, vibrant colors, or reflections. The labeling process was conducted with a focus on consistency and the meticulous identification of relevant characteristics. To enhance the model's robustness, data augmentation techniques [18] such as rotation, zooming, noise addition, and more were systematically applied. In this study, we create a unified comprehensive dataset by combining two component datasets: one is a reference dataset (FireSmokeDataset) which is publicly available at [19], and our supplementary dataset (FIRE_DETECTION) can be downloaded from [20]. The goal is to ensure that the model can accurately and efficiently detect and respond to fire incidents across various environmental contexts, which is essential for practical applications.

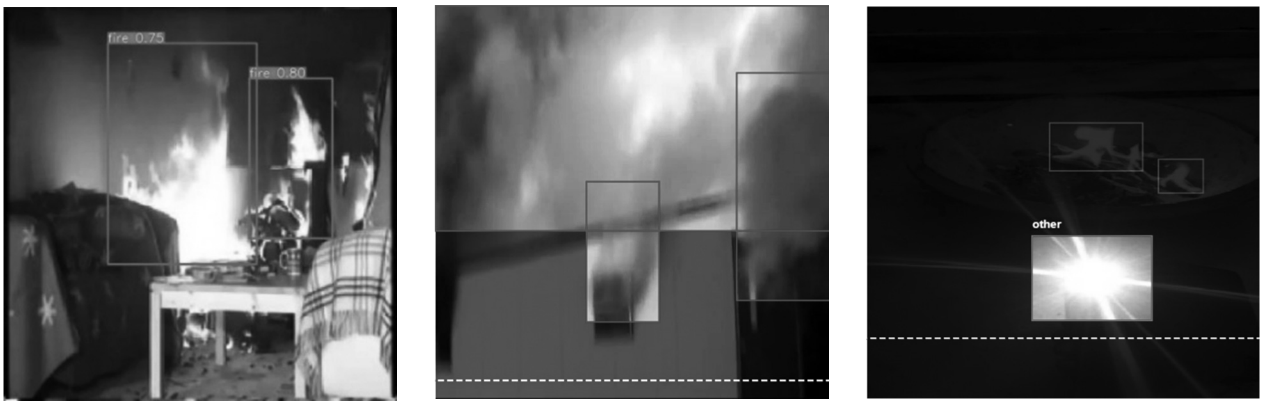
Each image in the dataset belongs to one or more categories (fire, smoke, other) as described in Table 2. The reference dataset contains a total of 34,410 images with 48,718 annotations, categorized into three groups: fire, smoke, and other. The "Fire" group includes 14,725 images with 22,605 annotations, averaging 1.5 annotations per image. These images depict flames in various shapes, colors, sizes, and brightness levels, enabling the model to effectively detect fire signs. The "Smoke" group consists of 16,038 images with 19,378 annotations, averaging 1.2 annotations per image. These images show smoke in diverse shapes, colors, sizes, and densities, allowing the model to detect fire signs even before flames appear. The "Other" group contains 3,647 images with 6,735 annotations, averaging 1.8 annotations per image. These include environmental confounding factors such as light and reflections, helping the model distinguish between actual fire signs and unrelated light sources. This dataset has an average image size of 0.92 megapixels, ranging from 0.03 to 62.10 megapixels, with a median resolution of 1280x720 pixels, ensuring a diverse range of scenarios and high-quality annotations.

Figure 3 shows several annotated images illustrating the three categories (fire, smoke, other). The image on the left is labeled as "fire" (bounding boxes around the flames), the middle image is labeled as "smoke" (bounding boxes around the smoke region), and the image on the right is labeled as "other" (bright light or reflective objects).

In addition to utilizing the publicly shared dataset [19], we expanded the dataset by incorporating our supplementary dataset that we collected ourselves [20], comprising 5,000 additional images. In Table 3, we summarize the distribution of images in the dataset based on area type and label categories. The dataset consists of 5,000 images, including 3,295 indoor images and 1,705 outdoor images. All images comprise 5,882 objects labeled as "fire" and "smoke". Among them, 4,857 objects are labeled with the "fire" class (3,243 indoor and 1,614 outdoor), and 1,025 objects are labeled with the "smoke" class (767 indoor and 258 outdoor). Our supplementary dataset is structured to represent diverse scenarios to enhance the performance and robustness of fire and smoke detection models. This dataset has an average image size of 0.41 megapixels, with a median resolution of

Table 2. Structure of the reference dataset.

Category	**Number of images**	**Number of annotations (Avg per image)**	**Description**
Fire	14,725	22,605 (1.5)	Flames in various shapes, colors, sizes and brightness, helping the model in detecting fire signs.
Smoke	16,038	19,378 (1.2)	Smokes in various shapes, colors, sizes, and densities, aiding the model in early detection of fire signs, even in the absence of flames.
Other	3,647	6,735 (1.8)	The images include environmental confounding factors, helping model distinguish between actual fire signs and unrelated light sources.



(a) Fire sample. *(b)* Smoke sample. *(c)* "Other" sample.

Figure 3. Annotated image samples from each category.

Table 3. The images are supplemented to the dataset from public sources.

Scenario	Images	Objects	
		Fire	Smoke
Indoor	3,295	3,243	767
Outdoor	1,705	1,614	258
Total	5,000	4,857	1,025

640 x 640 pixels. At training phase, we made specific adjustments to seamlessly integrate two datasets, create a unified comprehensive dataset that enhances the model's ability to generalize across various environments.

Notably, our combined dataset comprises approximately 65% indoor images and 35% outdoor images. This imbalance may introduce bias during training, causing the model to favor detecting fire/smoke patterns in indoor environments. In outdoor applications (*e.g.*, forest monitoring), challenging conditions such as harsher lighting and complex backgrounds (*e.g.*, vegetation, direct sunlight) may reduce detection accuracy. To mitigate this, we applied specific augmentation techniques during training - such as adding noise, simulating smoke blurring, and reducing brightness - to mimic outdoor conditions. However, we acknowledge that a larger and more diverse outdoor dataset (*e.g.*, forest scenes, outdoor industrial zones) would significantly improve generalization. Expanding the outdoor dataset is part of our future development direction.

Each image annotation includes a label representing a specific object within the context of the detection task (*e.g.*, "fire," "smoke," or "other" for confounding factors), along with a bounding box that identifies the area containing the object. The bounding box is typically represented as a normalized rectangle to ensure consistency across images of varying sizes. These two main components are detailed through five key fields, as presented in Table 4.

The "class_index" is an integer representing the index of the object class in the configuration file, corresponding to the bounding box in the image. This field is used to identify the type of object being detected. The "x_coordinate" is a floating-point value in the range [0, 1], indicating the horizontal position of the bounding box's center relative to the image width, where 0 represents the left edge and 1 represents the right edge. This normalization ensures the model can process images of different sizes consistently. The "y_coordinate" is a floating-point value in the range [0, 1], indicating the vertical position of the bounding box's center relative to the image height, where 0 represents the top edge and 1 represents the bottom edge. The normalization of both x and y coordinates allows the model to handle images with varying aspect ratios effectively. The "width" is a floating-point value in the range [0, 1], representing the relative width of the bounding box within the image, where 0 indicates no width and 1 indicates the entire width of the image. The "height" is a floating-point value in the range [0, 1], representing the relative height of the bounding box within the image, where 0 indicates no height and 1 indicates the entire height of the image. The normalization of width and height ensures that the bounding box dimensions are represented consistently, regardless of the actual size of the image. This normalization approach enables the model to effectively process images at varying resolutions, ensuring consistent and accurate object detection across different image sizes.

It is evident that the training dataset consists of carefully selected images tailored for detecting fire signs and designed to meet the standards of the YOLO model family, specifically YOLOv10. The use of three labels (fire, smoke, and other) helps minimize false alarms, enhancing the model's practical applicability. To ensure stable performance, various data augmentation techniques such as rotation, cropping, blurring, brightness changes, and adding environmental noise (rain, smoke, fog) were applied during preprocessing to enrich the dataset, enabling the model to generalize better and improve its ability to recognize fire signs across different environments and conditions.

4.2. Model Optimization

The overall architecture of YOLOv10 [11] consists of three main blocks: the Backbone, responsible for feature extraction; the Neck, which synthesizes features; and the Head, which performs fire detection. Derived from [21], we optimize the training model by modifying the internal components of each block to reduce computational complexity while balancing accuracy and detection time summarized in Table 5.

In the Backbone, we utilize the Cross Stage Partial Network (CSPNet) to reduce computational complexity by dividing the base layer into two parts. One part is passed through a ResBlock with bottleneck, improving gradient flow before performing partial transition. Additionally, to minimize computational costs and maximize the retention of important information in each image, we divide the image processing into two stages: spatial downsampling and depth enhancement. During the downsampling stage, we replace Partial Self Attention in YOLOv10 with Self Extraction Attention, a lightweight and computationally efficient mechanism designed to enhance neural network performance by focusing on important features in the data. This improves feature discrimination at the channel level without incurring significant

Table 4. Annotation details for object labels and bounding box information.

	Field	Possible value	Description
Label	<class_index>	integer	The index of the label in the dataset configuration file
Bounding box	<x_coordinate>	[0,1]	The relative coordinate on the x-axis of the center point
	<y_coordinate>	[0,1]	The relative coordinate on the y-axis of the center point
	<width>	[0,1]	The relative width of the bounding box in the image
	<height>	[0,1]	The relative height of the bounding box in the image

Table 5. Architectural modifications from YOLOv10 baseline.

Component	**YOLOv10 baseline**	**YOLOv10-S (modified)**	**Advantage**
Backbone	Partial Self Attention	Self Extraction Attention	Reduce computational cost, improve performance at the channel level
Neck	PAN	PAN + shortcut connections	Enhance multi-scale information flow
Head	Non-Maximum Suppression (NMS)	OneToMany (training) + OneToOne (testing)	Eliminate dependency on NMS, reduce post-processing time
Optimizer	Adam	SGDM (Momentum)	Improve generalization capability, prevent overfitting
Augmentation	rotation, zoom, noise...	rotation, zoom, noise, blur, brightness, fog, rain...	Simulate diverse conditions and reduce bias

costs. In the Neck, to enhance the synthesis of input features from different resolution levels, we introduce shortcut connections into the Path Aggregation Network (PAN). This optimizes the flow of information from lower to higher layers, improving multi-scale feature representation. In the Head, to reduce computational costs during prediction, we replace Non-Maximum Suppression (NMS) with two labeling methods: OneToMany and OneToOne. The OneToMany network is used during training, the OneToOne network is used during testing. This modification reduces the model's dependence on post-processing algorithms and enhances real-time performance.

Furthermore, as highlighted in [21], Adam optimizer often leads to models that perform well on the training set but fail to generalize well on the test set, especially when dealing with highly diverse data. In contrast, Stochastic Gradient Descent with Momentum (SGDM), although slower in convergence, is better at finding global minima, leading to improved generalization on new data. In preliminary experiments, when using Adam optimizer, the mAP@50–95 on the validation set reached approximately 0.56 after 50 epochs but then declined due to overfitting. In contrast, SGDM, although converging more slowly (early stopping occurred around epoch 77), achieved a final mAP@50–95 of 0.588, an improvement of about 3% compared to Adam. This indicates better generalization, particularly on diverse 'other' and 'smoke' images. Thus, to enhance accuracy while ensuring real-time fire detection performance, we replace the Adam optimizer with SGDM in this study.

The training environment is set up on Google Colab, running on a Linux-based system with a Tesla T4 GPU (15,102 MB), providing high computational power for demanding tasks. The environment utilizes Python 3.10.12 along with the Ultralytics 8.3.39 and PyTorch 2.5.1 libraries. This combination of tools ensures efficient execution and optimization of the fine-tuned YOLOv10 model. This algorithm consists of the following key steps: First, the data (fire and smoke prepared earlier) is downloaded and extracted from Roboflow [18]. Then, the training process runs with fine-tuned parameters such as the number of epochs, image size, batch size, and GPU utilization options. During training, metrics such as loss and mAP are monitored to evaluate performance. Once training is complete, the model is evaluated on the validation set and tested for predictions on sample images. Finally, the trained model is saved in a suitable format for deployment.

The fine-tuning process of YOLOv10-S is divided into three main parts as follows: (1) Data splitting: the dataset was split with 8:1:1 ratio for training, validation, and testing, respectively. Specifically, from a total of 39,410 images in the reference dataset plus 5,000 additional images, 35,528 images were used for training, 4,426 for validation, and 4,456 were reserved for final evaluation (test set); (2) Hyperparameters: initial learning rate (LR): 0.01 using SGDM, scheduler: cosine annealing with a warm-up for the first 3 epochs, then gradually reduced following the cosine rule to a final LR of 0.0001 by epoch 100, batch size: 32 images, weight decay: 0.0005, momentum: 0.937, maximum number of epochs: 100, with early stopping if the validation loss does not improve after 5 consecutive epochs; (3) Training procedure: the YOLOv10-S model was initialized with pre-trained weights on the COCO dataset. The first two layers (backbone) were frozen during the first 10 epochs to stabilize feature extraction, after which the entire network was unfrozen and training continued for an additional 90 epochs. The following metrics were monitored after each epoch: train/val losses (box_loss, cls_loss, dfl_loss), precision, recall, mAP@50, and mAP@50–95. The entire training process on a Tesla T4 GPU took approximately 4 hours and 30 minutes (270 minutes) until early stopping at epoch 77. During training, GPU utilization ranged between 85–90%.

Figure 4 illustrates the training and validation metrics of the YOLOv10-S model over 100 epochs, with a batch size of 32, using the SGDM optimization algorithm. The training loss curves for box loss (train/box_loss), classification loss (train/cls_loss), and distribution focal loss (train/dfl_loss) show significant improvements over time, reflecting the model's ability to minimize errors during learning. The validation losses for box loss (val/box_loss), classification loss (val/cls_loss), and distribution focal loss (val/dfl_loss) also gradually decrease, albeit with some fluctuations, indicating the model's capacity to generalize. The precision and recall metrics steadily increased, exceeding 0.75 by the end of training, demonstrating the model's success in detecting fire incidents. The Mean Average Precision (mAP) values, including mAP50 and mAP50-95, indicate strong performance. In particular, mAP50 continues to improve consistently, suggesting that the model's detection capabilities enhance with each epoch. The training process stops early at epoch 77 due to meeting the Early Stopping criteria, as no significant improvement in validation loss is observed over the last five epochs. This prevents overfitting and ensures

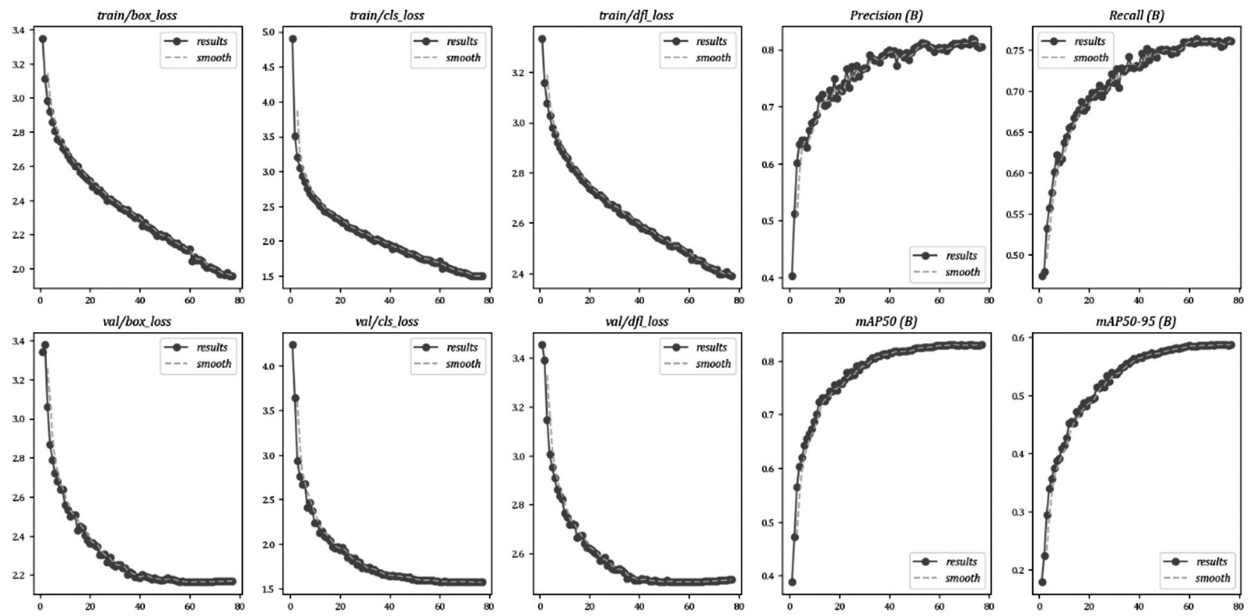


Figure 4. Training result with the fine-tuned YOLOv10-S.

efficiency in training. These results demonstrate that the YOLOv10-S model is learning effectively and shows strong potential for fire detection tasks.

4.3. Deployment Process

The process of converting the model to the TensorFlow Edge TPU format is designed to accelerate and optimize machine learning tasks, making it particularly suitable for applications with constraints on power, computational resources, and connectivity. Edge TPU, a hardware accelerator developed by Google, is specifically built to enhance the performance of TensorFlow Lite models on edge devices. Figure 5 illustrates the implementation of this conversion process.

The process of converting to the Edge TPU TFLite format is essentially an optimization process, compiled through the Edge TPU Compiler into the TensorFlow Lite format. PyTorch Script is the standard format of the PyTorch framework, generated during the training of the YOLO model using the Ultralytics library. Open Neural Network Exchange (ONNX) is an open standard format that facilitates model conversion between different frameworks. Protocol Buffers is the standard format for the TensorFlow framework, serving as the core platform optimized for deploying deep learning models. TensorFlow Lite (TFLite) is a lightweight version of TensorFlow, optimized for deployment on mobile and embedded devices. TensorFlow Lite Edge TPU is the TensorFlow Lite version compiled to run on Edge TPU specialized hardware.

During conversion to Edge TPU format using the Edge TPU Compiler, all operations, including custom layers, were successfully compiled in 99% of the cases. Only 1% of the layers (very large sockets) required fallback to INT8, but this had no significant impact. Specifically, out of a total of 175 layers, only 2 layers were converted using INT8 fallback (1.14%), resulting in a 98.86% full conversion success rate to Edge TPU.

The Edge TPU operates exclusively with models that have been quantized. Quantization reduces the model size and increases its speed without significantly affecting accuracy. This is particularly ideal for edge devices with limited computational resources, enabling faster application response times by reducing latency and processing data locally without relying on cloud technologies. Furthermore, local data processing enhances privacy and security, as data is not transmitted to remote servers. The final model, after training and testing, is converted into the TensorFlow Lite Edge TPU format. Python-based tools such as Ultralytics and PyCoral facilitate seamless deployment on Coral Accelerator. This configuration ensures real-time performance on low-power IoT devices.

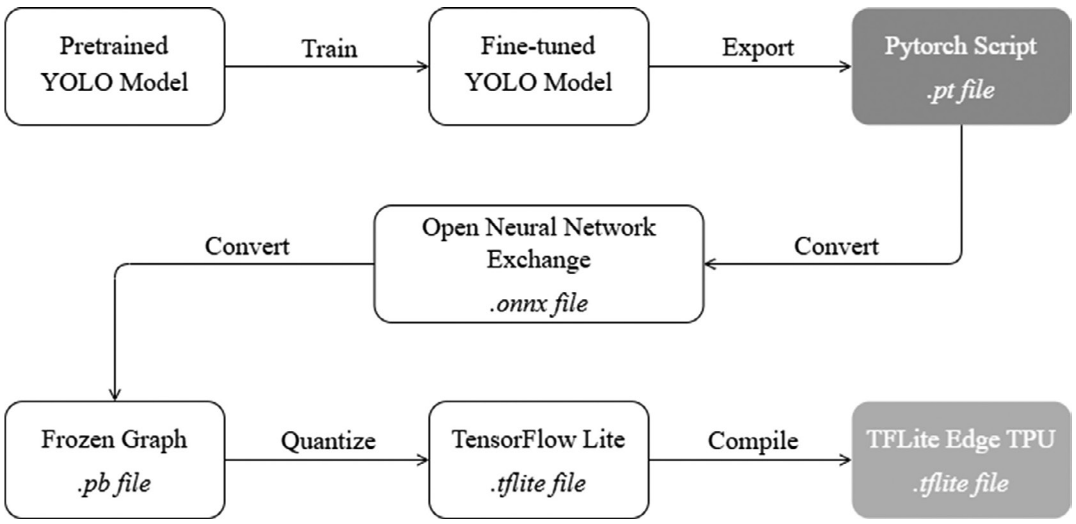


Figure 5. Edge TPU model compilation process.

Algorithm 1. Pseudo code of the Early Fire Detection.

```
Input: captured frames, refined YOLOv10-S model for frame detection, states in firebase
Output: notification/confirmation of fire detection, frames uploaded to firebase
Step 1.  FRAME ← get frame from camera module  //capture frame
Step 2.  STATE ← YOLOv10.detect(FRAME)  //detect state of frame
Step 3.  if Detection_Timer is not active then //check detection timer
             if STATE = True or STATE ≠ Previous_Frame_State then
                 Activate Detection_Timer
                 Previous_Frame_State ← STATE
                 Firebase.update("Detect_State", STATE)
                 if STATE = True then
                     Detecting_State ← True
                     Timestamp ← Current_Time
                     Mobile_App.notify("Fire detected!")
                 end if
             end if
         end if
Step 4.  if Detecting_State = True then   //handle uploading when detecting state is true
             if Upload_Quantity < 3 then
                 if Upload_Timer is not active then
                     Activate Upload_Timer
                     Firebase.store_frame(FRAME)
                     Upload_Quantity ← Upload_Quantity + 1
                 end if
             else
                 Detecting_State ← False
                 Timestamp ← NULL
                 Upload_Quantity ← 0
             end if
         end if
```

The IoT device processing workflow is implemented as described in Algorithm 1. This algorithm utilizes a Camera Module to provide real-time frames and YOLOv10-S to detect the status of the frame (fire or no fire). Firebase is used to store detection statuses and frames when necessary, while a mobile application receives notifications upon fire detection. The algorithm outputs include sending fire notifications to the mobile application and uploading frames to Firebase. The core logic involves classifying frame status using YOLOv10-S, managing a counter to avoid unnecessary repeated actions, updating the detection status in Firebase, and handling notifications along with storing a limited number of frames. Once the specified number of frames is uploaded, the detection status and counter are reset to maintain efficiency. Algorithm 1 is evaluated for performance on Raspberry Pi 4 and Coral USB Accelerator, based on the following main functions (1) get_frame: approximately 30 ms/frame (camera capture); (2) YOLOv10-S.detect(FRAME): approximately 1.7 s/frame (with TPU) or around 4.0 s/frame (with CPU); (3) Firebase.update: approximately 100ms/call (with 250MB/s Internet speed); (4) Firebase.store_frame: approximately 150 ms/frame (including compression and upload). When Detecting_State is false, the total time per cycle is approximately 1.83 seconds per frame, consisting of 30 ms (capture), 1.7 s (TPU inference), and 100 ms (Firebase update). When Detecting_State is true (upload ≤ 3), an additional 150 ms per frame is added for uploading, resulting in a total of approximately 1.98 s/frame.

Algorithm 2. Pseudo code of the Real-time Fire Alert.

```
Input: detect state from firebase, notifications from firebase
Output: updated mobile layout (Safe or Danger), captured data displayed on mobile
Step 1.  DETECT_STATE ← Firebase.listen("Detect_State") //detect state from firebase
Step 2.  if DETECT_STATE = False then // handle state changes
             Mobile_App.update_layout("Safe Layout") // update to safe layout
Step 3.  else
         NOTIFICATION ← Firebase.receive_notification("Fire Alert")  // update layout
         Mobile_App.update_layout("Danger Layout")
         CAPTURED_DATA ← Firebase.load("Captured_Data") // display captured data
         Mobile_App.display_data(CAPTURED_DATA)
         end if
```

The mobile application processing workflow for early fire warnings in monitored areas is implemented as described in Algorithm 2. This algorithm uses Firebase to receive fire detection statuses (Safe or Danger), notifications, and data uploaded from the IoT device in case of a fire. The outputs include displaying the "Safe Layout" interface when no fire is detected and the "Danger Layout" when a fire is detected, along with data such as frames, timestamps, and related information presented on the mobile application. The core logic of the algorithm involves listening to real-time detection statuses from Firebase, dynamically switching the interface based on the detection status, and displaying detailed data when a fire is detected. The algorithm ensures timely fire alerts and provides essential information to the user. Algorithm 2 is evaluated for performance based on the following main functions (1) Firebase. listen: approximately 50 ms; (2) Layout update: approximately 10ms; (3) Firebase.load(CAPTURED_DATA): approximately 150ms (depending on image size).

5. Results and Discussion

5.1. Performance Metrics

The experimental results we conducted to evaluate the metrics comparing the S version with other versions of YOLOv10 are presented in Table 6. The format conversion time and memory usage of YOLOv10 models increase progressively from the N version to the X version. This indicates that larger and more complex models require more resources, with significantly increased processing time and memory usage to ensure stable conversion and operation. This increase is observed regardless of whether the model is deployed to operate on a CPU or an Edge TPU. For all versions of the YOLOv10 model, memory usage in the TPU format is significantly smaller compared to the CPU format. This highlights that the optimized model format for Edge TPU incorporates special optimization techniques to ensure efficient operation on Edge hardware. Consequently, the TPU-format model not only consumes less memory space but also enhances the model's processing performance.

Based on the data from Table 6, compared to YOLOv10-N, the S version shows a significant improvement in mAP50-95, increasing from 39.5% to 46.8% (+7.3%), reflecting a noticeable boost in detection performance. Parameters (Params) increase from 2.3 M to 7.2 M, while still maintaining a moderate size, making it suitable for high-efficiency applications without being overly resource-intensive. Latency increases slightly from 1.84 ms to 2.49 ms, which remains acceptable for real-time applications. Conversion time and memory usage also increase but stay within reasonable limits, ensuring a balance between performance and resource requirements. When compared to YOLOv10-M, the S version is significantly lighter in terms of Params (7.2 M vs. 15.4 M), nearly halving the size, which saves resources in constrained environments. Conversion time is also lower at 495.9 s compared to 872.2 s, making it more suitable for systems that do not require handling large-scale data simultaneously. While the mAP50-95 of YOLOv10-S is slightly lower than YOLOv10-M (46.8% vs. 51.3%), this difference is acceptable for applications that require a balance between performance and resource utilization. Compared to YOLOv10-L and YOLOv10-X (the largest version), YOLOv10-S requires significantly fewer resources. It has Params of 7.2M, which is 3.4 times smaller than YOLOv10-L and more than 4 times smaller than YOLOv10-X. Memory usage on CPU is 15.8MB compared to 49.9 MB and 61.4 MB for YOLOv10-L and YOLOv10-X, respectively, representing a reduction of nearly fourfold. Similarly, Memory (Edge TPU) is only 9.66 MB, about one-third of YOLOv10-X. With a latency of 2.49 ms, YOLOv10-S is much faster than YOLOv10-L (7.28 ms) and YOLOv10-X (10.7 ms), making it more suitable for real-time applications.

Consequence, YOLOv10-S strikes an excellent balance between performance (mAP50-95) and resource usage. Its low latency makes it ideal for real-time applications, while its moderate resource consumption makes it suitable for IoT devices or resource-constrained environments. For higher performance needs, YOLOv10-M or larger versions can be considered. However, for applications that prioritize a balance between performance and resource efficiency, YOLOv10-S is the most suitable choice.

Table 7 shows that Edge TPU significantly improves inference time for YOLOv10 models compared to CPU. For both CPU and Edge TPU, the inference time of the model consistently increases from YOLOv10-N to YOLOv10-X, reflecting the inherent complexity and size of the models. Edge TPU consistently delivers significantly lower inference time than CPU across all model versions. The difference becomes more pronounced as the model complexity increases, highlighting the superior processing performance of Edge TPU. Lightweight versions like YOLOv10-N and YOLOv10-S have very low inference time on both CPU and Edge TPU, making them ideal choices for applications requiring real-time recognition.

The inference speed of YOLOv10-S is significantly faster than larger versions (M, B, L, X). On both CPU and TPU, the S version is faster by 124% - 489% (CPU) and 132% - 480% (TPU), respectively. For instance, compared to the L (Large) version, YOLOv10-S is up to 348% faster on CPU and 351% faster on TPU. This makes the S version well-suited for tasks requiring high speed while maintaining good accuracy, such as object detection on mobile devices, IoT systems, or real-time applications. Additionally, on TPU, YOLOv10-S achieves an inference time of just 1716.16 ms, which is fast enough to handle multiple simultaneous tasks without increasing latency. YOLOv10-S is slower than the N version on both CPU (147% slower) and TPU (144% slower) due to its increased complexity compared to YOLOv10-N.

Table 6. The comparison metrics between the S version and other versions of YOLOv10.

YOLOv10 version	Params (M)	mAP50-95 (%)	Latency (ms)	Conversion time (s)	Memory (MB, CPU)	Memory (MB, Edge TPU)
N	2.3	39.5	1.84	313.8	5.58	3.32
S	7.2	46.8	2.49	495.9	15.8	9.66
M	15.4	51.3	4.74	872.2	32	17.8
L	24.4	53.4	7.28	1,436.1	49.9	26.3
X	29.5	54.4	10.7	1933	61.4	32.1

Table 7. The inference speed of the S version compared to other versions on CPU and Edge TPU.

YOLOv10 Version	CPU (ms)	Edge TPU (ms)	Speed improvement of the S version vs other versions (CPU) (%)	Speed improvement of the S version vs other versions (Edge TPU) (%)
N	1,189.32	703.59	147% slower	144% slower
S	2,938.97	1,716.16	-	-
M	6,598.81	3,980.39	124% faster	132% faster
B	10,531.06	6,041.29	259% faster	252% faster
L	13,149.98	7,737.84	348% faster	351% faster
X	17,297.36	9,954.26	489% faster	480% faster

Despite being slower, YOLOv10-S is rated higher in terms of detection capability and accuracy (as shown in Table 6). This makes YOLOv10-S a more balanced choice between inference speed and accuracy.

Table 8 presents the accuracy results of the YOLOv10-S model in detecting different classes, with metrics such as precision, recall, mAP50, and mAP50-95 evaluated for each category. Overall, the model achieves an average precision of 80.9% and recall of 76.1% across all classes, indicating a good balance between correctly identifying fire-related objects and minimizing false alarms. The "fire" category exhibits particularly high precision (84.8%) and recall (84.1%), corresponding F1 score is 84.4%, reflecting the model's strong capability in detecting fire incidents. The "smoke" category also performs well, with precision of 85.7% and recall of 82.0%, and the corresponding F1 score is 83.8%, demonstrating effective smoke detection. However, the "other" category shows lower values for both precision (72.1%) and recall (62.3%), corresponding F1 score is 66.8%, suggesting room for improvement in distinguishing non-fire-related objects. The mAP50 values further reinforce these observations, with the "fire" (91.1%) and "smoke" (89.6%) categories showing robust performance, while the "other" category has a lower mAP50 of 68.2%. These results highlight that the YOLOv10-S model is highly effective in detecting fire and smoke but may require additional adjustments to improve accuracy in classifying non-fire-related objects.

Table 8. Accuracy of the YOLOv10-S model in detecting different classes.

Category	Precision (%)	Recall (%)	F1 score (%)	mAP50 (%)	mAP50–95 (%)
all	80.9	76.1	78.4	83.0	58.8
fire	84.8	84.1	84.4	91.1	66.6
smoke	85.7	82.0	83.8	89.6	64.0
other	72.1	62.3	66.8	68.2	45.9

The confusion matrix, measured on the test set of 4,456 images, is presented in Table 9, where each value represents the percentage of predicted images relative to the actual number of images in that class. The results show (1) For the "fire" class: only 8% were misclassified as "other" and 1% as "smoke"; (2) For the "smoke" class: only 3% were misclassified as "fire" and 7% as "other"; (3) For the "other" (interference) class: the false alarm rate - *i.e.*, "other" images misclassified as "fire" - reached 21%, and 10% were misclassified as "smoke". This reflects that while misclassification is low for "fire" and "smoke," the model still struggles with the "other" class, which remains frequently confused with the other two.

Reasons for high misclassification rate of the "other" class into "fire" or "smoke" include (1) Class imbalance: the supplemented dataset contains only around 3,647 "other" images, compared to approximately 14,725 "fire" and 16,038 "smoke" images, limiting the model's ability to learn distinct features for "other"; (2) Color/brightness similarity: many "other" images share similar hue/gamma values with flames (*e.g.*, halogen lights, bright sunlight), causing the Self Extraction Attention mechanism to sometimes fail in distinguishing them; (3) Annotation quality: some "other" bounding boxes are too large and include background areas, introducing noise during training. To improve this, in future work, we plan to include harder negative annotations and increase the diversity of "other" images (*e.g.*, light bulbs, artificial flames, cosplay fire) to reduce the false alarm rate.

Table 10 presents the inference performance of the YOLOv10-S model, comparing configurations using CPU and TPU. When using a CPU, the model requires 15.7MB of memory and takes 4,043.1ms per inference, indicating relatively slow processing time on standard hardware. In contrast, the Edge TPU configuration significantly improves performance, reducing inference time to 1,698.1ms and memory usage to 9.28MB, demonstrating the superior benefits of hardware acceleration for faster and more efficient fire detection. These results highlight the potential for deploying the model on embedded systems with Edge TPU support, enabling real-time and efficient fire detection.

Additionally, to evaluate the impact of quantization on the accuracy of the proposed model, we compared the mAP before and after quantization based on the YOLOv10-S version. Details are shown in Table 11 (with INT8 values measured on the Edge TPU). All evaluations are conducted on the same validation set. The mAP degradation is not more than 2.5%, indicating that quantization has minimal impact on accuracy.

Table 9. Confusion matrix for the test set of 4,456 images.

		Predict (%)		
Actual		**Fire**	**Smoke**	**Other**
	Fire	91.0	1.0	8.0
	Smoke	3.0	90.0	7.0
	Other	21.0	10.0	69.0

Table 10. Efficiency of the YOLOv10-S model in detecting different classes.

	Memory (MB)	**Inference Time (ms)**
CPU	15.7	4,043.1
Edge TPU	9.28	1,698.1

Table 11. mAP comparison before vs. after quantization (YOLOv10-S).

Category	**mAP50 (float32) (%)**	**mAP50 (int8 TPU) (%)**	**Δ@50 (%)**	**mAP50-95 (float32) (%)**	**mAP50-95 (int8 TPU) (%)**	**Δ@90 (%)**
all	83.0	82.0	−1.2	58.8	57.3	−2.5
fire	91.1	90.3	−0.9	66.6	65.5	−1.7
smoke	89.6	88.8	−0.9	64.0	63.3	−1.1
other	68.2	67.0	−1.8	45.9	44.5	−3.0

5.2. Deployment Results

Figure 6 illustrates the real-world deployment results of the real-time fire warning system on IoT devices accessed via a smartphone. The system operates by using a Camera Module to capture real-time frames from the environment, which are then sent to a Raspberry Pi for processing. The YOLO model is employed to analyze and detect fire signs. If a fire is detected, the status information and frame data are updated to Firebase. The mobile application accesses Firebase to synchronize data, display status and notify users. When no fire is detected, the app interface shows a "Safe Zone" status with a green icon, indicating a safe area. Conversely, if a fire is detected, the interface switches to "Danger Zone", displaying a "Fire Alert" notification along with the detected image and timestamp. The system also sends emergency notifications to users through the mobile app, providing timely warnings. The final outcome is a real-time fire detection system that delivers detailed information and enables users to act quickly in case of incidents or potential fire hazards.

The integration of YOLOv10 with the Coral Accelerator addresses critical challenges in IoT-based fire detection systems. Key contributions include reduced latency, improved accuracy, and compatibility with resource-constrained devices. The modular design of the system ensures adaptability to various operational environments, from residential safety to large-scale industrial monitoring. Challenges such as detection under low-visibility conditions (*e.g.*, dense smoke or dim lighting) remain and present opportunities for future research. Incorporating additional sensor data (*e.g.*, thermal imaging or gas sensors) could further enhance the accuracy and reliability of detection.

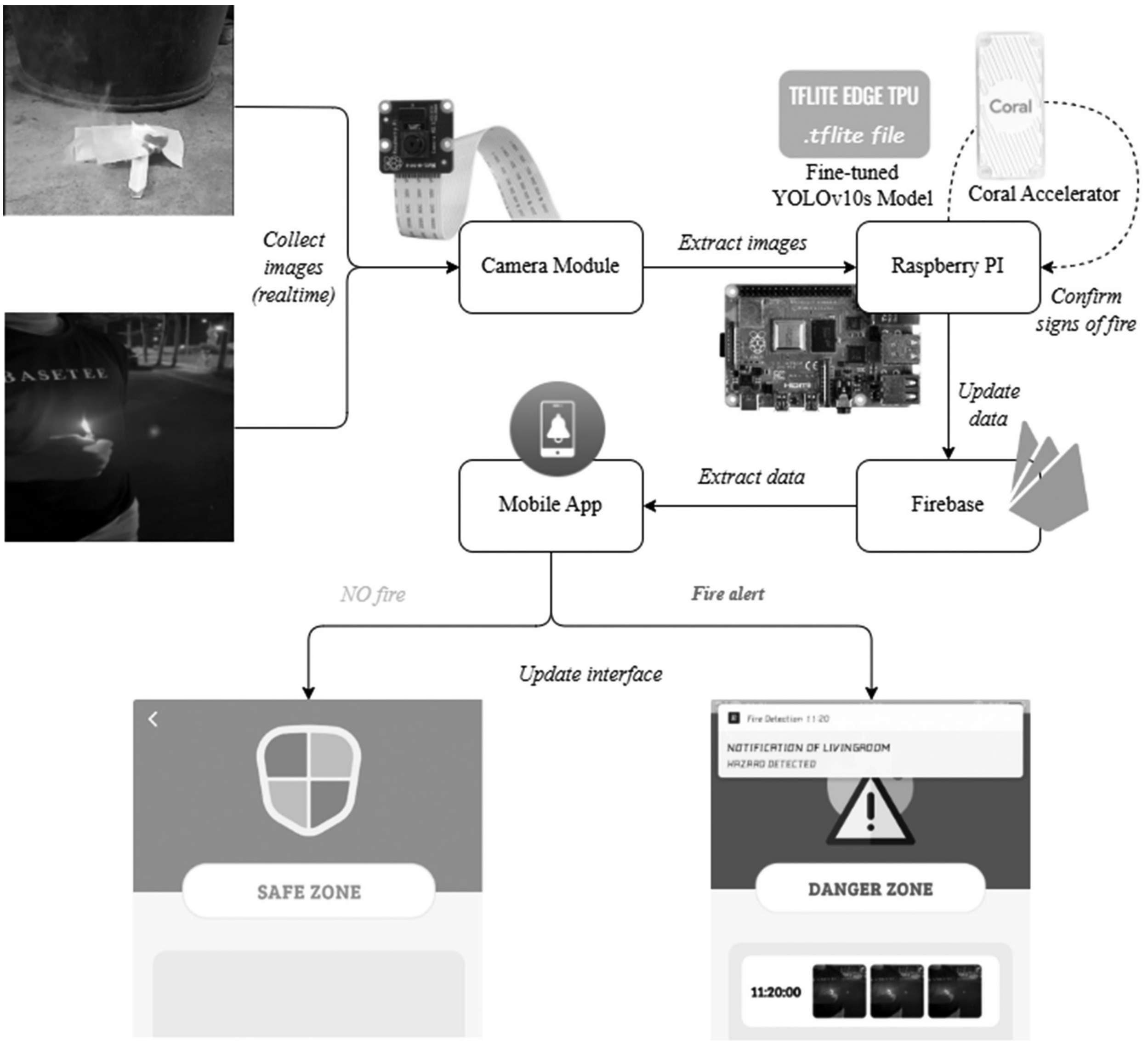


Figure 6. Real-time fire detection system on IoT device.

5.3. Effectiveness of Combining Coral Accelerator with Raspberry Pi

In early fire detection systems, computational performance and power consumption are two critical factors that determine the effectiveness of real-world deployment. Table 12 presents a comparison of key criteria between the Coral Accelerator combined with Raspberry Pi and Jetson Nano.

In terms of power consumption, the Coral Accelerator (Edge TPU) consumes only about 2W, while the Raspberry Pi 4 consumes between 3-7W. The total system consumption remains below 10W, making it suitable for battery or solar-powered operation, optimizing operational costs and mobility. In contrast, Jetson Nano consumes 5−10W in power-saving mode and up to 15W when handling heavy tasks, requiring a more stable power source. This limits its deployment on mobile devices or in remote areas where the power supply is unstable. When deploying outdoor IoT systems, such as forest monitoring or large-scale surveillance, the Coral Accelerator combined with Raspberry Pi offers a significant advantage due to its ability to operate continuously with limited power resources. In terms of AI processing performance, although Jetson Nano has a more powerful GPU, the Coral Accelerator (Edge TPU)

Table 12. The combination of Coral Accelerator with Raspberry Pi vs. Jetson Nano.

Criteria	**Jetson Nano**	**Coral Accelerator with Raspberry Pi**	
		Coral Accelerator	Raspberry Pi
Power Consumption	5–10W	2W	3–7W
AI Performance	0.5–1.3 TOP	4 TOPS	-
Cost	99–150 USD	60-100 USD	
Deployment	Complex	Simple	

has an advantage in specialized tasks such as real-time image recognition, thanks to its high efficiency with TensorFlow Lite. The Coral Accelerator can achieve 4 TOPS, specializing in image recognition and object detection with low latency. Jetson Nano's GPU supports multiple AI models, but its real-world performance is only 0.5–1.3 TOPS when running on the Maxwell GPU. In early fire detection, where detecting smoke and fire from camera images is crucial, the Coral Edge TPU is better optimized for TensorFlow Lite, enabling faster recognition and immediate alerts, reducing the risk of delays. In terms of cost, the Coral Accelerator and Raspberry Pi setup costs approximately $60–$100 (including Raspberry Pi 4 and Coral USB Accelerator), while Jetson Nano costs $99–$150, excluding a higher-power adapter and additional accessories. Using Coral combined with Raspberry Pi helps reduce deployment costs for large-scale systems, especially when installing dozens or hundreds of monitoring points. In terms of deployment, the Coral Edge TPU can connect to the Raspberry Pi via USB or PCIe, making it easy to integrate into existing systems. In contrast, Jetson Nano requires a more complex setup and a stronger power supply, which can be challenging for IoT deployments in remote areas. Additionally, temperature, humidity, and gas sensor modules on Raspberry Pi can be easily integrated with Coral, providing multi-source data to improve fire detection accuracy.

Thus, when comparing Coral Accelerator combined with Raspberry Pi and Jetson Nano for early fire detection applications, Coral Accelerator offers several key advantages: Lower power consumption, making it ideal for continuous IoT operation; Specialized AI acceleration, enabling faster smoke/fire recognition with TensorFlow Lite; Lower cost, facilitating large-scale deployments with an optimized budget; Greater flexibility for IoT sensor integration, allowing easy scalability and upgrades. Therefore, if an AI-embedded fire detection system is required, the Coral Edge TPU combined with Raspberry Pi is a better solution in terms of performance, power efficiency, and cost-effectiveness compared to Jetson Nano.

We used several other tools to evaluate the performance of combining the Coral Accelerator with the Raspberry Pi. Specifically, we (1) use the top/htop tools, recording every 5 seconds, to measure the average CPU utilization during continuous inference of 60 frames (on CPU only) and inference of 60 frames (using Coral); (2) measure the memory used by the Python inference process at steady state after running for 1 minute; (3) utilize edgetpu_monitor –graph=false –interval=1 to record readings every second. After 5 minutes, we compute the average and deviation to assess the Coral temperature; (4) used vcgencmd measure_temp, recording every 10 seconds during inference, to determine the Pi SoC Temperature; and utilize a watt-meter connected to the power supply for the Raspberry Pi and Coral, taking 5 consecutive measurements and averaging them to calculate the power consumption.

The measured results are shown in Table 13. When running YOLOv10-S solely on the Raspberry Pi 4's CPU, the average CPU usage reached 85 ± 5% and the RAM usage was about

2200 ± 50MB, causing the SoC temperature to rise to 65 ± 4°C, which could lead to throttling if run for extended periods. The overall power consumption of the system was approximately 7.8 ± 0.2W. In contrast, when using the combination with the Coral USB Accelerator, the average CPU usage was only 45 ± 5%, the RAM usage was about 1900 ± 60MB, the Coral Accelerator temperature was 58 ± 3°C, and the Raspberry Pi SoC temperature was 57 ± 2°C, with the overall system power consumption measured at approximately 9.1 ± 0.3W.

5.4 Comparison with Models Optimized for Edge Devices

To evaluate the advantages of YOLOv10-S when deployed on the Coral Accelerator, we collected reference metrics from recent studies on models optimized for edge devices, including PG-YOLO [4], EFA-YOLO [15], and YOLOv8n (implemented on Edge TPU). The benchmark data includes mAP@50–95, latency on Edge TPU, memory usage, and false positive rate (FPS). From Table 14, it can be observed that YOLOv10-S achieves a comparable mAP@50–95 to YOLOv8n (46.8% vs. 47.0%) but has a larger number of parameters (7.2 M vs. 4.2 M), resulting in a slightly higher inference time (1,698 ms vs. 1,650 ms). However, YOLOv10-S has the lowest false positive rate (4.7%), attributed to a diverse dataset and thorough fine-tuning. Although PG-YOLO and EFA-YOLO have smaller model sizes, they exhibit longer inference times and 2.3–5.6% lower mAP@50–95 compared to YOLOv10-S. These results indicate that YOLOv10-S offers a well-balanced trade-off between accuracy and latency when deployed on the Coral Accelerator.

Table 13. Resource utilization for YOLOv10-S on Raspberry Pi 4 and Coral USB Accelerator.

Parameter	Unit	CPU alone	With Coral TPU
CPU utilization (avg)	%CPU	85 ± 5%	45 ± 4%
RAM usage (avg)	MB	2,200 ± 50	1,900 ± 60
Coral Accelerator Temperature	°C	N/A	58 ± 3°C
Raspberry Pi SoC Temperature	°C	65 ± 4°C	57 ± 2°C
Power consumption (whole system)	W	7.8 ± 0.2	9.1 ± 0.3

Table 14. Comparison of YOLOv10-S with other edge-optimized models.

Model	Params (M)	mAP50-95 (%)	Inference Time (ms)	Memory (MB)	False Positive Rate (%)
PG-YOLO [4]	3.6	41.2	1,950	10.5	6.3
EFA-YOLO [15]	1.8	44.5	1,720	9.8	5.8
YOLOv8n (TPU)	4.2	47.0	1,650	9.5	5.2
YOLOv10-S	7.2	46.8	1,698	9.28	4.7

6. Conclusion

This study introduces a lightweight and efficient framework for real-time fire detection, integrating the fine-tuned YOLOv10 small model with the accelerator module, specifically optimized for IoT devices. The system not only addresses critical challenges such as latency, accuracy, and deployment in resource-constrained environments but also ensures reliable operation even under limited network connectivity. Its compact design, edge processing capabilities, and reduction in false alarms enhance the system's practicality for fire monitoring in residential, industrial, and natural environments. Experimental results demonstrate that deploying YOLOv10-S with the Coral Accelerator significantly outperforms CPU-based solutions, achieving inference speed four times faster and reducing power consumption by approximately 30%. These improvements not only affirm the feasibility of the proposed framework but also make it an ideal choice for energy-constrained geographic regions. The full source code including setup instructions for Raspberry Pi and Coral Accelerator is publicly available at [22].

This research lays a strong foundation for low-cost real-time fire detection systems on IoT platforms, promising to enhance early detection capabilities, reduce response times, and mitigate damages. These advancements contribute to a safer environment through intelligent edge-based monitoring solutions. However, some limitations remain to be addressed. Detection under challenging conditions, such as dense smoke or low light, could be improved by integrating multimodal data sources, such as thermal imaging or gas sensors. Future efforts will focus on enhancing adaptability, scalability, and system performance to ensure effectiveness across a wide range of challenging scenarios. Additionally, upcoming research will explore the application of advanced network compression and structured pruning techniques, and more importantly, the use of quantization-aware pruning on YOLOv10-S to reduce the model size to below 3MB, lower the inference time to under 1 s/frame, while still maintaining mAP@50 more than 80%. We also aim to combine knowledge distillation from YOLOv10-M with dynamic channel pruning to preserve high accuracy on ultra-low-power edge devices.

Declaration of Competing Interests

The authors declare no conflict of interest.

Funding

This research is supported by Posts and Telecommunications Institute of Technology (PTIT).

Data availability

Data used in this article is openly available at: https://universe.roboflow.com/binbin-iz5rn/fire_detection-ckgf5/dataset/4

References

[1] S. Hall, "Fire Loss in the United States During 2023", NFPA Research, 2024. https://www.nfpa.org/education-and-research/research/nfpa-research/fire-statistical-reports/fire-loss-in-the-united-states Accessed on Jan. 2, 2025.

[2] State and Agency, "2018 National Year-to-Date Report on Fires and Acres Burned", National Interagency Fire Center, 2024. Accessed on Jan. 2, 2025.

[3] W. F. Magalhães *et al.*, "Investigating Mobile Edge-Cloud Trade-Offs of Object Detection with YOLO", in *Symposium on Knowledge Discovery, Mining And Learning (KDMILE), Porto Alegre, Brasil*, 2019, pp. 49–56. https://doi.org/10.5753/kdmile.2019.8788

[4] C. Dong *et al.*, "PG-YOLO: A Novel Lightweight Object Detection Method for Edge Devices in Industrial Internet of Things", *IEEE Access*, vol. 10, pp. 123736–123745, 2022. https://doi.org/10.1109/ACCESS.2022.3223997

[5] M. A. Saleem, "An Edge Computing Infrastructure and Deep Learning Methods for Wildfire Monitoring and Anomalies Detection", Ph.D. dissertation, Ministry of Higher Education, 2022.

[6] F. M. Talaat and H. ZainEldin, "An Improved Fire Detection Approach Based on YOLO-v8 for Smart Cities", *Neural Comput. & Applic.*, vol. 35, pp. 20939–20954, 2023. https://doi.org/10.1007/s00521-023-08809-1

[7] S. Saponara *et al.*, "Enabling YOLOv2 Models to Monitor Fire and Smoke Detection Remotely in Smart Infrastructures", *Lecture Notes in Electrical Engineering*, vol. 738, Springer, Cham, 2021. https://doi.org/10.1007/978-3-030-66729-0_4

[8] G. Chenglin *et al.*, "Real-time Fire Detection and Alarm System Using Edge Computing and Cloud IoT Platform", *Int. J. Wireless Mobile Comput.*, vol. 22, no. 3/4, pp. 310–318, 2022. https://doi.org/10.1504/IJWMC.2022.124822

[9] A. A. Briley and F. Afghah, "Hardware Acceleration for Real-Time Wildfire Detection Onboard Drone Networks", *IEEE INFOCOM 2024 – IEEE Conf. on Computer Communications Workshops (INFOCOM WKSHPS), Vancouver, Canada*, 2024, pp. 01–06. https://doi.org/10.1109/INFOCOMWKSHPS61880.2024.10620877

[10] I. Shamta and B. E. Demir, "Development of a Deep Learning-based Surveillance System for Forest Fire Detection and Monitoring Using UAV", *PLoS ONE*, vol. 19, no. 3, e0299058, 2024. https://doi.org/10.1371/journal.pone.0299058

[11] G. Jocher *et al.*, "YOLOv10 – Ultralytics YOLO Documents", *Ultralytics*, 2024. [Online] Available: https://docs.ultralytics.com/vi/models/yolov10/

[12] Google Research, "Get Started with the USB Accelerator", Edge TPU Documentation. [Online] Available: https://coral.ai/products/accelerator-module Accessed on Jan. 30, 2025.

[13] D. Mamadaliev *et al.*, "ESFD-YOLOv8n: Early Smoke and Fire Detection Method Based on an Improved YOLOv8n Model", *Fire*, vol. 7, no. 9, p. 303, 2024. https://doi.org/10.3390/fire7090303

[14] S. Liang and X. Zhang, "An Object Detection System for Automatic Driving: MEC-YOLO Based on Cloud-Edge-End", in *Proceedings of the 6th Int. Conf. on Natural Language Processing (ICNLP), Xi'an, China*, 2024, pp. 536–541. https://doi.org/10.1109/ICNLP60986.2024.10692461

[15] P. Weichao *et al.*, "EFA-YOLO: An Efficient Feature Attention Model for Fire and Flame Detection", *Computer Vision and Pattern Recognition*, 2024. https://doi.org/10.48550/arXiv.2409.12635

[16] M. A. Firdaus, "Prototype Smart Integrated Fire Detection Based on Deep Learning YOLO V8 and IoT (Internet of Things) to Improve Early Fire Detection", *Int. J. Appl. Math., Sci. & Technol. for Natl. Defense*, vol. 2, no. 2, 2024. https://doi.org/10.58524/app.sci.def.v2i2.437

[17] S. Banerjee *et al.*, "Real-Time Fire Detection in Unmanned Ground Vehicles Integrating YoloV5 and AWS IoT", in *Proceedings of the Int. Conf. on System, Computation, Automation and Networking (ICSCAN), Puducherry, India*, 2023, pp. 1–6. https://doi.org/10.1109/ICSCAN58655.2023.10394971

[18] J. Gallagher, "How to Augment Images for Object Detection", Roboflow, 2024.[Online] Available: https://blog.roboflow.com/object-detection-augmentation/

[19] C. Constantin, "FireSmokeDataset Dataset", *Roboflow Universe*, 2024. https://universe.roboflow.com/catargiuconstantin/firesmokedataset/dataset/3 Accessed on Jan. 6, 2025.

[20] BINBIN, "FIRE_DETECTION Dataset", *Roboflow Universe*, Jan. 2025. https://universe.roboflow.com/binbin-iz5rn/fire_detection-ckgf5/dataset/4 Accessed on Jan. 6, 2025.

[21] T. T. Huynh *et al.*, "Enhancing Fire Detection Performance Based on Fine-Tuned YOLOv10", *Comput. Mater. Contin.*, vol. 81, no. 2, pp. 2281–2298, 2024. https://doi.org/10.32604/cmc.2024.057954

[22] BIN-PDT, "AN AI & IOT FIRE DETECTION FRAMEWORK", GitHub, 2025. [Online] Available: https://github.com/BIN-PDT/AIOT_FIRE_DETECTION_FRAMEWORK

Received: February 2025
Revised: June 2025
Accepted: June 2025

Contact addresses:
Trong Thua Huynh
Posts and Telecommunications Institute of Technology
Ho Chi Minh City
Vietnam
e-mail: thuaht@ptit.edu.vn

De Thu Huynh
The Saigon International University
Ho Chi Minh City
Vietnam
e-mail: huynhdethu@siu.edu.vn

Du Thang Phu
Posts and Telecommunications Institute of Technology
Ho Chi Minh City
Vietnam
e-mail: thangpd@ptithcm.edu.vn

Anh Hao Nguyen
Posts and Telecommunications Institute of Technology
Ho Chi Minh City
Vietnam
e-mail: haona@ptit.edu.vn

TRONG THUA HUYNH is currently the Head of the Information Security Department, Faculty of Information Technology II, at the Posts and Telecommunications Institute of Technology (PTIT), Vietnam. He received a bachelor's degree in information technology from Ho Chi Minh City University of Natural Sciences, a master's degree in computer engineering from Kyung Hee University, Korea, and a PhD degree in computer science from the Ho Chi Minh City University of Technology, Vietnam National University at Ho Chi Minh City in 2018. His key areas of research include cybersecurity, AI and big data, and intelligent information systems.

DE THU HUYNH is currently the Head of the Software Technology Department, School of Computer Science & Engineering, at the Saigon International University (SIU), Ho Chi Minh City, Vietnam. He received a PhD degree in computer science from Huazhong University of Science and Technology, China in 2018. His research interests focus on edge AI for 6G networks, autonomous and cooperative UAV systems, energy efficiency in wireless networks, and intelligent information systems.

DU THANG PHU is currently a researcher at the Information Security Laboratory, Faculty of Information Technology II, at the Posts and Telecommunications Institute of Technology (PTIT), Vietnam. He received his engineer's degree in information technology from PTIT in 2025. His current research interests focus on machine learning and deep learning, with a particular emphasis on computer vision.

ANH HAO NGUYEN is a lecturer at the Faculty of Information Technology II, Posts and Telecommunications Institute of Technology (PTIT), Vietnam. He holds a bachelor of science in mathematics – informatics from Ho Chi Minh City University of Natural Sciences and a master's degree in information management from the Asian Institute of Technology (AIT), Thailand, from 2003. His key research areas are AI planning and automation.