UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION AND ELECTRICAL ENGINEERING AND APPLIED MATHEMATICS

Report - Machine Learning

ONFIRE

Team 36

Group members

Name and surname	**ID**	**E-mail**
Adinolfi Teodoro	0622701902	t.adinolfi2@studenti.unisa.it
Amato Emilio	0622701903	e.amato16@studenti.unisa.it
Bove Antonio	0622701898	a.bove57@studenti.unisa.it

ACADEMIC YEAR 2022/2023

INDICE

CAPITOLO 1

INTRODUCTION

Computer vision is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos and other visual inputs. Within this domain, a particularly intriguing application arises in the realm of territorial video surveillance. Based on the WWF information the frequency, extent and intensity of fires have increased enormously over the last century: the fire season is becoming more extreme and longer, by 15% over the last 50 years, fuelled by long periods of extreme heat and little rain. For this reason, the **analysis and detection of fires using intelligent algorithms** capable of running on small embedded systems within security cameras is an interesting challenge.

1.1 An hard goal

Because of the particular characteristics of a fire, it is not easy to detect at times even for a human being, we can in fact distinguish two types of situations that we can categorise as 'fire':

Figura 1.1: Fire Examples

In these two situations, the information we obtain by analysing the image is very different, in one case the fire is not visible, however the shape of the smoke and its contextualised location makes us think of a fire. In the second case, however, the presence of a fire is clearly rendered by the presence of flames.
If we focus only on the first photo then, the pitfalls are numerous: the smoke could easily be confused with the clouds (and this often happens even to a human observer), moreover, there is not only one type of smoke but there are more, it can be "greyish" and not very dense, due to the burning of shrubs or black and dense due to the burning of plastics. If we consider the second case, even fire can be challenging to interpret accurately. When capturing flames through a video camera, the visual effect can closely resemble situations where the camera lens is exposed to intense light sources such as sunlight, a car's headlight, or a flashlight. These various potential distractions make it considerably difficult to differentiate between genuine fire and false information. The problem therefore has many facets that make the challenge very complex.

1.1.1 What we need

Based on the statements made in the previous paragraphs, we are seeking a system capable of detecting a fire, regardless of whether visible flames or only distant smoke are present. The system should be lightweight and suitable for deployment on low-power computing systems, specifically on security cameras. Furthermore, it should have a very low rate of both false positives and false negatives. A reliable security system should always identify a fire and do so only when it is actually present. Based on the aforementioned points, in the upcoming chapters, we will analyze some of the potential solutions identified and provide detailed insights into the one that has been implemented. We will emphasize the entire process, including algorithm design, dataset creation, and conducted testing.

CAPITOLO 2

PROPOSED APPROACHES

In this chapter, we aim to analyze various approaches that have been identified for addressing the problem at hand. We will carefully evaluate each approach, emphasizing their respective advantages and disadvantages, with a focus on the selected model and the rationale behind its choice. By examining different approaches, we can gain a comprehensive understanding of the diverse strategies that have been employed in the field. This comparative analysis allows us to highlight the strengths and weaknesses of each approach, enabling us to make an informed decision on the most appropriate solution for our specific problem. Factors such as model accuracy and computational efficiency are pivotal in determining the most appropriate choice.

2.1 Convolutional Neural Network

The Convolutional Neural Network (CNN) is a subtype of Neural Networks that is mainly used for applications in image recognition.
Given the nature of these networks, it seemed immediately reasonable to approach the problem via a CNN-based classifier. The main pre-trained CNN architectures on large datasets have been taken into consideration. In particular, the following networks, which have been trained on the provided dataset, were considered:

- EfficientNet

- ShuffleNetV2
- ResNet-18
- MobileNetV3

During the fine-tuning process of multiple networks, we encountered similar issues across all models. The major challenge was the excessive number of false positives, where clouds were frequently misclassified as smoke, and various light sources were erroneously identified as fire. These difficulties led us to conclude that the learning task was particularly demanding, especially considering that we had grouped two distinct scenarios, as depicted in Figure 1.1, under a single "fire" class. In an attempt to address these challenges, we modified the networks and shifted our approach from a binary classification problem (fire versus non-fire) to a multi-class classifier with three classes: fire, smoke, and fire with smoke. Unfortunately, this revised approach still encountered the same issues, and the observed loss function values remained higher than expected. Upon careful analysis, we determined that the problem stemmed from the inclusion of a "fire and smoke" class, which inadvertently introduced bias by suggesting that the network's predictions fell outside the fire or smoke categories. In light of these challenges and the limitations encountered, we decided to explore an alternative direction by combining the strengths of Convolutional Neural Networks (CNNs) in visual element recognition with the capabilities of Recurrent Neural Networks (RNNs) in capturing dynamic information from videos, such as the movement of smoke or fire.

2.1.1 Training

Dataset

In this first approach, we used the dataset provided for the competition. All the videos were sampled at a rate of 1 frame per second. This sampling strategy allowed us to capture the dynamic nature of fire or smoke without relying on highly correlated frames. By selecting frames at regular intervals, we aimed to capture the overall movement and changes in the fire or smoke patterns over time. This approach allowed us to maintain a balance between capturing important temporal information and avoiding excessive redundancy in the dataset.

The dataset was divided into training and validation sets using the classic strategy of an 80% training set and a 20% validation set. Although the frames belong to the same video, considering the sampling rate, each frame captures different characteristics and variations in the fire or smoke patterns. This reduced the problems caused by correlation between frames, as the sampled frames provided diverse and independent information for training and evaluating the model. By including frames from different time points in the same video, we ensured that the model could learn and generalize well across various instances of fire and smoke, enhancing its overall performance. In the augmentation phase of our work, we utilized Albumentations, which helped us increase the number of frames we could work with. Augmentation is an important aspect of training as it is a technique used to enhance the model's generalization capability. We applied the following operations:

Preprocessing

- **Resize**: all the pretrained models used for fine-tuning required the input image to be resized to a specific dimension. Resizing the images ensured that they were compatible with the input requirements of the pretrained models
- **Normalization**: the different pretrained networks used for fine-tuning had their fixed preprocessing requirements, which we applied to normalize the input images accordingly

Augmentation

- **Horizontal Flip**: this transformation helps the model to generalize the concept of fire by considering different orientations. By applying horizontal flips to the images, the model becomes more robust to variations in the geometry or direction of the fire, allowing it to learn features that are not solely dependent on the specific orientation of the fire in the images
- **Hue**: the hue augmentation alters the hue component of the image, introducing variations in color. By applying this transformation, we aimed to make the model more robust to changes in the color representation of fire or smoke in different lighting conditions or environments. This helps the model generalize well to various color variations of fire or smoke
- **Saturation**: the saturation augmentation adjusts the saturation level of the image, creating variations in color intensity. By applying this transformation, we aimed to

enhance the model's ability to handle different levels of color saturation in fire or smoke. This augmentation helps the model to generalize across images with different color intensities of fire or smoke

By applying these augmentations, we aimed to increase the diversity of the training data and expose the model to a wider range of variations in fire and smoke patterns. This enables the model to learn robust and generalized features, improving its performance on unseen data.

Hyper-Parameters settings

For the training of these networks we used different optimizator ad hyperparameters settings, a small summary is here.

Optimizer	Learning Rate	Momentum	Wheight Decay
SGD	0.0001	0.9	n.a
SGD	0.001	0.9	n.a
ADAM	0.00001	0.9	10^{-4}
ADAM	0.0001	0.9	10^{-5}
ADAM	0.001	0.9	10^{-4}

Tabella 2.1: Differents training settings

We tried all this differents combinations trying to reduce the loss and have huge improvements on the accuracy.

2.1.2 Detection Algorithm

The CNN-based approach utilized a single frame to make predictions and determine the correct class. However, to reduce the number of false positives, we implemented an algorithm that incorporated temporal information. This algorithm introduced a parameter called `fire_threshold`, which specifies the number of frames that need to consistently indicate the presence of fire for an alarm to be triggered. The algorithm operates as follows:

1. the CNN analyzes each frame independently and predicts the class (fire or non-fire) for that frame

2. the algorithm keeps track of the number of consecutive frames where fire is predicted
3. if the number of consecutive frames with fire predictions reaches or exceeds the `fire_threshold`, the alarm is triggered and fire is considered detected
4. if the number of consecutive frames with fire predictions drops below the `fire_threshold`, the counter returns to 0 and the prediction is considered incorrect

By incorporating temporal consistency in the detection process, this algorithm helps reduce false positives by ensuring that fire predictions are consistent over multiple frames before raising an alarm. This approach adds an additional level of confidence to the detection system and helps filter out temporary or spurious fire predictions that may occur in isolated frames.

2.2 Convolutional Neural Network and Recurrent Neural Networks

This new approach aims to leverage the CNN's ability to extract features and recognize visual patterns from individual frames, while the RNN's sequential modeling capabilities can capture temporal dependencies and detect dynamic changes over time.

This shift in our approach marks a promising endeavor in addressing the challenges faced in the previous models. By exploiting both spatial and temporal information, we aim to enhance the model's ability to distinguish between genuine fire instances and false positives caused by clouds or light sources. Through the integration of CNN and RNN components, we hoped to achieve a more robust and accurate fire detection system.

Regrettably, despite the modifications made and the adoption of a CNN-RNN hybrid model, the results did not meet our expectations. The problem of false positives persisted, and the precision of the fire detection was not satisfactory in many cases.

Upon careful analysis, we identified several factors that may have contributed to these continued challenges. Firstly, the inherent complexity and variability of fire and smoke patterns in videos pose inherent difficulties for accurate detection. The dynamic nature of fire and smoke, combined with variations in lighting conditions, camera angles, and environmental factors, make it challenging for the model to differentiate between genuine

fire instances and similar visual patterns.

Secondly, the limited availability of high-quality labeled training data specifically tailored to the problem of fire detection in videos may have hindered the model's ability to learn discriminative features effectively. Insufficient training data can lead to difficulties in capturing the diverse range of fire and smoke characteristics, resulting in decreased detection performance.

After careful consideration of the observations and challenges encountered, we began to explore a different perspective for solving the problem of fire detection. Usually, in classification datasets, there is a huge amount of samples that have one clear subject in the photo, such as problems like "cats vs dogs" or "MNIST," but also more difficult datasets designed for the recognition of a wide range of objects like COCO. In our case, smoke is usually difficult to see and is never clearly highlighted in the image. We hypothesized that explicitly highlighting the presence of fires in both smoke and flame situations could alleviate the confusion caused by non-fire elements present in the environment, ultimately improving the performance of the detection system. This led us to the use of an object detection approach, which will be discussed in detail below.

2.2.1 Network High level Architecture

We will now introduce with an high level of detail the architecture of the realized network:

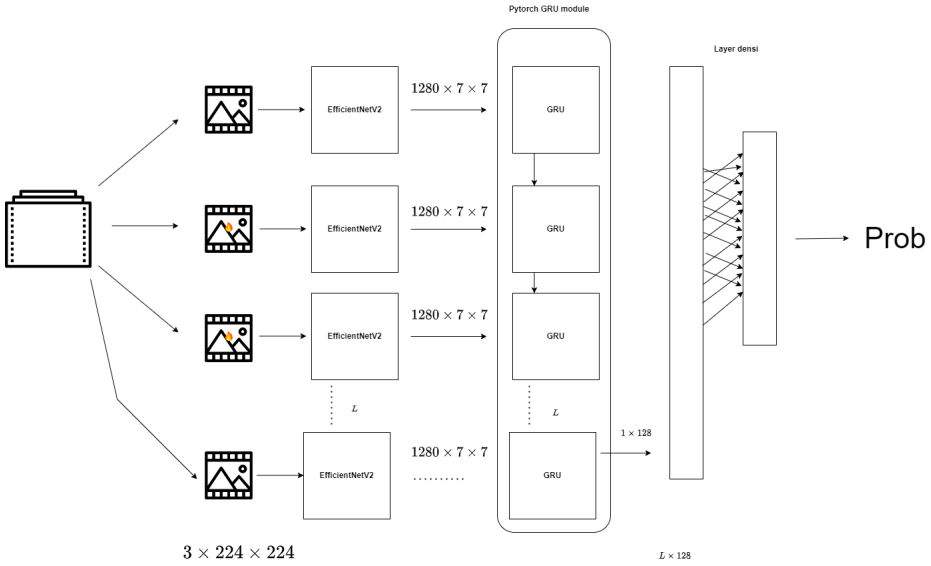

Figura 2.1: High-level architecture

The image shows the architecture of the proposed approach, with the CNN part specialized for "EfficientNetV2" and an RNN layer consisting of a Gated Recurrent Unit (GRU). The setup for the RNN is a sequence-to-single value transformation, where the layer takes a sequence of L frames as input and returns a single vector of 128 features. At the end of the network, there are two fully connected layers serving as the head of the network. The CNN part, specialized for EfficientNetV2, is responsible for extracting visual features from individual frames. These features capture important spatial information related to fire and smoke. The RNN layer, using the GRU architecture, takes a sequence of frames as input and processes them sequentially, capturing temporal dependencies and changes over time. By compressing the sequence of frames into a single vector of features, the RNN layer summarizes the temporal information and provides a high-level representation of the fire and smoke dynamics within the sequence. The two fully connected layers at the end of the network serve as the classification head, taking the output from the RNN layer and mapping it to the desired output classes (e.g., fire, smoke, non-fire). These layers enable the network to make predictions based on the combined spatial and temporal information extracted throughout the architecture. In addition to using the Gated Recurrent Unit

(GRU), we also experimented with the Long Short-Term Memory (LSTM) architecture. The LSTM is another type of recurrent neural network that is well-suited for modeling long-term dependencies in sequential data, such as videos.

2.2.2 Training and Dataset

Dataset For this new approach, the structure of the dataset and the preprocessing/augmentation used is the same as described in the **CNN** chapter.

Training During the training phase, we experimented with various configurations as discussed in the previous chapter. Both the CNN and RNN layers were trained together, with the CNN layers being pre-trained and the RNN layers being newly added and requiring training. We explored different training strategies, including adjusting the learning rate, using different optimization algorithms (e.g., SGD, Adam), and varying the batch size and number of training epochs. We carefully monitored the training process, observing the loss function and evaluating the model's performance on the validation set. The training process aimed to optimize the model's ability to capture both spatial features (using the CNN) and temporal dependencies (using the RNN) in order to accurately detect fire and smoke. By training the model on a combination of visual and sequential information, we expected it to improve its understanding of the dynamic nature of fire and smoke in videos.

2.2.3 Detection Algorithm

In this approach, the algorithm used is similar to the one employed with the CNN. However, in this case, we accumulate a batch of L frames and provide this batch to the network, which utilizes this information to predict a class. If a fire is detected, we wait for a consecutive number of detections (specified by the `fire_threshold` parameter) before triggering an alarm.

The rationale behind waiting for a longer period of time compared to the CNN-only approach is to capture the dynamic aspect of fire and smoke. By accumulating multiple frames and considering them as a batch, the network can analyze the temporal progression of fire and smoke patterns. This investment of time allows for a more accurate prediction, as the network can better understand the dynamic behavior of fire and smoke over time.

By waiting for a sufficient number of consecutive fire detections, we aim to ensure that the fire signal is consistent and not a temporary anomaly or false positive. This approach adds an additional level of confidence in the detection system, reducing the likelihood of false alarms.

The combination of spatial information from the CNN and temporal information from the batched frames enables the network to make more informed and accurate predictions about the presence of fire. By leveraging both static and dynamic cues, this approach enhances the model's capability to distinguish between real fire instances and other visual patterns that may resemble fire.

2.2.4 The problems

We had hoped that the added modifications to the CNN in this new approach would yield generally better results. However, after conducting a series of tests on previously unseen videos obtained from the internet, we noticed that the presence of smoke-like patterns or objects with similar color and/or characteristics to fire easily deceived the network, leading to a significant increase in false positives. Additionally, the accuracy of fire detection within the videos was unsatisfactory. Based on these observations, we decided to modify our approach.

By employing a network like YOLO, which is designed for object detection, the information obtained from the network (a bounding box containing fire or smoke) can be further processed using heuristics based on human considerations. This eliminates the reliance on an RNN to learn dynamic features and may give us better results.

2.3 Object detection: YOLO

2.3.1 Why object detection?

As analyzed so far, either a convolutional network alone for fire detection or in combination with a recurrent network could face some challenges that can lead to problems including especially false positives. Here are some reasons:

- **Enviromental context**: fire detection requires understanding the environmental context in which the scene is located. A simple convolutional network may not be able to effectively capture the spatial and contextual information needed to correctly distinguish a fire from other similar objects or phenomena. For example, it may mistakenly detect objects such as streetlights or light reflections on metal surfaces as fires with higher probability

- **Size and shape of fires**: fires can vary greatly in size and shape. A basic convolutional network can be limited in its detection potential because of the size of the filters and feature maps used. It may not be able to capture important details or model the variety of shapes a fire can take. This can lead to false positives or false negatives, dependening on the specific characteristic of the model

- **Reducing false positives with object detection models**: object detection models integrate both object detection and classification into a single framework. These models can detect object in the scene and classify them simultaneously. This can help reduce the false positives associated with simple classification. For example, an object detection model can correctly detect a fire as an object, but later can use contexual information to determine whether the dected object is actually a fire or something else

- **Use of pyramidal features**: object detection mode often use pyramidal features, which are feature maps at different scales. This allows them to capture both fine details and context information on different dimensions of objects. In the case of fire detection, the ability to work on different scales can be advantageous for capturing fires of different sizes present in the scene

In summary, a simple convolutional network could address challenges such as understanding the environmental context, variation in fire sizes and shapes, and the presence of false positives. The use of object detection model with pyramidal features can help overcome these challenges by providing better performance for this kind of problems, which is why YOLO network was chosen.

2.3.2 YOLO network

Assuming that a network has enough information to determine whether an object is present in an image, it seems plausible to believe that the network also has information to find the location of the object in the image. This is the idea behind the you only look once (YOLO) algorithms. This joint approach helped YOLO become the state-of-the-art method for fast object identification and classification and is the reason why we use it in our method.

YOLO uses a single deep convolutional neural network to simultaneosuly predict class probabilities and bounding box positions directly from images in a single pass. Its original version was proposed by Joseph Redmon in 2016 and later improved in other versions. After Redmon announced he would stop his computer vision research due to the massive use of his algorithms for military purposes, Alexey Bochkovskiy continued the development of YOLO series and its framework by publishing YOLOv4: these framework usually released two version of the network, a tiny architectures designed for resource-constrained device and a large architecture designed for Graphics Processing Units (GPUs). YOLOv5 was released shortly after YOLOv4, proposed independently by Jocher in the PyTorch framework.

High-level architecture for single-stage object detector

There are two types of model for object detection: two-phase object detectors and single-phase object detectors. The architecture of single-phase object detector (such as YOLO) consists of three components: **Backbone**, **Neck** and **Head** to make dense predictions, as shown in the figure below.

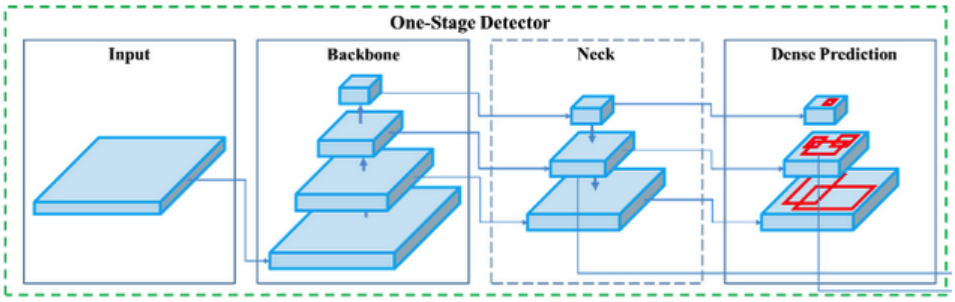

Figura 2.2: High-level architecture

Backbone is a pretrained network used to extract rich feature representation for images, helping to reduce the spatial resolution of the image and increasing its feature (channel) resolution.

Neck is used to extract feature pyramids, helping the model to generalize well to objects on different sizes and scales.

Head is used to perform the final stage operations, applying anchor boxes on feature maps and render the final output: classes , objectness scores and bounding boxes.

YOLOv5 architecture

YOLOv5 was released with five different sizes:

- **n** for extra small (nano) size model
- **s** for small size model
- **m** for medium size model
- **l** for large size model
- **x** for extra large size model

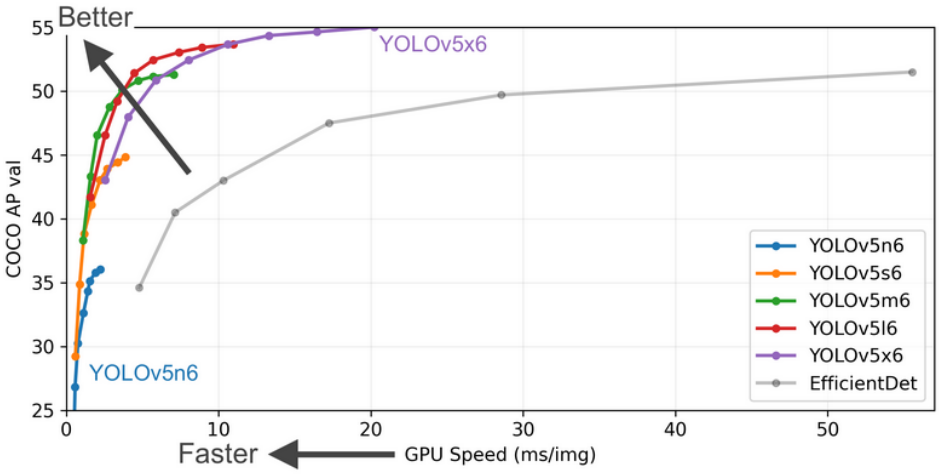

Figura 2.3: Performance of YOLOv5 different size models

There is no difference between the five models in terms of operations used except the number of layers and parameters as shown in the table below.

Model	size (pixels)	mAP^{val} 0.5:0.95	mAP^{val} 0.5	Speed CPU b1 (ms)	Speed V100 b1 (ms)	Speed V100 b32 (ms)	params (M)	FLOPs @640 (B)
YOLOv5n	640	28.0	45.7	**45**	**6.3**	**0.6**	**1.9**	**4.5**
YOLOv5s	640	37.4	56.8	98	6.4	0.9	7.2	16.5
YOLOv5m	640	45.4	64.1	224	8.2	1.7	21.2	49.0
YOLOv5l	640	49.0	67.3	430	10.1	2.7	46.5	109.1
YOLOv5x	640	50.7	68.9	766	12.1	4.8	86.7	205.7
YOLOv5n6	1280	36.0	54.4	153	8.1	2.1	3.2	4.6
YOLOv5s6	1280	44.8	63.7	385	8.2	3.6	12.6	16.8
YOLOv5m6	1280	51.3	69.3	887	11.1	6.8	35.7	50.0
YOLOv5l6	1280	53.7	71.3	1784	15.8	10.5	76.8	111.4
YOLOv5x6 + TTA	1280 1536	55.0 **55.8**	72.7 **72.7**	3136 -	26.2 -	19.4 -	140.7 -	209.8 -

Figura 2.4: Details on different size models

All the YOLOv5 models are composed of the same 3 components: **CPS-Darknet53** as a backbone, **SPP** and **PANet** in the model neck and the **head** used in YOLOv4. The next figure shows the architecture of the network, based on the statements just made.

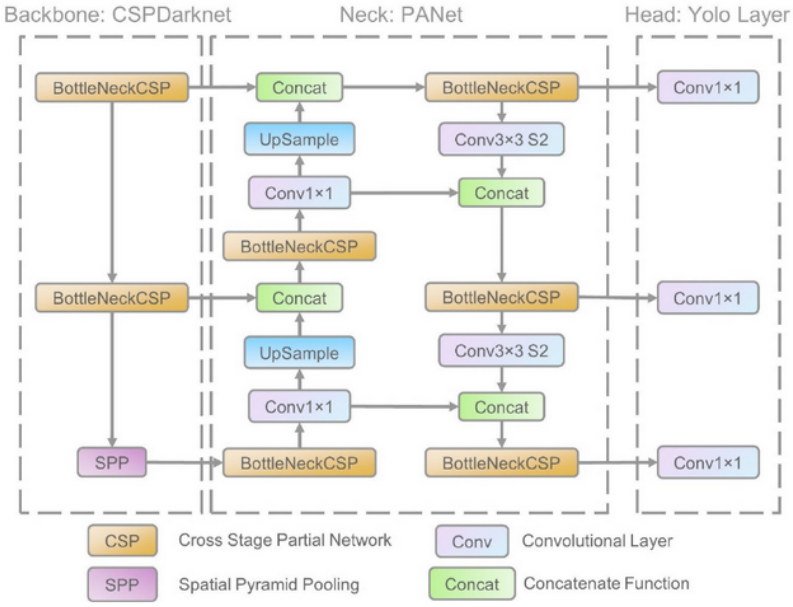

Figura 2.5: Network architecture for YOLOv5

CSP-Darknet53

CSP-Darknet53 is just the convolutional network **Darknet53** used as the backbone for YOLOv3 to which the authors applied the **Cross Stage Partial** (CSP) network strategy. YOLOv5 employs CSPNet strategy, that helps to overcome the **vanishing gradient problem** and **redundant gradient** by truncating the gradient flow, because of its depth. Applying this strategy comes with big advantages to YOLOv5, since it helps reducing the number of parameters and helps reducing an important amount of computation (less FLOPs) which lead to **increasing the inference speed** that is a curcial parameter in real-time object detection models.

Neck of YOLOv5

YOLOv5 brought two major changes to the model neck. First a variant of **Spatial Pyramid Pooling** (SPP) has been used, and the **Path Aggregation Network** (PANet) has been modified by incorporating the **BootleNeckCSP** in its architecture

Path Aggreation Network (PANet) is a feature pyramid network, that has been used in the previous version of YOLO (4th version) to improve information flow and to help in the proper localization of pixels in the task of mask predicition, which consists of

generating pixel-per-pixel masks that identify the exact shape and location of detected objects, allowing for more precise segmentation of objects within the image. In YOLOv5 this network has been modified by applying the CSPNet strategy to it.

Spatial Pyramid Pooling (SPP) as a block performs an aggregation of the information that receives from the inputs and returns a fixed length output. Thus it has the advantage of significantly increasing the receptive field and segregating the most relevant context features without lowering the speed of the network. This block has been used in previous version of YOLO (3 and 4) to separate the most important features from the backbone, however in YOLOv5 **SPPF**, which is just another variant of the SPP block, has been used to **improve the speed of the network**.

For the head of the network YOLOv5 uses the same head as YOLOv3 and YOLOv4. It is composed from three convolutional layers that predicts the location of the bounding boxes (x,y,height,width), the scores and the object classes.

Activation Function

Choosing an activation function is crucial for any deep learning model, for YOLOv5 the authors went with SiLU and Sigmoid activation function.

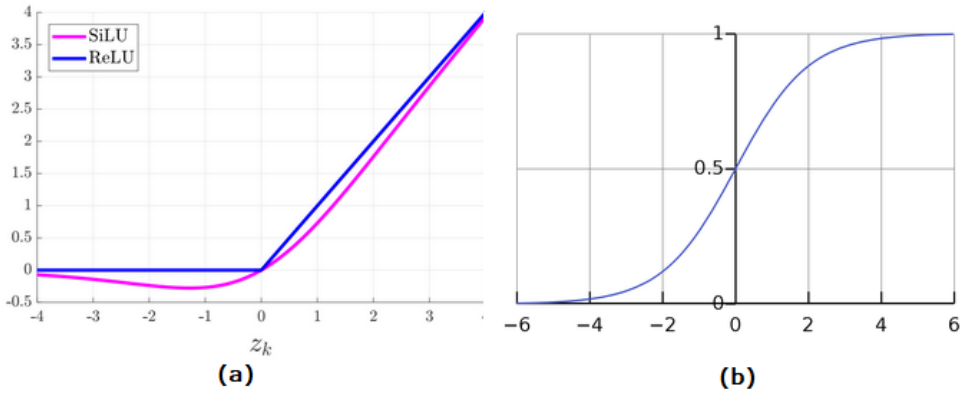

Figura 2.6: (a) SiLU function graph. (b) Sigmoid function graph.

SiLU stands for Sigmoid Linear Unit and, for YOLOv5 architecture, it has been used with the convolution operations in **the hidden layers**, while the **Sigmoid** activation function has been used with the convolution operations in **the output layer**.

Loss Function

YOLOv5 returns three outputs: the **classes** of the detected objects, their **bounding boxes** and the **objectness scores**. Thus, it uses **BCE** (Binary Cross Entropy) to compute the **classes loss** and the **objectness loss**. While **CloU** (Complete Intersection over Union) **loss** to compute the **location loss**. The formula for the final loss is given by the following equation

$$\text{Loss} = \lambda_1 L_{cls} + \lambda_2 L_{obj} + \lambda_3 L_{loc} \tag{2.1}$$

Other improvements

In addition to what have been stated above, there are still some minor improvements that have been added to YOLOv5 and that are worth mentioning.

1. **The Focus Layer**: replaced the three first layers of the network, helping to reduce the number of parameters, the number of FLOPs and the CUDA memory while improving the speed of the forward and the backward passes with minor effects on the mAP (mean Average Precision).

2. **Eliminating Grid Sensitivity**: it was hard for the previous versions of YOLO to detect bounding boxes on image corners mainly due to the equations used to predict them, but the new equations presented in YOLOv5 version helped solving this problem by expanding the range of the center point offset.

3. **The running environment**: the previous versions of YOLO were implemented on the Darknet framework that was written in C, however YOLOv5 is implemented in Pytorch giving more flexibility to control the encoded operations.

About our choice: YOLOv4 vs YOLOv5

In addition to the ability to integrate YOLO into the Pytorch framework, our reasons for choosing version 5 over version 4 are supported by many beanchmarks made by **Roboflow** after a discussion with the author of the model. About them, we can see:

- YOLOv5 **is faster to train** than YOLOv4

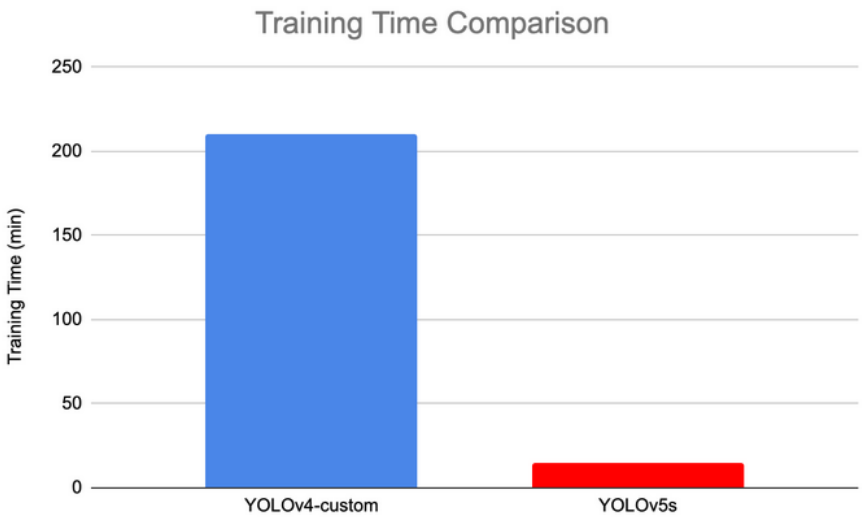

Figura 2.7: Training time comparison

- YOLOv5 is **lighter** than YOLOv4

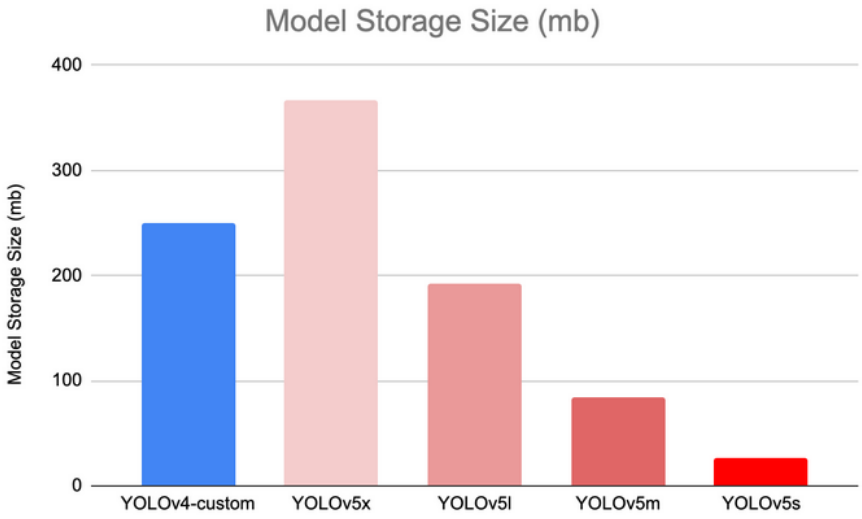

Figura 2.8: Storage size comparison

- YOLOv5 **inference time is better** than YOLOv4 inference time

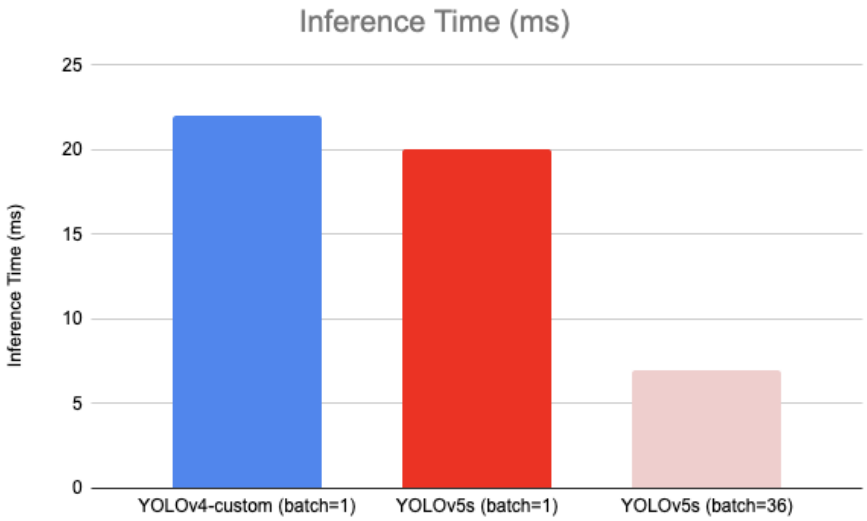

Figura 2.9: Inference time comparison

- YOLOv5 and YOLOv4 **have the same mAP** (mean Average Precision)

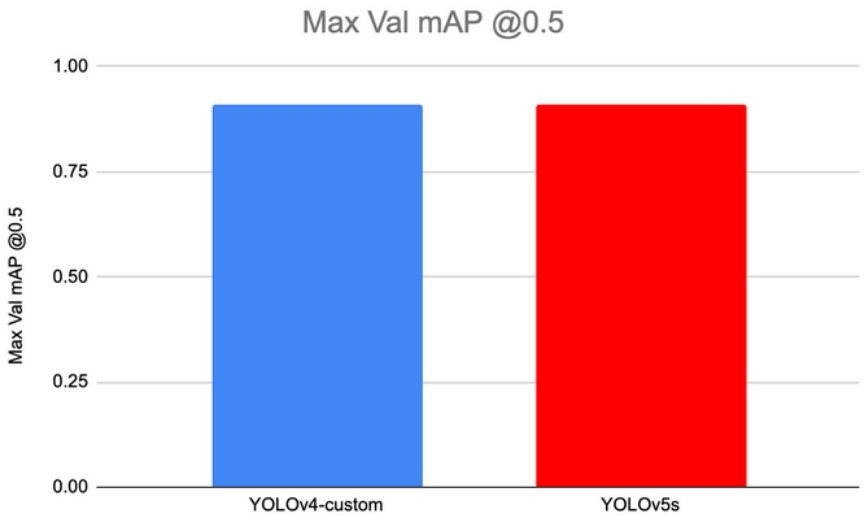

Figura 2.10: mAP comparison

Therefore, we selected YOLO version 5 for several reasons. One of the key changes introduced in this version is the utilization of the CSPNet with the Darknet53 backbone. Additionally, the integration of the **Focus layer** into the CSP-Darknet53 backbone, the replacement of the **SPP** block with the **SPPF** block in the model's neck, and the application of the CSPNet strategy to the PANet model were crucial improvements.

YOLOv5 also addressed the issue of grid sensitivity and now has the ability to easily detect bounding boxes with center points on the edges. Furthermore, YOLOv5 is lighter and faster compared to its previous versions.
In addition, as previously mentioned, we can further process the output bounding boxes from the network to determine whether a fire is present or not based on additional information such as:

- the position of the detected object (fire or smoke) or objects, which usually varies over time in the case of false positives (consider the headlights of a car)
- the confidence level with which the network identifies the object

We can also use with this techniques a temporal aspect (as done with RNN and CNN) by waiting for a certain number of consecutive samples with the presence of fire before triggering a fire alarm.

CAPITOLO 3

FINAL APPROACH

3.1 Dataset Collection and Processing

The learning effectiveness of a deep learner on target features is highly dependent on the annotation of the dataset. Therefore, the quality of the dataset has a very strong relationship with the effectiveness of the model performance.
The dataset provided to us comprises 312 videos, of which 219 have fire/smoke and 93 do not. An initial visual analysis of this dataset revealed the presence of many similar videos, which were then eliminated from the dataset leaving only the most significant ones, and a majority of videos in which smoke is present compared to those in which fire is present. At this point, we proceeded to extract frames from each video using a sampling time of 1 frame per second. Once all frames had been obtained, having dealt with an object detection problem, we continued with the annotation phase, using *Roboflow* as the platform. In particular, two classes were used, one representing fire and the other smoke. Again, in order to favour a greater diversity of samples in the dataset, only the most significant frames were selected.
The annotation file for an image used by the YOLO object detection system contains information about the "bounding boxes" of the objects present in the image. Each line of the annotation file represents one object and consists of five space-separated values:

- **object class ID**: Yolo requires that each type of detectable object has a unique

ID. For example, in our case 0 was assigned to fire and 1 to smoke

- **x center**: the x position of the center of the object's bounding box, expressed as a fraction of the entire image width
- **y center**: the y position of the center of the object's bounding box, expressed as a fraction of the entire image height
- **width**: the width of the object's bounding box, expressed as a fraction of the entire image width
- **height**: the height of the object's bounding box, expressed as a fraction of the entire image height

So, each line will look something like this:

`<object-class> <x> <y> <width> <height>`

In addition, all measurements are normalized to the image size, so the values range from 0 to 1.

As then suggested by the Yolo documentation, a number of background frames amounting to approximately 10 per cent of the entire dataset were included in order to reduce the number of false positives. At the end of this process the dataset comprises 16287 images, where the 80% of which were used as train and 20% as validation. The distribution of images in the dataset for training and validation is shown in the following table:

Dataset	Fire	Smoke	Smoke Fire	No Fire	Smoke labels	Fire labels	Total images
Train	3192	8187	478	1195	9978	4255	16287
Validation	772	2057	112	294	2444	1011	3235

Tabella 3.1: Onfire dataset description

As we can see, the dataset contains far more images in which smoke is present than those in which fire is present, and, of course, this is also reflected in the number of total bounding boxes for each class. An initial idea to try to balance the dataset in question was to perform data augmentation on the fire class samples, but although we can get a large number of samples with that technique, the augmented data *are not independent* of the data to generate them. Therefore, a very large data set obtained only through this technique may be inadequate for the learning task. In order to proceed with the

improvement and balancing, as far as possible, of the dataset, it was decided to extend it, adding more images for each class in a way that would also encourage more divergence, in the face of better generalization. Specifically, the following table provides a description, in numerical terms, of the added samples:

Dataset	Fire	Smoke	Smoke Fire	No Fire	Smoke labels	Fire labels	Total images
Train	3270	4071	6722	2239	12707	22122	16302
Validation	836	1015	1688	426	3181	5490	3965

Tabella 3.2: Additional dataset description

Therefore, the final dataset used has the following configuration:

Dataset	Fire	Smoke	Smoke Fire	No Fire	Smoke labels	Fire labels	Total images
Train	6462	12258	7200	3434	22685	26377	32589
Validation	1608	3072	1800	720	5625	6501	7200

Tabella 3.3: Full dataset description

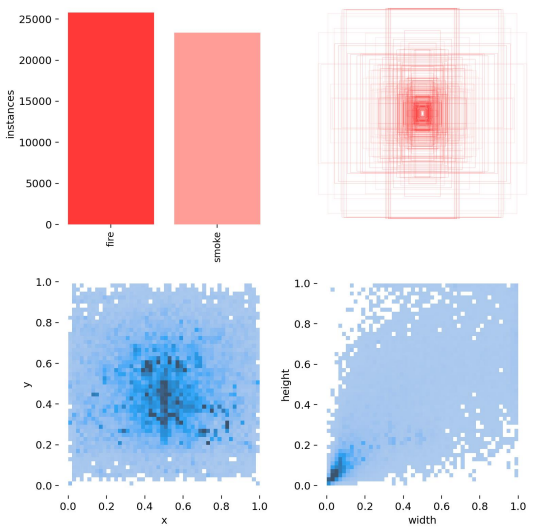

Figura 3.1: Labels

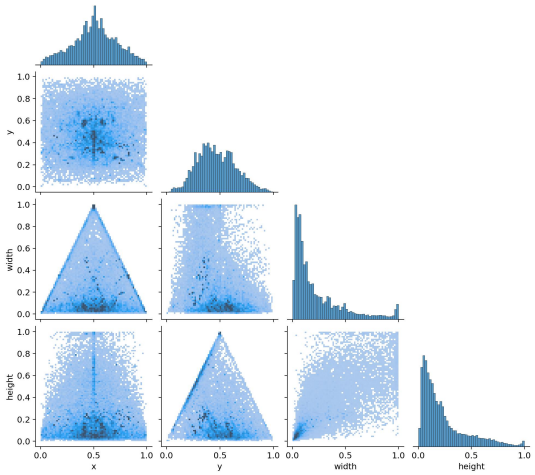

Figura 3.2: Labels correlogram

A sample images from the full dataset is shown in the following figure:

Figura 3.3: Fire

Figura 3.4: Smoke

Figura 3.5: Smoke Fire

Figura 3.6: No Fire

3.1.1 Preprocessing and data augmentation

Data augmentation is an essential strategy when training deep learning models, such as YOLOv5 for object detection. It provides a way to artificially enhance the diversity and size of the training dataset by creating modified versions of the images, thereby increasing the robustness of the model to variations in the input data. Given the inherent challenges associated with object detection tasks, such as dealing with different object scales, orientations, and varying lighting conditions, it was decided to employ specific data augmentation techniques, namely horizontal flipping (represented by the `fliplr` parameter) and mosaic data augmentation (the `mosaic` parameter):

- the *fliplr* parameter dictates the probability of an input image being flipped horizontally during data augmentation. This augmentation is instrumental when the model is expected to recognize objects that can appear in mirror-reversed orientations in real-world scenarios. By setting a fliplr value of 0.5, for instance, we are introducing a 50% chance for each image to be mirrored during the

training process, thereby teaching the model to recognize objects regardless of their horizontal orientation

- the *mosaic* parameter governs the likelihood of using mosaic data augmentation. This technique amalgamates four training images into one, arranged in a mosaic pattern. This approach assists the model in learning to detect various sizes of objects and objects in different contexts within the same image. Setting the mosaic parameter to 1.0 ensures that every image used in the training process will undergo this augmentation, providing the model with more diverse and complex training data

The use of fliplr and mosaic data augmentations ensures our model's increased robustness and improved generalization capability on unseen data.
In addition to the data augmentation strategies described, it is also important to mention that all images in the dataset were resized to a uniform size of 640 x 640 pixels. This preprocessing step is critical for several reasons.
Firstly, having all images in the dataset of the same size simplifies the data pipeline and ensures that batches of images can be processed simultaneously, thereby utilizing GPU resources more effectively. This batch processing is crucial for accelerating the training phase, and having inconsistent image sizes could lead to complications or inefficiencies.
Secondly, resizing to a square format, specifically 640 x 640 pixels, is important as it retains the aspect ratio of the objects within the images. Distortion or skewing could occur if the images were resized to a non-square format, which could potentially negatively impact the model's ability to accurately detect objects.
Finally, the choice of 640 x 640 as the specific size is a trade-off between computational efficiency and detection accuracy. Larger images would contain more detailed information, potentially improving the model's performance, but at the same time, they would increase the computational cost and the amount of memory required for training. On the other hand, smaller images would be faster to process but might lack necessary details. Thus, the chosen size of 640 x 640 pixels offers a balanced compromise.

3.2 Training

The first step taken to develop the proposed fire detection system was the training of the YOLOv5 network. As stated in the section dedicated to the YOLO model, there are different versions of YOLO that differ solely in the number of parameters. Consequently, this variation affects the network's performance in terms of **processing speed** and **memory consumption**. After conducting extensive research on the addressed problem and comparing various benchmarks published by the authors of the model, the choice has fallen on the "*s*" (*small*) version.

Indeed, the authors themselves state that larger models such as YOLOv5x and YOLOv5x6 yield better results in almost all cases but come with more parameters, require more CUDA memory for training, and are slower to execute. Given that the problem at hand is real-time fire recognition, it necessitates the use of a model capable of detecting anomalies as quickly as possible. Moreover, in addition to speed, memory consumption is a key parameter for systems operating within the context of the addressed problem. Hence, to support the authors' claim and considering implementations on devices with limited resources, it is highly recommended to use the YOLOv5s/m versions. This is precisely why we have chosen the previously mentioned version.

The training of the selected model was conducted using the script called `train.py`, provided by the authors themselves. The overall structure involves implementing the entire workflow for model training, including preprocessing operations, actual training, evaluation metric computation, and result generation. An overview of the script includes the following steps:

1. **Importing Libraries**: the script starts with importing necessary libraries such as PyTorch, Numpy, and other YOLOv5-specific dependencies

2. **Command-Line Argument Parsing**: the script parses the command-line arguments to configure various training options, such as data path, model architecture, optimizer, image dimensions, number of epochs, and other configuration options

3. **Data Loading**: the dataloader is set up to load training and validation data, supporting various dataset formats and options for specifying the data path and

dataset format

4. **Model Configuration**: the YOLOv5 model is configured for training, supporting different variants of the model and allowing it to be specified as an argument for training

5. **Training Phase**: during this phase, the chosen YOLOv5 model is trained using the provided training data and specified optimizer (defaulting to Stochastic Gradient Descent). Each training iteration involves the following steps:

   - loading a batch of training images along with their corresponding labels (bounding boxes and object classes) from the dataloader
   - passing the images through the model to obtain predictions
   - calculating the loss based on the predictions with respect to the associated labels, particularly using Intersection over Union (IoU), that is a quantitative measure to score how the ground-truth and predicted boxes match helping to understand if a region has an object or not, and Binary Cross Entropy (BCE) metrics
   - performing backpropagation to compute the gradients of the loss with respect to the model's weights
   - updating the model's weights using the specified optimizer and associated algorithms as per the desired configuration

6. **Validation**: after each training epoch, the validation phase is executed to assess the model's performance. During this phase, predictions are made on the images belonging to the Validation Set, and several evaluation metrics are calculated, that's are:

   - **Precision**, that measures how much of the bounding boxes predictions are correct, which corresponds to

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.1}$$

   - **Recall**, that measures how much of the true bounding boxes were correctly predicted, which corresponds to

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.2}$$

- **mAP_0.5**, that is the mean Average Precision (**mAP**) at IoU (Intersection Over Union) threshold of 0.5. 'mAP_0.5:0.95' is the average mAP over different IoU thresholds, ranging from 0.5 to 0.95.

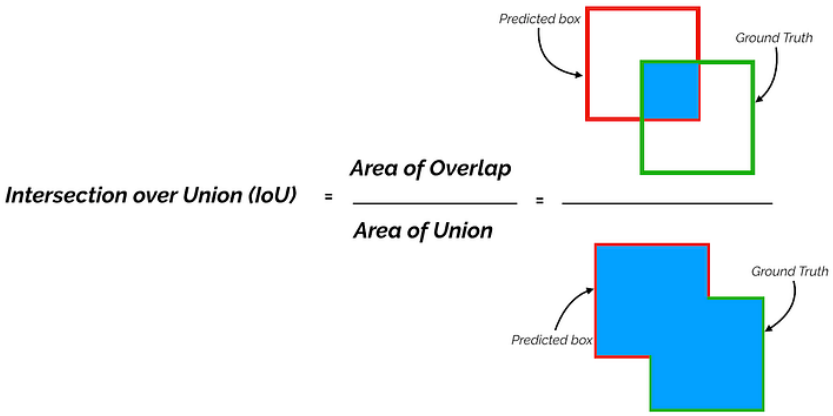

Figura 3.7: Intersection over Union

So, the mAP compares the ground-truth bounding box to the detected box and returns a score. The higher the score, the more accurate the model is in its detections.

7. **Weight Saving**: after completing the training or whenever improvements are observed in the model based on the previously analyzed metrics, the weights are automatically saved. Specifically, the concept of improvement is understood according to a function the authors termed "Fitness," which represents exactly what is intended to be maximized: it is a weighted combination of the metric `mAP@0.5` that contributes 10 % and the `mAP@0.5:0.95` that contributes the remaining 90 %, without considering either Precision or Recall

8. **Results Generation**: the script generates various types of results, such as images with model predictions overlaid on the input image, log files containing training and validation statistics, validation metric graphs, and so on

At this point, after providing a high-level description of the training process conducted using the script provided by the authors, let's discuss the strategy employed and describe the chosen configuration parameters. Given the complexity of the problem and having an adequately large dataset, the training was conducted on the entire model structure, from the backbone to the head. The training at this stage was carried out by configuring:

- **img**: this parameter represents the size of the images in the Training and Validation sets and was set to 640. Specifically, this size was chosen as the minimum dimension for conducting training, as detecting small objects in lower-resolution images would result in significantly lower inference performance

- **batch**: this parameter was set to 64, based on the available hardware capabilities. Following the authors' advice, the preferred batch size is generally the one that the hardware can efficiently support. Having a small batch size can lead to poor batch normalization statistics, which should be avoided

- **epochs**: the parameter representing the total number of training epochs was set to 200. After conducting various experiments, this value was chosen as a compromise between the computational time required to train the model and the possibility of achieving significant performance. Additionally, the presence of early stopping helps prevent overfitting and stops the training when the model has reached an optimal level of generalization regarding the training dataset

- **data**: this parameter specifies the path of the data configuration file. The YAML file contains information about the structure of the training and validation data, including paths to image files, label files, and the number of classes to be detected. This parameter was configured based on the location of the aforementioned file, named `data.yaml`

- **weights**: this parameter specifies the path to the initial model weights. In this case, the YOLOv5s model (Small version) will be used as a starting point for training. The use of pre-trained weights (transfer learning) helps improve the model's performance as it has already learned from a larger and diverse dataset

- **save-period**: this parameter specifies the frequency at which the trained model's weights are saved to disk during training. In this case, the weights are saved every 10 epochs. This option was useful to ensure that the training progress was regularly saved to avoid data loss in case of any issues during the training process

In YOLOv5 training, it is recommended to use the pre-defined hyperparameters present in the `hyp.scratch.yaml` file because these have been optimized and fine-tuned especially for training YOLOv5 models from scratch. This configuration file contains a set of

hyperparameters that work well as a starting point for the initial training of the model on a new dataset. By using these pre-defined hyperparameters, one can achieve more stable training and better results compared to using arbitrary values for the hyperparameters, and that is why the training we conducted was based on the default one. Now, let's through each hyperparameter contained in the `hyp.scratch.yaml` file and what they represent:

1. **lr0 = 0.01**: this is the initial learning rate used by the optimizer during the training (SGD in our case). The learning rate controls the size of the steps that the optimizer takes during the weight updates of the model. An appropriate value (like the default one) can help ensure stable and efficient convergence of the model.

2. **lrf = 0.01**: this parameter represents the learning rate reduction factor. It is used to gradually decrease the learning rate over the course of training epochs, especially for fine-tune the model's weights as the training progresses.

3. **momentum = 0.937**: this parameter is used in optimization algorithms like SGD and represents the amount of consideration given to previous gradients in the direction of weight updates, contributing into a more stable convergence during training.

4. **weight_decay = 0.0005**: this is a term used to introduce a penalty on the model's weights during training, helping in reduce the risk of overfitting.

5. **warmup_epochs = 3.0**: this parameter represent the number of epochs during which the learning rate gradually increases from zero to the initial value `lr0`, helping the model start training with a lower learning rate and gradually increases it, allowing for a stabilizing initial phase.

6. **box = 0.05**: this parameter represent the attenuation used in the bounding box regression loss, so an higher value can provide grater attention to the bounding box coordinates during weight updates.

7. **cls = 0.5**: this parameter represents the attenuation used in the classification loss, so an higher value of cls can provide greater attention to class predictions during weight updates.

8. **cls_pw = 1.0**: this parameter represent a class prediction weight factor, that helps in balance the weight of classes based on the number of samples per class in the dataset, in this way classes with fewer samples will have a higher weight compared to those with more samples.

These are some of the main hyperparameters contained in the descripted file. There are also other hyperparameters that control specific training details related to image augmentations and scaling that play an essential role in enhancing the training process of YOLOv5 like mosaic augumentation, mixup augumentation, scale, hsv (hue,saturation,value), exposure and so on, but the ones listed above are the most significant ones for the training from scratch.

3.2.1 Training results

In the analysis of our object detection model's results, special attention was given to three key metrics: mAP_0.5, precision, and recall. The mAP_0.5 (Mean Average Precision with IoU > 0.5), representing the average of the APs for each class in the dataset, has shown a consistent and positive trend over time. This confirmed the effectiveness of the model in terms of its ability to detect objects of interest with significant overlap accuracy.

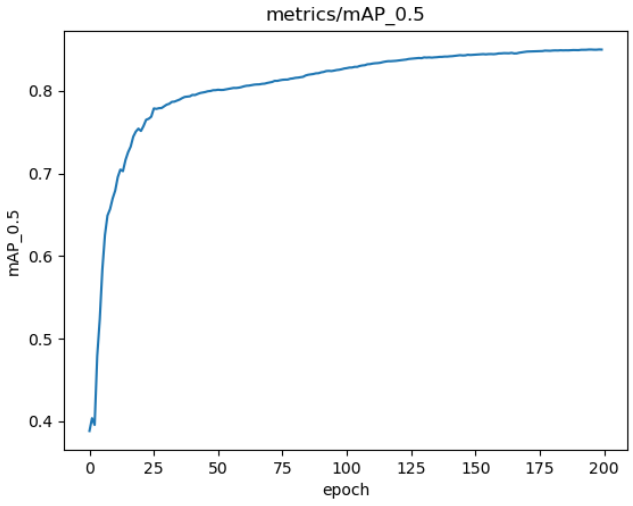

Figura 3.8: Intersection over Union

The precision graph demonstrated a good ability of the model to reduce false positives. This indicates that when our model detects an object, it is highly likely that the object is indeed present.

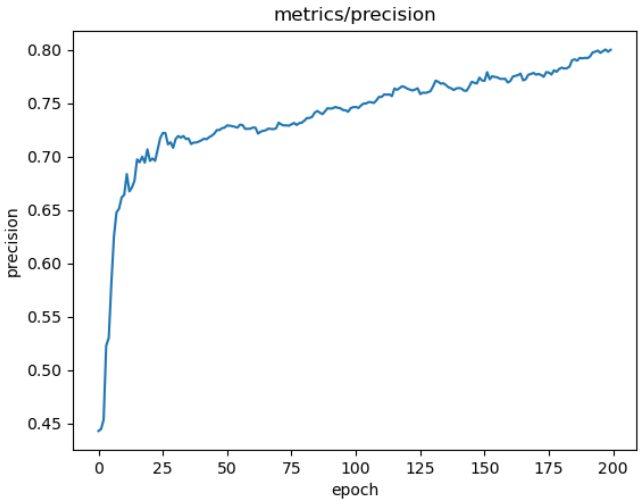

Figura 3.9: Intersection over Union

Finally, the recall graph showed a good balance relative to precision. This suggests that the model has been able to detect most of the objects of interest present in the images, thereby reducing false negatives.

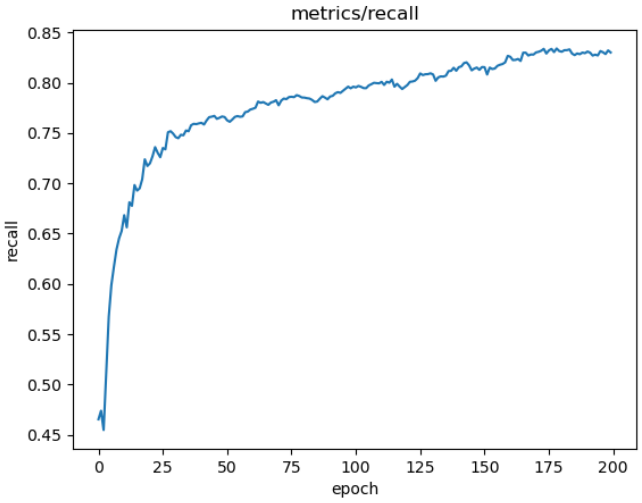

Figura 3.10: Intersection over Union

Overall, the graphs indicate a good balance between precision and recall, and an increasing mAP_0.5, confirming the efficacy of our model in object detection.
It can be seen in the following figures that the model can correctly detect fire and smoke pictures in different scenairos.

Figura 3.11: val_batch1_labels

Figura 3.12: val_batch1_pred

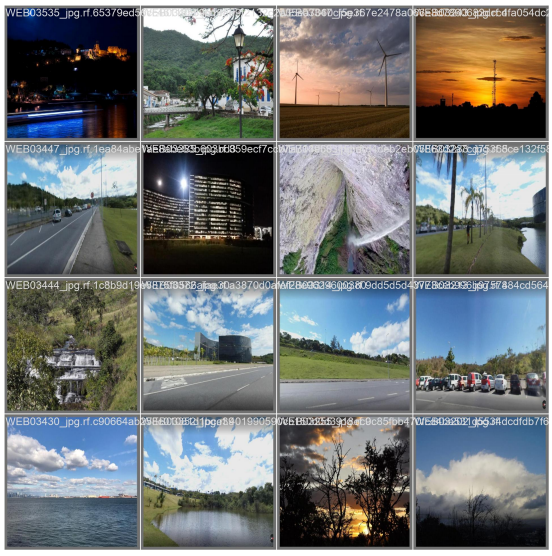

Figura 3.13: val_batch2_labels

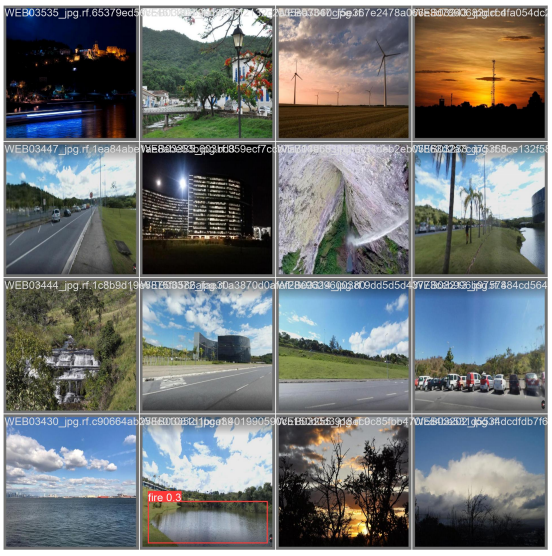

Figura 3.14: val_batch2_pred

3.3 Detection Algorithms

Now, let's delve into the functioning of the detection algorithm employed in the final version of our network. It utilizes a series of features, including:

- Dynamic selection of the `Fire_Threshold`. As previously mentioned, this term refers to the number of consecutive frames in which the network must perform **at least one** detection.
- *Accumulation Matrices* is a matrix of size width × height, where each element $x_{i,j}$ represents the weighted sum of confidences assigned by the network to the presence of fire at that specific pixel over the course of the `Fire_Threshold` frames.

3.3.1 Dynamic Selection of the *Fire_Threshold*

The underlying idea behind this function is as follows: in order for the network to examine a sufficient number of frames to be certain of the presence of fire, a minimum duration of `FIRE_DETECTION_TIME` seconds must elapse. Assuming a video is recorded at a certain frame rate denoted as FPS, we can calculate the `Fire_Threshold` as:

$$Fire\_Threshold = \lfloor \text{FIRE\_DETECTION\_TIME} \cdot \text{FPS} \rfloor$$

This equation determines the minimum number of consecutive frames required for the network to make a confident fire detection decision.

3.3.2 Accumulation Matrices

Let's now examine in detail the functioning of the Accumulation Matrix. This matrix "covers" each pixel of the video as a grid and accumulates the network's confidence regarding the possibility of that pixel belonging to fire or not. The values added to the matrix are weighted, where the weight is determined by the sum of the inverse of the confidence values for that pixel. After accumulating *Fire_Threshold* consecutive frames in which a flame or smoke is present, the algorithm proceeds as follows.
For each pixel appearing in at least one bounding box, the weighted average is calculated

as:

$$\text{Prob}_{\text{fire}}[x_{i,j}] = \frac{\sum_{k=1}^{\text{Fire\_Frames}_{i,j}} \text{confidence}_k(x_{i,j}) \cdot \text{weight}_k(x_{i,j})}{\sum_{k=1}^{\text{Fire\_Frames}_{i,j}} \text{weight}_k(x_{i,j})}$$

where:

- $\textit{Fire\_Frames}_{i,j}$ represents the number of times the pixel has appeared in a bounding box and is within the range [1, *Fire_Threshold*]
- $\textit{confidence}_k(x_{i,j})$ represents the confidence of the k-th bounding box to which the pixel $x_{i,j}$ belongs.
- $\textit{weight}_k(x_{i,j})$ denotes the weight associated with the pixel $x_{i,j}$ within the k-th frame in the sequence $[1, \textit{Fire\_Frames}_{i,j}]$.

In particular, we chose to define $\textit{weight}_k(x_{i,j})$ as $\frac{1}{\text{confidence}_k(x_{i,j})}$. Thus, the above equation becomes:

$$\text{Prob}_{\text{fire}}[x_{i,j}] = \frac{\sum_{k=1}^{\text{Fire\_Frames}_{i,j}} 1}{\sum_{k=1}^{\text{Fire\_Frames}_{i,j}} \text{weight}_k(x_{i,j})}$$

Following the computation of this matrix, if there exists at least one point where the probability is greater than a threshold indicated as `confidence_threshold`, then the presence of fire is declared.

3.4 Other Algorithm considered

3.4.1 Aritmetic Mean

In the initial implementation, after defining a static *Fire_Threshold*, we implemented the accumulation matrix by simply adding the confidence associated with the model's prediction for each pixel (if the pixel falls within the bounding box) or 0 otherwise. During the analysis phase, we divided this accumulated value by the number of frames in which fire was present, effectively obtaining the average probability for that pixel. After obtaining the average probability for each pixel, a fire is declared if there exists at least one pixel whose probability exceeds a fixed `confidence_threshold`.

3.4.2 Presented Detection Algorithm without Dynamic Selection of the *Fire_Threshold*

We can consider this version as a specific case of the presented algorithm, where, regardless of the video's frame rate (FPS), the number of consecutive frames required for the presence of fire was fixed to a predetermined value decided as a configuration hyperparameter.

3.5 Experimental Results and Discussion

To evaluate the performance of the proposed algorithms, we created our own dataset to use as a test set. The videos included in this dataset are all previously unseen and therefore uncorrelated with those in the training set. We collected these videos primarily from the internet, taking care to:

- only consider videos with a fixed or slowly moving camera perspective to simulate a security camera setup
- cover a wide range of scenarios to ensure the dataset is as representative as possible

By following these guidelines, we aimed to create a diverse and realistic dataset for evaluating the performance of the fire recognition algorithms.

3.5.1 Test Set

Our test set consists of 173 diverse videos depicting a wide range of scenarios. Among these videos:

- 60 videos do not contain any fires (True Negatives - TN).
- 113 videos contain flames and/or smoke (potential True Positives - TP).

We carefully curated the test set to ensure a balanced representation of both fire and non-fire scenarios. This balanced distribution allows for an accurate evaluation of the models in both absence of fires (TN) and presence of fires (TP), providing a variety of cases to thoroughly assess their performance. All labels were created by us and are considered as ground truth information for the purpose of evaluation. Each video is provided with labels, and these labels fall into the following categories:

- **Empty Labels**: these correspond to videos that do not contain any fires (True Negatives - TN)
- **Containing a Value**: these labels consist of an integer representing the timestamp (in seconds) when a fire is clearly visible within the video

In the sample images presented, we have highlighted situations where clouds or a sunset share similarities in color and movement with smoke or fire. These instances can be particularly challenging for fire recognition algorithms, as the visual characteristics of clouds or a sunset might lead to false positives or misclassifications. Addressing such scenarios effectively is crucial for improving the overall accuracy and robustness of the fire recognition system.

Figura 3.15: Clouds

Figura 3.16: Sunset

Figura 3.17: Smoke

Figura 3.18: Fire

3.5.2 Results on the test set with different algorithms

In this section, we present the results obtained by testing the proposed fire recognition algorithm with various hyperparameters on our test set. The algorithm's performance is evaluated using different combinations of model confidence (*model_conf*), evaluator threshold (*evaluator_limit*), and fire threshold (*fire_limit*). We have recorded the true positives (TP), false positives (FP), false negatives (FN), precision (P), recall (R), accuracy (acc) and F-score metrics for each configuration.

model_conf	**eval_limit**	**TP**	**FP**	**FN**	**P**	**R**	**acc**	**F**
0.25	0.4	107	15	3	0.877	0.973	0.896	0.923
0.25	0.45	106	15	4	0.876	0.964	0.890	0.918
0.25	0.5	104	15	6	0.874	0.945	0.879	0.908
0.35	0.4	106	13	4	0.891	0.964	0.902	0.926
0.35	0.45	105	13	5	0.890	0.955	0.896	0.921
0.35	0.5	104	13	6	0.889	0.945	0.890	0.916

Tabella 3.4: Results obtained with the proposed algorithm

model_conf	fire_limit	eval_limit	TP	FP	FN	P	R	acc	F
0.25	3	0.4	107	25	3	0.811	0.973	0.838	0.884
0.25	3	0.45	106	23	4	0.822	0.964	0.832	0.887
0.25	4	0.4	107	19	3	0.849	0.973	0.873	0.907
0.25	4	0.45	106	19	4	0.848	0.964	0.867	0.902
0.25	5	0.4	107	19	3	0.849	0.974	0.873	0.907
0.25	5	0.45	106	19	4	0.848	0.964	0.867	0.902
0.35	3	0.4	107	21	3	0.835	0.972	0.861	0.900
0.35	3	0.45	106	20	4	0.841	0.963	0.861	0.896
0.35	4	0.4	107	18	3	0.856	0.973	0.879	0.910
0.35	4	0.45	106	18	4	0.854	0.963	0.872	0.901
0.35	5	0.4	107	18	3	0.856	0.973	0.879	0.910
0.35	5	0.45	106	18	4	0.854	0.963	0.872	0.901

Tabella 3.5: Dynamic mean without Dynamic Fire Treshold

model_conf	fire_limit	eval_limit	TP	FP	FN	P	R	acc	F
0.25	3	0.4	97	17	13	0.850	0.881	0.826	0.865
0.25	3	0.45	91	16	19	0.850	0.827	0.797	0.838
0.25	4	0.4	97	17	13	0.850	0.881	0.826	0.865
0.25	4	0.45	91	14	19	0.866	0.827	0.809	0.846
0.25	5	0.4	96	17	14	0.849	0.872	0.820	0.860
0.25	5	0.45	92	13	18	0.876	0.836	0.820	0.855
0.35	3	0.4	105	20	5	0.840	0.954	0.855	0.893
0.35	3	0.45	99	18	11	0.846	0.900	0.832	0.872
0.35	4	0.4	104	18	6	0.852	0.945	0.861	0.896
0.35	4	0.45	98	16	12	0.859	0.890	0.838	0.875
0.35	5	0.4	103	18	7	0.851	0.936	0.855	0.891
0.35	5	0.45	98	16	12	0.859	0.890	0.838	0.875

Tabella 3.6: Arithmetic Mean

3.5.3 Processing Frame Rate and Memory Usage

In our work, we also evaluated two significant performance metrics: *processing frame rate* and *memory usage*.

The processing frame rate, in our case around 82.32 frames per second, is particularly crucial in real-time applications. The frame rate indicates how many frames the model can process in a second. A higher frame rate ensures a smoother transition between frames and less lag, enabling the system to provide real-time or near real-time insights. This aspect is particularly critical in video surveillance applications where a delay in detection could lead to undesirable consequences.

Memory usage, approximately 1GB in our experiments, is another vital performance indicator. The amount of memory consumed by the model during its operation can heavily impact its deployability, especially in resource-constrained environments. A model that requires excessive memory might not be practical for deployment on low-power devices or embedded systems, such as drones or handheld devices.

Thus, striking a balance between processing speed (frame rate) and resource consumption (memory usage) is an essential aspect of designing a model for real-time applications. The optimal balance depends on the specific constraints and requirements of the use case. In our work, we found that the YOLOv5 model was able to deliver satisfactory performance on both these fronts, making it a feasible option for real-time video object detection tasks.

CAPITOLO 4

CONCLUSIONS

In this study, we have presented and discussed an approach to video classification based on object detection, using specifically the YOLOv5 architecture. The objective was to detect and classify specific objects within complex videos, and the results obtained demonstrate a satisfactory level of precision and recall.

However, as with any machine learning approach, certain limitations emerged during our work. In particular, a significant challenge was the management of car headlights in the videos. Since headlights emit intense light, they can create significant variations in the contrast and brightness of the images, which can, in turn, affect the performance of the object detection model. This issue is especially prominent in night scenes or in low-light conditions, where car headlights can dominate the scene.

Some potential solutions to this challenge might include the use of image enhancement techniques to adjust the contrast and brightness of images, or training the model on a dataset that explicitly includes images with car headlights. However, these solutions require further research and experimentation.

Overall, despite these challenges, we believe our work demonstrates the potential of object detection for video classification. Furthermore, the problems encountered provide valuable insights into how to further improve the model's performance. We are confident that with further research and development, it will be possible to overcome the current limitations and further enhance our model's precision and robustness.

ELENCO DELLE FIGURE