Alma Mater Studiorum - Università di Bologna

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea in Ingegneria Informatica Magistrale

Investigations on the Effectiveness of Tools for Optimizing Object Detection Training Performance

Tesi di laurea magistrale

in

Intelligent Systems M

Presentata da:

Pietro Santandrea

Relatore:

Prof.ssa Michela Milano

Correlatori:

Andreas Leich

Gurucharan Srinivas

Sessione II

Anno Accademico 2022-2023

A nonna Checca,
l'angelo che mi ha
accompagnato
in questo viaggio.

Abstract

Convolutional Neural Networks (CNNs) have emerged as a pivotal technology in computer vision, particularly excelling in object detection tasks. Their adeptness at feature extraction and recognition has elevated them to the best technology for that purpose. Nevertheless, training a CNN-based object detector that performs well takes work. Often, every new scene, changed weather conditions, and other factors hamper the recognition capability of a CNN-based object detector. Thus, in practice, a re-training is necessary for every novel measurement campaign. Various tools have been devised to enhance the performance of CNN-based object detectors. Techniques like data augmentations facilitate the augmentation of training images, introducing diversity. Dropout regularization prevents overfitting, while hard example mining focuses on identifying the most challenging instances to refine the model. This thesis delves into an investigation of the efficacy of these tools in improving learning efficiency and enhancing network training. Furthermore, the research attempts to emulate the typical scenario developers face when applying a model to a new context, particularly on a customized dataset characterized by a scarcity of images and significantly different class frequencies. The focal model architecture in this study is YOLOv7, and the dataset comprises a limited collection of urban scene images with diverse perspectives. The outcomes of this research culminated in the development of multiple models, the best of which achieved a score of 80.3% on mAP@.5 and 50.7% on mAP@.5:.95, with an increase over the base model of 15.1% after training with a dataset with only 174 training images.

Contents

Chapter 1

Artificial Intelligence and Object Detection

1.1 Introduction

Artificial intelligence (AI) is a cutting-edge field that has impacted many facets of contemporary technology and brought revolutionary possibilities to a wide range of applications. One of the most important among them is computer vision, which gives machines the capacity to comprehend and interpret visual input. It enables computers and systems to derive meaningful information from digital videos, images or other visual inputs and and perform actions based on the input data.
Among computer vision tasks, object detection is one of the most crucial ones. It deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos. It is essential to AI systems' development since it helps them become more perceptive and can navigate and interact with their surroundings more like humans. Object detection's application is widespread:

- **Autonomous Driving:** To identify other vehicles, traffic signs, and pedestrians, self-driving cars utilize object detection [1]. This technology enables cars to identify environmental risks and ensure safety by recognizing approaching vehicles and obstacles.
- **Agriculture:** Counting, tracking animals, and evaluating the quality of agricultural output are just a few of the jobs in agriculture that use object detection. During processing, machine learning algorithms can detect damaged products as humans [2], which helps to enhance quality control.
- **Video Security:** A crucial aspect of many security applications, especially

video surveillance, is object detection. Through computer vision, it is used to identify people in dangerous or hazardous situations [3], stop suicide attempts [4], and automate inspection chores in far-off places.

- **Healthcare:** Medical science has advanced significantly due to object detection. It is especially useful for analyzing pictures, scans, and photos in medical diagnostics. Early disease identification, including tumor detection, is facilitated by machine learning techniques that use object detection in CT [5] and MRI [6] scans.

1.1.1 DLR - Deutsches Zentrum für Luft- und Raumfahrt

The following thesis was made at the German Aerospace Center (DLR) in Berlin. It is Germany's national center for aerospace, energy, and transportation research, founded in 1969. It is headquartered in Cologne and has many locations around Germany, including Berlin. It has approximately 11.000 employees. DLR is involved in both domestic and foreign collaborations in a broad spectrum of research and development initiatives, and it represents the German federal government as the country's space agency, overseeing the development and execution of the country's space project. DLR is a project management institution that oversees and responds to the organizational and technical execution of projects supported by several federal ministries in Germany. The Berlin office belongs to the Transport Institute, whose mission is: "Shaping the mobility of people and the transportation of goods," particularly automotive, rail systems, traffic, and mobility management. The department's group takes the name IGM (Information Gathering and Modelling) and specializes in the field of sensor technology and data applied to rail and road traffic. Finally, the team is called AIS (Algorithms and AI for Sensor data processing) and focuses on AI solutions to the topics described.

1.2 Neural Network

One of the most powerful and interesting artificial intelligence solutions is Neural Networks. They are inspired by the human brain's intricate network of interconnected neurons. A neuron is connected to many, and they communicate with each other by sending electrical and chemical signals, called neurotransmitters, across a tiny space called a synapse, between the axons and dendrites of nearby neurons. This operation is repeated for about 86 billion neurons and 100 trillion connections, and it enables human beings to think, recognize, and reason. Similarly, artificial neural networks are composed of layers of neurons in which each neuron is connected with all preceding and following ones. Most nets are feed-forward, meaning that data moves through them in only one direction.

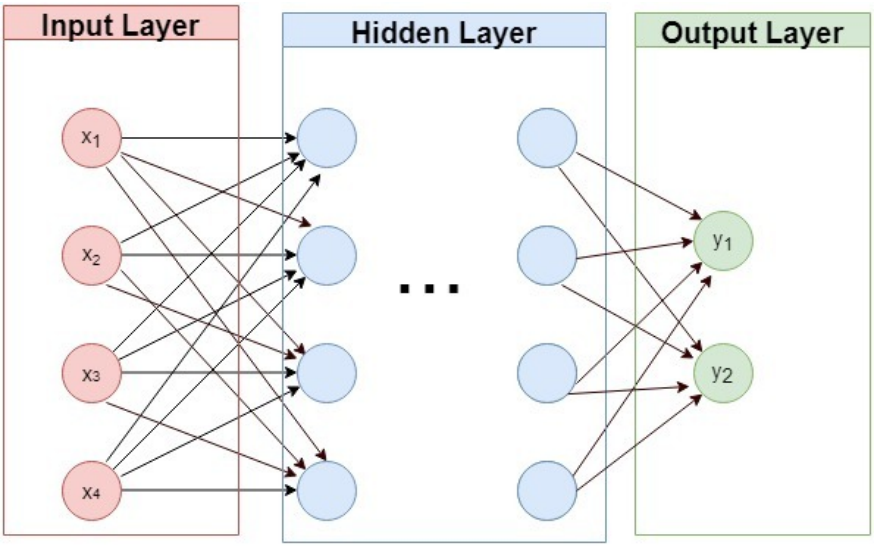


Figure 1.1: Common Neural Network architecture

The first layer of neurons is called the input layer through which information enters the network; it must be as big as the data processed. For example, if an image size is 640x640x3, the input layer has the same number of neurons. Then, data are passed from layer to layer, called hidden layers, to the last one, called output layer (see Fig 1.1). The latter is responsible for showing the model output, which must be as big as the predicted outputs.

Each neuron has associated weights, which are crucial for its functioning. These weights represent the strength of connections between neurons. When information passes through the network, these weights are multiplied by the input values. The weighted inputs are then summed up, passing this sum through an activation function. The activation function decides whether a neuron should be activated or not.

This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations.

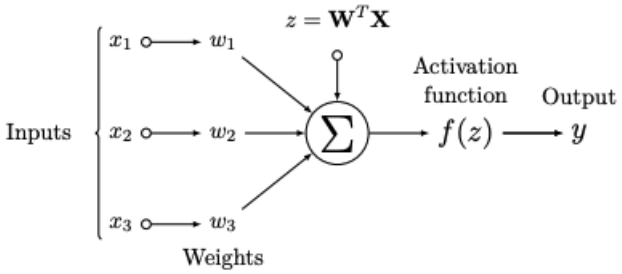


Figure 1.2: Functioning of a neuron

This technology can process any data, from Excel tables to medical reports, from images to sentences. It begins with training, which is when the network learns by adjusting the weights so as to be able to predict the correct class label of input samples. These samples are the so-called Ground Truth (GT) data, which refers to the true information of a specific problem or question; in other words, it is the input training data and the corresponding correct output. In an optimal situation, there is GT data available; numerous public datasets provide GT data to train models. In practice, it can be different, especially in tailored situations; it is common to miss large quantities of GT data, which are also difficult to collect manually. This knowledge is transformed into numerical values and fed into the network during a phase called forward propagation. During the phase, the data travels through the layers, encountering weighted connections and biases. Each neuron processes the information, computes the weighted sum of its input, and then applies a bias to it (1.1).

$$\text{net}_i = \sum_{i=1}^{n}(w_i \times x_i) + b \tag{1.1}$$

$$\text{ReLU}(x) = \max(0, x) \tag{1.2}$$

Then, it passes this resultant term through an activation function, gradually generating the output prediction. The ReLU function (1.2) is one of the most frequently used functions. After forward propagation, the network compares its predictions to the actual outcomes from the dataset, computing the disparity through a loss function that quantifies the dissimilarity between the predicted values and the actual data; basically, it tests how wrong the prediction is. At the end of this first process, two new actors come into play: the optimizer and the concept of learning rate. The

first assesses the current state of the network using the loss function and employs mathematical techniques, such as gradient descent, to adjust the network's internal parameters (weights and biases). It iteratively refines these parameters, steering the network towards the configuration that minimizes the loss function. This process must tune the weights gradually; the neural network aims to generalize the ground truth data sufficiently to correctly predict new unknown data that belongs to the same information it has learned. Therefore, the optimizer must govern the weights carefully and according to specific rules so they do not learn to recognize only the last data obtained but keep a historical generalization. The learning rate serves this purpose; it determines the size of the steps taken during these adjustments. Too large, and the optimizer might overshoot the optimal values; too small, and the process becomes painstakingly slow. In other words, it decides how much neurons learn from the loss function. Updating the weights is called back-propagation and is repeated for every ground truth information.

The fundamental idea behind back-propagation is to decrease the network's total error by computing the gradient of the error concerning the network weights and using that information to update the weights. Until the network reaches a good level of generalization capability, it is trained multiple times on the same dataset; each training is called *epoch*. A neural network is often trained for hundreds of epochs.

1.3 Convolutional Neural Network (CNN)

A convolutional neural network, also known as CNN, is a neural network that specializes in processing data, especially images, with a grid-like topology. For example, this thesis will use YOLOv7, one of the most important CNNs. A digital image is a three-dimensional matrix in which each cell represents a pixel with three color channels, like RGB (Red, Green, Blue). Just as each human neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged to detect simpler patterns first (lines, curves) and more complex patterns (faces, objects) further along. By using a CNN, one can enable sight to computers.

CNN Architecture

A CNN is largely composed of 3 types of layers: convolutional layer, pooling layer, and fully connected layer.

Convolutional layer

The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load. It applies a convolution operation to the input, passing the result to the next layer. The convolution operation converts all pixels in its receptive field into a single value; as a consequence, the visual expressiveness that previously had N pixels is condensed into one. It can be understood, therefore, how a convolutional layer simplifies the image from a dimensional point of view, extracting the most significant patterns, which in jargon are called features. This process is achieved by performing a dot product between two matrices; the first is the set of learnable parameters, otherwise known as a kernel or filter, and the other matrix is the restricted portion of the receptive field. During the forward pass, the kernel slides across the dimensions of the processed image, producing the image representation of that receptive region. The sliding size of the kernel is called stride. The main reasons why CNN adopted a large number of convolutional layers are as follows. First, an image can have millions or thousands of pixels. However, while processing it using a convolutional kernel, it is possible to detect meaningful information of tens or hundreds of pixels. This means that the network needs to store fewer parameters, which reduces the model's memory requirement and improves the model's statistical efficiency. Second, if computing one feature at a spatial point is useful, it should also be useful at another spatial point. This means that slice neurons are constrained to use the same set of weights for a single two-dimensional. In a traditional neural network, each element of the weight matrix is used once and then never revisited. In contrast, a convolutional network has shared parameters; weights applied to one input are the same as those applied elsewhere.

Pooling layer

An essential part of convolutional neural networks (CNNs) is the pooling layer, which replaces the output at a given position with a summary statistic extracted from nearby outputs. This tactical move accomplishes two goals: it reduces the representation's spatial dimensions while lessening the network's computational load and weight complexity. Furthermore, each slice of the representation undergoes the pooling process individually, which enhances the model's capacity to identify and extract unique characteristics at various levels. CNN architectures can perform a variety of pooling tasks. They include using the L2 norm in a rectangular neighborhood, a weighted average based on the distance from the center pixel, and the simple average computation over the rectangular neighborhood. Nevertheless, max pooling has become the best option out of all these possibilities. Because it is so ef-

fective at capturing the most notable features in each region, it is widely used. Max pooling works by choosing the maximum output value from the local neighborhood. This improves feature extraction robustness and makes it easier for the network to identify details and patterns in the input data; as a result, max pooling is now a fundamental component of CNN design and optimization, contributing significantly to the models' effectiveness.

Fully Connected Layer

Each neuron in a fully connected layer uses a weights matrix to apply a linear transformation to the input vector before a non-linear activation function is activated. This architecture ensures that every connection between layers is present, meaning that each layer's output is affected by each layer's input, which promotes thorough information integration. Convolutional neural networks (CNNs) often use fully connected layers in their final layer, which enhances the network's overall capacity to recognize complex patterns and features. The ability of fully connected layers to learn and model intricate non-linear relationships between inputs and outputs improves the network's generalization to a variety of datasets, and this is one of their main advantages. This flexibility is handy for tasks like image recognition, where objects are identified based on complex relationships between pixels. Because fully connected layers have complete connectivity, which helps the neural network identify subtle dependencies and extract high-level representations from the input data, these layers are essential for improving neural networks' overall performance and learning ability in various applications.

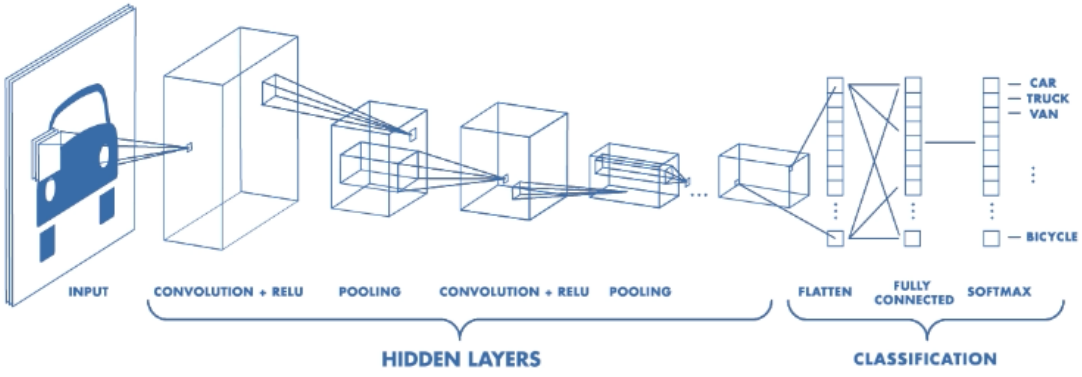


Figure 1.3: CNN pipeline

Applications

Convolutional Neural Networks are used for many purposes concerning computer vision tasks. The common denominator is recognizing and localizing objects within an image.

- **Object Detection:** the most widely used, it is the identification and classification of specific objects drawing a bounding box around the object of interest and providing a label or class for each bounding box.
- **Semantic Segmentation:** classification of every pixel of an image into global classes like "road," "sky," "vegetation," or "people". It does not detect specific instances of objects but classifies every pixel of the image into a category.
- **Instance Segmentation:** it combines object detection and semantic segmentation techniques. It deals with the identification and separation of individual objects, including detecting the boundaries of each object and assigning a unique label to each object.
- **Pose Estimation:** estimation of the position and orientation of an object.
- **Object Tracking:** computation of the movement of a specific object across frames in a video sequence.

1.4 Object Detection

Object detection was realized through a feature extraction algorithm in the past. For example, SIFT [7] (Scale-Invariant Feature Transform) identifies image key points on local gradients and generates descriptors that are invariant to transformations like scaling and rotation, or SURF [8] (Speeded-Up Robust Features) accelerates the computation of features compared to SIFT and maintains robustness to scale and rotation changes, making it suitable for real-time applications. Other important algorithms were Histogram of Oriented Gradients (HoG) [9], which captures local gradient information in image regions, and Support Vector Machine (SVM) [10], which was employed for classification and regression tasks. It finds an optimal hyperplane to separate data into different classes while maximizing the margin. Then, the advent of deep learning revolutionized the landscape, with CNNs outperforming traditional methods by autonomously acquiring intricate hierarchical features from data. In stark contrast to manually crafted features, CNNs dynamically adjust and enhance their internal representations through training, capturing shaded

patterns and contextual nuances crucial for precise object localization and classification. Consequently, handcrafted algorithms have been increasingly reduced in favor of higher-performance object detectors designed to identify and outline objects in intricate, real-world scenarios accurately. This innovative approach seamlessly integrates the potency of two pivotal computer vision tasks: localization (pinpointing object locations) and classification (assigning accurate labels or categories to each identified object).

Before delving into the latest advancements in the field of metrics, it's important to establish a clear understanding of fundamental concepts related to metrics.

1.4.1 Key Metrics

Object detection metrics play an important role in assessing the efficacy of models in object detection tasks, providing a means for comparison and benchmarking across multiple detection systems. They offer a comprehensive framework for evaluating the model's accuracy, recall, and precision, essential for understanding its overall performance.

Name	Function	Description
Intersection over Union	$\text{IoU} = \frac{\text{A Overlap}}{\text{A Union}}$	Literally the intersection over union of detection bounding box and ground truth bounding box
True Positives	(TP) IoU $>$ Threshold	Detections with IoU greater than a threshold
False Positives	(FP) IoU $<$ Threshold	Detections with IoU less than a threshold or detected more than once
False Negatives	(FN) IoU $<$ Threshold	Number of objects not detected or detected with IoU less than threshold
Precision	$\text{P} = \frac{\text{TP}}{\text{TP+FP}}$	Measures the accuracy of true positives out of all positives
Recall	$\text{R} = \frac{\text{TP}}{\text{TP+FN}}$	Measure how many true positives were found
F1 Score	$\text{F1} = 2 \times \frac{\text{P}\times\text{R}}{\text{P+R}}$	Combines Precision and Recall into a single value
Average Precision	$AP = \int_{r=0}^{1} p(r)dr$	Area under the Precision-Recall curve
Mean Average Precision	$mAP = \frac{1}{n}\sum_{k=1}^{n} AP_k$	Average of the AP calculated for all the classes, mAP@0.5 means it is calculated with IoU threshold of 0.5

Table 1.1: Main metrics for analysing a model

These metrics are particularly important because they allow us to make an accurate comparison of the performance of this case study. In particular, within the analyzed context, the two main metrics will be the mAP and the Uncertainty metric; the latter will be discussed later.

1.5 Types of Object Detectors

As written before, the object detection task aims to draw multiple bounding boxes of objects in a given image, and CNNs are used to achieve the goal. Generally, these object detectors can be classified into two categories: single-stage and multi-stage models. The former performs object detection in a single step, directly predicting bounding boxes and class labels from the input image. The latter, on the other hand, involves multiple steps, including region proposal and object classification. The result is the same for both, but the process from the starting image to the predicted bounding boxes differs, even if some features are the same. Only two-stage object detectors will be analyzed within the multi-stage category, as they are the most common.

1.5.1 Two-stage Object Detector

The two-stage object detectors divides the whole process (and the model itself) into some steps:

- **Region proposal network (RPN):** responsible for generating potential regions of interest (RoIs) within an image. It achieves this by scanning the entire image and proposing candidate bounding boxes that might contain objects. Then, the RoIs are passed to the next stage.
- **RoI Pooling:** converts variable-sized RoIs into fixed-sized feature maps.
- **Classification network:** RoIs are fed into the object detection network that further refines the bounding boxes and performs object classification, determining the class labels for the objects within the refined regions. The finale object detection network is crucial for accurately identifying and classifying objects within the proposed regions finalizing the object detection process.

Region Proposal Network (RPN)

In the first step, the input image goes through the CNN, which applies convolutional operations and gives the feature maps as output. Then, a sliding window is run through the feature maps obtained before. The sliding window size is n*n; for each sliding window, a particular set of anchors (which are predefined boxes) are generated but with three different aspect ratios (e.g., 1:1, 1:2, 2:1) and different scales (e.g., 3). As a consequence, there are a total of 9 proposals for each image pixel. Therefore, imagining an image of a square shape 640 would result in 640x640x9 proposals. However, there may be many boxes that do not have any object, so

the model needs to learn which anchor box could have an object within. The RPN predicts two crucial pieces of information for each anchor box: the probability of the box containing an object (objectness score) and the adjustments needed to transform the anchor box into a more precise and accurate proposal (bounding box regression offsets). These predictions are made based on the features extracted from the sliding window positions: a model Bounding Box Classifier calculates the IoU score ground truth box with anchor boxes and classifies the anchor box in either foreground or background with a certain objectness score (or probability), while a Bounding Box Regression layer learns the offsets for x,y,w,h values concerning ground truth box for the anchor box that has been classified as foreground, where (x,y) is the center of the box, w and h are width and height. RPN does not care what final class is predicted; it only cares whether it is a foreground object or background. Finally, after the prediction of both objectness and regression, it computes the loss function and tries to minimize it by updating the model.

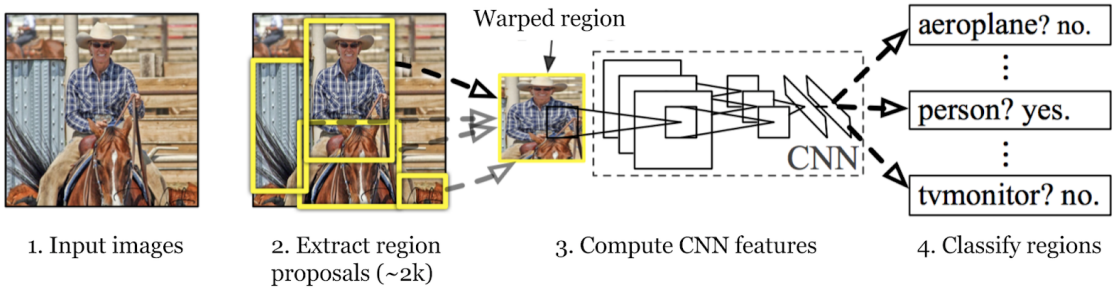


Figure 1.4: CNN pipeline

RoI Pooling

Before moving on to the classification part, the previously calculated ROIs must be redrawn to be analyzed by the classifier. RoI Pooling transforms these variable-sized regions into a uniform spatial extent, allowing seamless integration with fully connected layers for object classification and bounding box regression. By providing structured and uniform feature representations, RoI Pooling enables accurate object categorization and precise localization, essential for effective object detection in images (see Fig. 1.4).

Classification network

The last step deals with classification. This network typically consists of several convolutional layers followed by some fully connected layers. It analyses the features from the previous layer and predicts the probability distribution over predefined

object categories. It usually employs the softmax activation function and assigns a class label to each RoI, indicating the type of object it may contain (dog, car, person).

1.5.2 State of the Art - Two stage object detectors

Two-stage object detectors are a prominent class of models widely employed in computer vision for robust and accurate object detection. The Region-Based Convolutional Neural Network (R-CNN) family is among the most prominent groups. It introduced a novel paradigm by first suggesting regions of interest and then further classifying and optimizing them, and its successors, such as Fast R-CNN and Faster R-CNN, streamlined this approach, enhancing both speed and accuracy; as a result, it can be said that in the last decade, this family of object detectors has led the way. The Two-Stage Detector landscape also includes models like Mask R-CNN [11], which extends the capabilities to instance segmentation, providing detailed pixel-level information. Let us introduce R-CNN (2014), Fast-RCNN (2015) and Faster-RCNN (2015).

R-CNN

To bypass the problem of selecting a huge number of regions, Ross Girshick et al. [12] proposed a method where selective search [13] is used to extract just 2000 regions from the image and he called them region proposals. Therefore, instead of trying to classify a huge number of regions, it is possible to work with just 2000 regions. The selective search algorithm operates by employing a diverse set of low-level cues to segment the image into numerous small regions. These regions are then merged hierarchically based on color, texture, and shape similarities, resulting in progressively larger and more complex regions. By taking a hierarchical approach, the algorithm can identify objects at multiple scales and levels of granularity. Its ability to capture various object sizes and shapes enables the model to handle objects with different visual features.
The 2000 candidate region proposals are warped into a square and fed into a convolutional neural network that produces a 4096-dimensional feature vector as output. The CNN acts as a feature extractor, and the dense output layer consists of the features extracted from the image; these are inserted into a Support Vector Machine (SVM), a supervised machine learning algorithm used for classification and regression tasks, to classify the presence of the object within the proposed candidate region. The algorithm also predicts four values, which are offset values, to increase the precision of the bounding boxes (see Fig 1.5). In the case of a region proposal, for

instance, the algorithm would have predicted the presence of a person even though their face might have been halved. As a result, the offset values aid in modifying the region proposal's bounding box. The R-CNN model was powerful, but it still takes a huge amount of time to train the network as one would have to classify 2000 region proposals per image; as a consequence, it cannot be implemented in real-time as it takes around 47 seconds for each test image; furthermore, the selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.

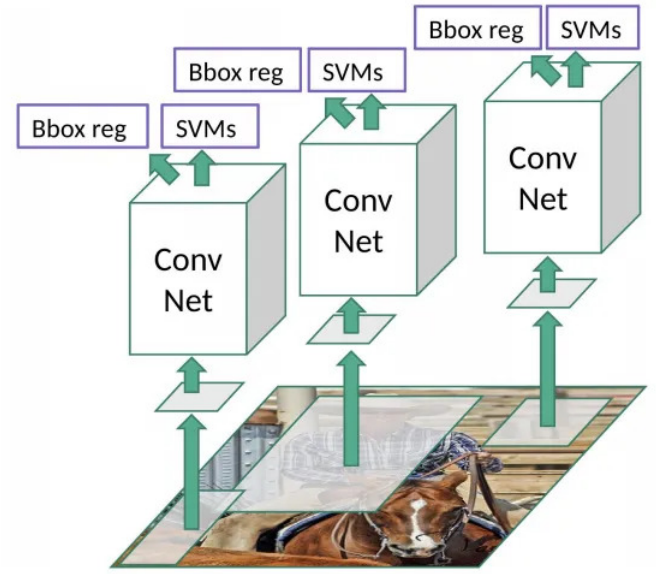


Figure 1.5: R-CNN

Fast R-CNN

The same author of the previous model solved some of the drawbacks of R-CNN to build a faster object detection algorithm, and it was called Fast R-CNN [14]. The process is similar to the R-CNN algorithm, but instead of feeding the region proposals to the CNN, convolutional operations are applied first to generate a convolutional feature map. Then, region proposals are generated from the map, warped into squares, and reshaped by using a RoI pooling layer into a fixed size to feed it into a fully connected layer. From the RoI feature vector, it is used a softmax layer to predict the class of the proposed region and the offset values for the bounding boxes, too.

Fast R-CNN is quicker than R-CNN because the model does not have to process thousands of region proposals every time. Instead, the convolution operation is done only once per image, and a feature map is generated from it. When looking at performance during testing time, including region proposals slows down the algorithm

significantly when compared to not using region proposals. Therefore, region proposals become bottlenecks in the Fast R-CNN algorithm, affecting its performance.

Faster R-CNN

Both of the above algorithms use selective search to find out the region proposals. The algorithm is a slow and time-consuming process affecting the network's performance. Therefore, Shaoqing Ren et al. [15] developed an object detection algorithm that deletes the selective search algorithm and lets the network learn the region proposals. The picture is fed into a convolutional network, which generates a convolutional feature map, much like Fast R-CNN does. A different network is employed to forecast the region proposals rather than utilizing a selective search technique on the feature map to find the region proposals. After using an ROI pooling layer to reshape the projected region proposals, the image inside the proposed region is classified, and the bounding box offset values are predicted.
Faster R-CNN is much faster than its predecessors. In addition, it can even be used for real-time object detection.

1.5.3 One-stage Object Detector

Regions are used by all of the earlier object detection techniques to locate the object within the picture. The network does not view the entire image. Rather, regions of the picture with a high chance of holding the object. The object detection technique known as YOLO, or You Only Look Once [16], differs significantly from the region-based algorithms previously mentioned. In YOLO, a single neural network predicts the bounding boxes and the class probabilities for these boxes. Other experiments were made in the same year, such as SSD [17], but YOLO led the revolution by far and set itself up as the state-of-the-art for single-stage models and object detectors in general.

YOLO

YOLO, or You Only Look Once, is an object detection model brought to us in 2016, and it revolutionized the CNN world by introducing a real-time object detection system. Unlike traditional methods, YOLO processes images in one pass, enabling faster and more accurate detection. This breakthrough streamlined complex tasks, making it a cornerstone in computer vision and deep learning applications. Using a single neural network, YOLO synthesizes a previously multi-step process by categorizing and bounding box prediction for identified objects. Because of this, it

can operate considerably more quickly and is highly optimized for detection performance compared to using two different neural networks for object detection and classification. It accomplishes this by reusing conventional image classifiers for object bounding box identification regression. Year after year, YOLO evolved from the first version to the eighth [18], becoming more precise and faster. However, although the subsequent iterations feature numerous improvements, the basic idea behind the architecture stays the same.

The YOLOv1 model, also called YOLO, is a highly suitable option for applications requiring real-time detection, as it can achieve a processing rate of 45 frames per second, surpassing the temporal demands of real-time object detection. It is 90 times faster than Fast R-CNN and 2.5 times faster than the quickest Faster R-CNN model; on the other hand, it sacrifices a little precision, which nevertheless remains high and overall accurate (see Fig 1.6). It can record the context of identified items since it looks at the entire image at once and only once, hence the term You Only Look Once. Additionally, YOLO can generalize the representations of various objects, making it more applicable to various new environments.

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [31]	2007	16.0	100
30Hz DPM [31]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [38]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[28]	2007+2012	73.2	7
Faster R-CNN ZF [28]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

Figure 1.6: YOLO performance compared to other detectors in 2016

YOLO is based on the idea of segmenting an image into smaller images. The image is split into a square grid of dimensions S×S. The cell in which the center of an object resides is responsible for detecting that object and the associated confidence score. With this architecture, the model predicts B bounding boxes where B=2 by default; as a result, the final number of bounding boxes is SxSxB. The confidence score will range from 0.0 to 1.0; it tells how confident the model is that the bounding

box contains an object, with 0.0 representing the lowest confidence level and 1.0 the maximum. It is computed as follows:

$$C = P_r(object) \times IOU_{truthpred} \tag{1.3}$$

Where $P_r(object)$ is 1 if the object exists, and 0 otherwise, when an object is present, the confidence score equals the IOU between the ground truth and the predicted bounding box. Each of these bounding boxes is made up of 5 numbers: the x center position, the y center position, the width, the height, and the confidence. In addition to outputting bounding boxes and confidence scores, each cell predicts the class of the object. This class prediction is represented by a vector with length C, the number of classes in the dataset. Then, let´s focus on the loss function of YOLO as shown below.

$$\begin{aligned}
&\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
&+\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} (C_i - \hat{C}_i)^2 \\
&+\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{noobj} (C_i - \hat{C}_i)^2 \\
&+ \sum_{i=0}^{S^2} 1_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2
\end{aligned} \tag{1.4}$$

At first glance, equation 1.4 might look lengthy and tricky, but it is not. The first part of the equation computes the loss between the predicted bounding boxes midpoints $(\hat{x}, \hat{y})$ and the ground truth bounding box midpoints (x, y). It is calculated for all SxS cells, and only the one with the highest IOU is considered in the equation. The weight $\lambda_{coord} = 5$ ensures that bounding box predictions are as close as possible to the target. Lastly, an identity function 1_{ij}^{obj} denotes that the jth bounding box predictor in cell i is responsible for that prediction. Thus, it will be 1 if the target object exists and 0 otherwise. The second part is similar to the first, but it is computed the loss between the predicted bounding box width and height $(\hat{w}, \hat{h})$ and the target width and height (w, h). The square root of values is taken because sum squared loss equally weighs errors in large and small scale boxes. In the third part, the loss between the predicted confidence score of the bound box $\hat{C}$ and the target

confidence score C is calculated. Here, the target confidence score C equals the IOU between the predicted bounding box and the target. If no object exists in the grid cell i, then $1_{ij}^{noobj} = 1$ and $1_{ij}^{obj} = 0$ will make the third part of the equation zero. The loss is computed between the box with a higher IOU, and the target C, which is 0 since the confidence for the cell with no object must be 0. The formula is weighed with $\lambda_{noobj} = 0.5$ since there can be many grid cells with no objects, and it´s not wanted this term to overpower the gradients for the cells containing objects. As highlighted in the paper, this could lead to model instability and be harder to optimize. Finally, for every cell, $1_{i}^{obj} = 1$ if an object appears in cell i, the loss is computed for every class of the dataset. Here, $\hat{p}(c)$ is the predicted conditional class probability, while $p(c)$ is the target conditional class probability.

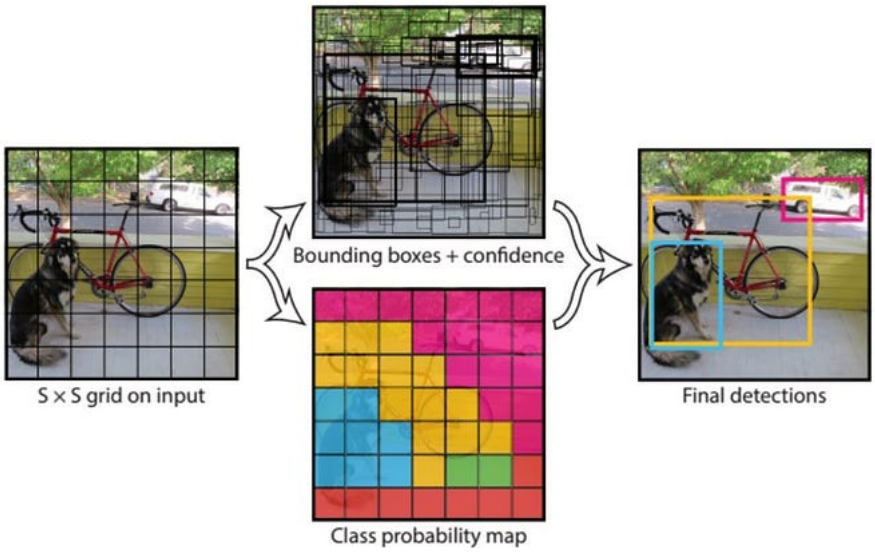


Figure 1.7: How YOLO works

In summary, CNNs and grid-based prediction work together to enable YOLO's groundbreaking ability to perform comprehensive object recognition in a single step. YOLO revolutionized computer vision by dividing the input image into a grid and employing CNNs to forecast bounding boxes and class probabilities simultaneously (see Fig 1.7). Nevertheless, there are still limitations. First, with a grid with $S = 7$ the model can detect a maximum of 49 objects; second, since each grid cell predicts only two boxes and can only have one class, this limits the number of nearby objects that YOLO can predict, especially the small objects that appear in groups. Third, the model struggles to detect smaller objects. Finally, the authors believe that the primary source of error is incorrect localizations since the loss function treats errors the same in small and large bounding boxes.

1.6 YOLOv7: paper review

Since YOLO is a technology without a single owner, its developers have changed over time. In certain cases, such as the most recent version, YOLO versions have been released without a corresponding scientific paper. In contrast, in other cases, companies have been involved in developing that version. For example, the latest YOLO version has been developed by Ultralytics, whose stated purpose is: "We're here to make AI easy. Our purpose is to make model development accessible, efficient to train, and easy to deploy".
Within the work of this thesis, YOLOv7 [19] was used, which was released in 2022 and marked a turning point that became state-of-the-art in the field of single-stage object detectors and object detectors in general. As described in the official paper: "YOLOv7 surpasses all known object detectors in both speed and accuracy in the range from 5 FPS to 160 FPS and has the highest accuracy 56.8% AP among all known real-time object detectors with 30 FPS or higher on GPU V100".

1.6.1 Notable Features within YOLOv7

Extended-Efficient Layer Aggregation Network (E-ELAN)

E-ELAN extends the concept of ELAN [20]; the scope of ELAN is to solve the problem that the convergence of the deep model will gradually deteriorate when executing model scaling. After the network reaches a certain critical depth, the accuracy gain will decrease if computational blocks are kept stacked, and its convergence will start to worsen, making the total accuracy worse than shallow networks. ELAN authors are trying to solve this problem by controlling the shortest gradient path so that it does not quickly become longer. In YOLOv7, the authors propose Extended-ELAN (E-ELAN). At E-ELAN, the gradient flow path remains the same as ELAN; only the cardinality increases through the use of group convolution.

Compounding Model Scaling

Model scaling refers to width (channel) or depth (number of layers) scaling. The authors found that when depth scaling was done in concatenation-based models (computation block and transition block), it increased input width in subsequent transition blocks. A computation block is a specific component or unit within the network architecture that performs a set of computational operations, such as convolutional layers, activation functions, pooling layers, and normalization layers. A transition block is a component or a layer that is commonly used in architecture like

DenseNet; its purpose is to reduce the spatial dimensions (width and height) of the feature maps while maintaining or controlling the number of channels. The scaling problem exists because the purpose of the transition layer is to make a reduction, either channel reduction or dimension reduction, of the feature map. So, to solve this problem, they did depth scaling for only the computational block and width scaling for the transition block.

Bag-of-freebies (BoF)

BoF refers to techniques that improve performance without increasing the training cost. It involves changing the training strategy or increasing the training cost without affecting the inference cost, meaning it does not affect the frames per second during testing. Empirical results [21] have shown that these freebies can improve object detection precision by up to 5% compared to state-of-the-art baselines.

Re-parameterization Convolution (RepConv)

Re-parameterization Convolution means different architectures can be mutually converted through the equivalent transformation of parameters. YOLOv7 applied a similar RepConv as RepVGG except for the identity connection of RepVGG [22]. RepConv combines 3×3 convolutions, 1×1 convolution, and identity connection in one convolutional layer. After conducting a reparametrization trial, the authors discovered that the concatenation of DenseNet [23] and the residual in ResNet [24] are destroyed by the identity connection in RepConv; as a result, the feature maps lose their varied slopes. Consequently, the authors suggested RepConvN, a RepConv without an identity connection.

Varying granularity of label assignment

YOLOv7 implements two different types of head:

- **Lead head:** responsible for final prediction.
- **Auxiliary head:** assists training in previous layers with assistant loss.

These heads work with the concept of label assigner. A label assigner considers network prediction results together with ground truth, and instead of just hard prediction results, it assigns soft labels. According to Galstyan and Cohen [25], a hard label is a label assigned to a member of a class where membership is binary: either this element is a member of the class (has the label), or it is not, while a soft label has a score (probability or likelihood) attached to it, this implies that an element

can be a member of multiple classes. YOLOv7 proposes two types of label assigners.

Lead head guided label assigner. This is computed using the ground truth and the lead head's prediction result, and the optimization process creates a soft label. The target training model for the auxiliary head and lead head will be this set of soft labels. The authors say: "The reason to do this is because lead head has a relatively strong learning capability, so the soft label generated from it should be more representative of the distribution and correlation between the source data and the target". Moreover, this kind of learning could be regarded as generalized residual learning. The lead head will be better able to concentrate on acquiring residual information that has yet to be taught if the shallower auxiliary head can study the knowledge the lead head has already learned directly.

Coarse (auxiliary) to fine (lead) label assigner.
This also uses the predicted result of the lead head and the ground truth to generate a soft label. However, in the process, YOLOv7 generates two sets of soft labels: coarse and fine. The first "is generated by allowing more grids to be treated as a positive target by relaxing the constraints of the positive sample assignment process", and the second "is the same as the soft label generated by lead head guided label assigner". It will focus on enhancing the auxiliary head's recall in the object detection task since its learning capacity is lower than that of a lead head. This will help to ensure that the necessary knowledge is not forgotten.

1.6.2 Github repository

The authors provided a full Python implementation packed in a GitHub repository [26]. YOLOv7 is built using PyTorch[27]. PyTorch is a high-performance deep-learning Python library developed by MetaAI, and it provides a flexible and efficient framework for training and deploying deep-learning models, focusing on ease of use and fast prototyping. Users can find short tutorials, scripts, and libraries in the repository to help simplify YOLO development. The main scripts are those for training, testing, and inference of the model; they are respectively *train.py*, *test.py* and *detect.py*. There are numerous configuration files, and the developers equipped the technology with different types of variation from the basic model and, consequently, different configuration files depending on the user's needs. The main model versions are YOLOv7, the basic model optimized for ordinary GPU computing, YOLOv7-tiny, which is optimized for edge-GPU and more lightweight to run ML on mobile computing devices or distributed edge servers and devices, and YOLOv7-W6, a optimized model for cloud GPU computing. Each model has

its own architecture, which is described by a .yaml file that maps the model layers into a single file. Furthermore, there are two other important configuration files: dataset and hyperparameter. The first contains the dataset path for train, validation, and test dataset; the second consists of a set of hyperparameters useful during training time, such as augmentation parameters and optimizer attributes; details regarding the hyperparameters mentioned above will be discussed and shown later. All YOLOv7 pretrained models have trained with the COCO (Common Objects in Context) dataset, which is a widely used large-scale image dataset and contains high-quality images with complex scenes. Each image is annotated with object bounding boxes, object categories, and segmentation masks for a total of 80 different classes.

1.7 Tools for Optimizing Object Detection

Object detectors learn from input ground truth and, through their architecture, how to generalize detected features to recognize classes in different contexts. Model knowledge is complex; the neural network does not just have to learn to detect dataset input images but must be sturdy enough to adapt to context-switching. Real-world scenarios constantly have variable contexts, including illumination, object position, and image resolution; consequently, models must generalize ground truth features to catch them in all situations. When this does not happen, we talk about underfitted or overfitted models. Insufficient model complexity results in underfitting; it is frequently the case that the model needs to be provided with more data to learn from, though there may also be other reasons. Consequently, the model performs badly on both the training dataset and fresh data points since it cannot identify the generalizable patterns in the data. Conversely, overfitting happens when a model is too complex for the data it represents. There are several possible causes for this, but the most frequent one is that the model needs to learn more from the data. This leads to the model not learning generalizable patterns but memorizing the training data. Consequently, the model exhibits strong performance on the training dataset but struggles to generalize to new data. Underfitting and overfitting are two of the main adversaries for deep learning models; over time, more and more techniques have emerged and been refined to go against these trends. Below, some of the most important will be analyzed.

Data Augmentation

The process of creating additional data points from preexisting data in order to increase the amount of information artificially is known as data augmentation. Amplifying the dataset can involve making small changes to the data. Applying aug-

mentations broadens the dataset's diversity and exposes the model to a greater range of data changes. This enhances the model's overall performance and robustness by helping it generalize to new, unseen data. When a limited training dataset is available, data augmentation comes in handy since it gives the model access to a wider range of instances, lowering the likelihood of overfitting and increasing the model's accuracy in practical applications. Common data augmentation techniques are rotations, flips, scaling cropping, changes in brightness and contrast, and adding noise. Furthermore, YOLOv7 adds mosaic [28] and mix-up [29] augmentation. The first is the combination of 4 images into one, stitching them together, and then taking a random cutout of the stitched images; the latter basically averages two images together based on a weighting parameter.

Dropout Regularization

Dropout regularization refers to the process of randomly ignoring certain neurons in a layer during training. For each iteration, a neuron has a probability of p to be dropped; as a result, it does not send any output signal to the next layer (see Fig 1.8). It was found [30] that models with dropout had fewer classification errors than the identical models without dropout because it helps minimize overfitting on the training set, which accounts for the decreased error.

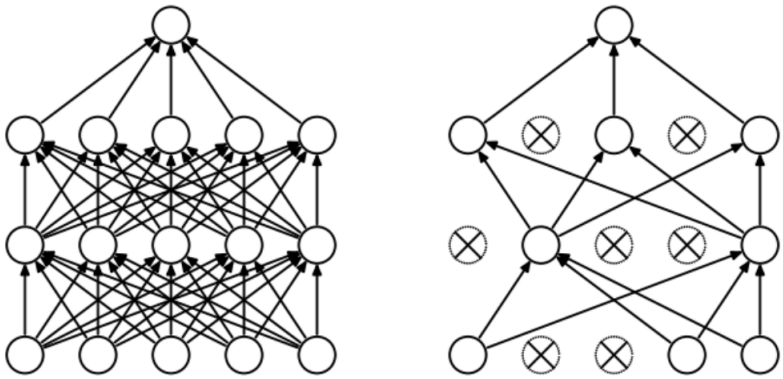


Figure 1.8: Dropout regularization, neurons are dropped with a probability p

The model picks up statistical noise in the overfitting issue. More specifically, given all the units (neurons), the primary goal of training is to reduce the loss function. Therefore, a unit may adjust in overfitting circumstances to correct the other units' errors. Because of the inability of these complicated co-adaptations to generalize to the unknown dataset, they result in complex co-adaptations, which in turn generate the overfitting issue. Using dropout stops these units from correcting the errors made by other units, prohibiting co-adaptation because a unit's presence is very unpredictable in every iteration. Thus, it compels the layers to adopt a probabilis-

tic approach and assume some degree of responsibility for the input by arbitrarily removing a few units. This ensures that the model is getting generalized and hence reducing the overfitting problem.

Batch Normalization

Batch normalization is used in Deep Learning to increase the efficiency of neural networks. A neural network is trained by varying the model's weights according to the input data in order to produce precise predictions. However, as a network gains experience, the distribution of values in the output of each layer may shift, which complicates the training process. Batch normalization solves this issue by normalizing each layer's output. Normalising values involves bringing their mean and standard deviation as near to zero as possible. Small groups of data, or mini-batches, are used for the normalization process. Because of memory and computational resource constraints, processing all the data at once is impractical and calls for the use of mini-batches.

Learning Rate Scheduling

Reducing the learning rate as the training progresses is useful when training a deep Neural Network. It guarantees a stable convergence and avoids oscillations around the global minimum by permitting incremental steps toward the ideal solution as training advances. Reducing the learning rate gradually also aids in a more in-depth exploration of the loss surface, which may assist in avoiding local minima and producing a more accurate model. Additionally, it reduces the risk of overfitting, which improves the model's capacity for generalization by keeping it from fitting the training data too closely. This can be done by using pre-defined learning rate schedules, such as constant learning rates, time-based decay, and step decay. For example, YOLOv7 implements three different learning rates regulated by a cosine decay function (see Fig 1.9) with an initial warm-up phase, according to Tong He et al. [31]. Learning rates condition object loss, box loss, and class loss, respectively.

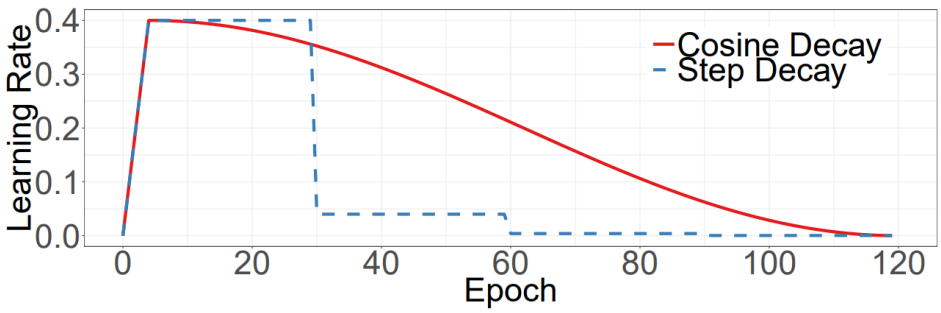


Figure 1.9: Cosine decay function

Transfer Learning

Transfer learning is a potent machine learning technique that enables developers to apply the knowledge acquired from one activity to another that is related. It is especially helpful when there is a lot of labeled data for a similar source task but little labeled data for the target activity. The main idea behind transfer learning is that models trained on large and diverse datasets, such as COCO or ImageNet, have learned general features that are applicable to a wide range of tasks. Everyone can save a lot of time and computing resources when training new models from scratch by starting with these pretrained models.

Hard Examples Mining

This method is used in machine learning to increase the efficacy and efficiency of the learning process, especially while classifiers are being trained. The term hard examples refers to instances in the training set that are frequently misclassified. These examples are frequently anomalies or situations close to the line dividing two classes. Usually, a common dataset contains an overwhelming number of easy examples and a small number of hard ones; as a result, a model is trained almost entirely on easy and unambiguous examples. A selection of these hard examples can make training more effective and efficient. One of the most common techniques for hard example mining is Online Hard Example Mining (OHEM); this method concentrates on locating and giving more weight to the more challenging training examples—those that are incorrectly classified or have low prediction confidence. According to Shrivastava et al. [32], there are two hard examples mining algorithms. The first is used when optimizing SVMs. In this case, the training algorithm maintains a working set of examples and alternates between training an SVM to converge on the working set and updating the working set by removing some examples and adding others according to a specific rule. This rule removes examples that are easy and adds new examples that are hard according to the current model´s margin. The working set is usually a small subset of the entire training set. The second method is used for non-SVMs. It starts with a dataset of positive examples and a random set of negative examples. The machine learning model is then trained to converge on that dataset and subsequently applied to a larger dataset to harvest false positives. The false positives are added to the training set, and the model is trained again. This process is usually iterated only once and has no convergence proofs.

Chapter 2

Object Detectors in the real world: A complex paradigm

Object detection algorithms have advanced significantly in the quickly changing field of computer vision, becoming more reliable and efficient, but developing a neural network for practical purposes is still challenging. Real-world scenarios conceal great complexity because of different factors that influence the views of the scene. A sunny early summer´s day influences the context differently than a winter's night and every year, new cars with different shapes and colors are released. As a result, finding an identical situation in the real world is impossible. Despite the strides made in object detection, tailored approaches are still needed due to the complexities of real-world settings, and sometimes, for every novel measurement campaign, a re-training is necessary. This calls for applying methods and tools that improve the neural network's performance and make it even more capable of handling the complexity of many real-world scenarios. Numerous researchers push the frontiers of technology beyond the limits every day, trying to improve the precision and adaptability of neural networks across different scenarios. This thesis work will examine the main tools for optimizing neural network performance in training, and these tools will be applied, sometimes extended, to YOLOv7 under the same initial condition. This approach allows for a thorough study of their effectiveness and facilitates direct comparisons. The tools in focus will be discussed individually in the following sections, while the same initial condition will be analyzed in the next chapter.

2.1 YOLOv7 performance

"All the YOLOv7 models surpass the previous object detector in speed and accuracy and has the highest accuracy 56.8% AP among all known real-time object detectors with 30 FPS or higher", so begins the official YOLOv7 paper. Fig 2.1 is taken from the article as well, it shows the aforementioned results and they are mostly grouped together according to a range of parameters for a particular set of models.

Model	#Param.	FLOPs	Size	AP^{val}	AP^{val}_{50}	AP^{val}_{75}	AP^{val}_{S}	AP^{val}_{M}	AP^{val}_{L}
YOLOv4 [3]	64.4M	142.8G	640	49.7%	68.2%	54.3%	32.9%	54.8%	63.7%
YOLOR-u5 (r6.1) [81]	46.5M	109.1G	640	50.2%	68.7%	54.6%	33.2%	55.5%	63.7%
YOLOv4-CSP [79]	52.9M	120.4G	640	50.3%	68.6%	54.9%	34.2%	55.6%	65.1%
YOLOR-CSP [81]	52.9M	120.4G	640	50.8%	69.5%	55.3%	33.7%	56.0%	65.4%
YOLOv7	36.9M	104.7G	640	**51.2%**	**69.7%**	**55.5%**	**35.2%**	**56.0%**	**66.7%**
improvement	-43%	-15%	-	+0.4	+0.2	+0.2	+1.5	=	+1.3
YOLOR-CSP-X [81]	96.9M	226.8G	640	52.7%	**71.3%**	57.4%	36.3%	57.5%	68.3%
YOLOv7-X	71.3M	189.9G	640	**52.9%**	71.1%	**57.5%**	**36.9%**	**57.7%**	**68.6%**
improvement	-36%	-19%	-	+0.2	-0.2	+0.1	+0.6	+0.2	+0.3
YOLOv4-tiny [79]	6.1	6.9	416	24.9%	42.1%	25.7%	8.7%	28.4%	39.2%
YOLOv7-tiny	6.2	5.8	416	**35.2%**	**52.8%**	**37.3%**	**15.7%**	**38.0%**	**53.4%**
improvement	+2%	-19%	-	+10.3	+10.7	+11.6	+7.0	+9.6	+14.2
YOLOv4-tiny-3l [79]	8.7	5.2	320	30.8%	47.3%	32.2%	**10.9%**	31.9%	51.5%
YOLOv7-tiny	6.2	3.5	320	**30.8%**	**47.3%**	**32.2%**	10.0%	**31.9%**	**52.2%**
improvement	-39%	-49%	-	=	=	=	-0.9	=	+0.7
YOLOR-E6 [81]	115.8M	683.2G	1280	55.7%	73.2%	60.7%	40.1%	**60.4%**	69.2%
YOLOv7-E6	97.2M	515.2G	1280	**55.9%**	**73.5%**	**61.1%**	**40.6%**	60.3%	**70.0%**
improvement	-19%	-33%	-	+0.2	+0.3	+0.4	+0.5	-0.1	+0.8
YOLOR-D6 [81]	151.7M	935.6G	1280	56.1%	73.9%	61.2%	**42.4%**	60.5%	69.9%
YOLOv7-D6	154.7M	806.8G	1280	56.3%	73.8%	61.4%	41.3%	60.6%	70.1%
YOLOv7-E6E	151.7M	843.2G	1280	**56.8%**	**74.4%**	**62.1%**	40.8%	**62.1%**	**70.6%**
improvement	=	-11%	-	+0.7	+0.5	+0.9	-1.6	+1.6	+0.7

Figure 2.1: YOLOv7 vs other baseline models

Starting with the YOLOv7-Tiny model, the smallest in the family with just over 6 million parameters. With a validation AP of 35.2%, it beats YOLOv4-Tiny models with similar parameters. The YOLOv7 normal model with almost 37 million parameters gives 51.2% AP. It surpasses variants of YOLOv4 and YOLOR, which have more parameters, easily. The larger models in the YOLO7 family are YOLOv7-X, YOLOv7-E6, YOLOv7-D6, and YOLOv7-E6E. All of these beat the respective YOLOR models having either a similar or less number of parameters and giving APs of 52.9%, 55.9%, 56.3%, and 56.8%, respectively. Another noteworthy fact, comparing the validation APs with that of the YOLOv5 and YOLOv7 models, which have parameters in the same range, it is pretty clear that YOLOv7 also beats all of the YOLOv5 models. All things considered, YOLOv7 and variants are a powerful set of models, they surpassed all the previous models becoming the new state-of-the-art in object detection.

2.2 Tools for optimizing training performance

The tools that will be used within YOLOv7 are: data augmentation, dropout regularization and hard samples mining. Some of the research results of recent years are given.

Data Augmentation

Convolutional neural networks' development and functionality depend heavily on data augmentation. It is a method that entails applying changes like cropping, flipping, rotating, scaling, or altering the colour, brightness, or contrast to current training data to create new variations. This procedure's purpose is to reduce overfitting, broadens and diversifies the training set, and strengthens the model's resistance to various inputs, furthermore, it facilitates the development of neural networks that have small datasets availability. Its main power lies in creating new useful data without increasing the computational demand to train the model, as a result, the same number of images has a greater expressiveness and the model is able to get a better training. Several researches [33] have been conducted on the efficacy of data augmentation approaches, yielding varying results at times, but generally indicating the critical importance and remarkable use of these methods. Every investigation is different from others because of different conditions, especially the use of a certain type of model and a specific dataset. For example, Perez and Wang [34] show an average increase of about 6% in the case of binary classifications, while Taylor and Nitschke [35] argue that geometric transformations, first and foremost cropping, which increases the probability that the highest output of the network corresponds to the right label by about 13 percentage points, are mainly functional. Additionally, Nanni et al. [36] use data augmentation techniques on more specialised datasets, such as datasets containing virus data (VIT), tree barks (BARK), gravitational wave images (GRAV) or paintings belonging to various artistic movements (POR). Although the accuracies were already high to begin with, excluding the results on the GRAV dataset, as these are already over 90% accurate, an average increase of 3.72% is obtained.

Dropout Regularization

Dropout [30] is not necessary in a neural network, or at least it is not as fundamental as data augmentation mechanisms are; however, its importance grows when there are large and deep neural networks with a huge number of parameters. In this context, dropout layers prevent units from co-adapting too much, which significantly reduces

overfitting and gives major improvements over other regularization methods. It's worth noting that while dropout is particularly beneficial in complex networks, its efficacy may vary depending on the specific architecture and dataset characteristics. Srivastava et al. showed that adding dropout layers to a neural network model reduces the error rate by about 4% on the CIFAR-10 dataset and about 6% on the CIFAR-100 dataset. This clearly illustrates how the usefulness of these kinds of layers grows with dataset size. The same outcome was achieved by Park and Kwak [37], their study adopts different types of dropouts such as basic dropout, spatial dropout and stochastic dropout, reaching a reduction of about 13% and 3% with respectively CIFAR-10 and CIFAR-100.

Hard Example Mining

Datasets contain an overwhelming number of easy examples and a small number of hard examples, Online Hard Example Mining (OHEM) [32] selects images with high uncertainty to reach a more difficult and challenging training. Shrivastava et al. demonstrate an improvement by 2.7% with Fast R-CNN on VOC 2007 dataset and 1.9% on COCO dataset that is much bigger than the other one while other experiments, such as Köksal et al. [38], perform an HEM approach to single shot object detectors, mixing two different techniques, Loss Rank Mining and Focal Loss, reaching an increase of mAP by 3% compared to using the original loss function and around 1-2% compared to using the hard-mining methods (LRM or FL) individually on 2021 Anti-UAV Challenge Dataset.

2.3 Application of the main optimization tools to a single-stage neural network

The previously stated approaches (Data Augmentation, Dropout Regularization, and Hard Examples Mining) serve as cornerstones of neural network training optimization, each specifically designed to tackle unique problems encountered in various applications. Though powerful in their own right to support model robustness, these methods have historically been examined separately, each offering crucial new information to the larger field of machine learning. Nonetheless, this thesis aims to investigate and implement optimization techniques within the same context. Instead of just analyzing these techniques separately, a step forward is made by combining them and applying them to a complex and cutting-edge technology like YOLOv7. By integrating these optimization threads into YOLOv7, the aim is to investigate their interactions and assess their effectiveness on their own. The following research

represents a move onward in the scientific landscape not only because it allows the comparison of optimization systems that are usually stand-alone but also because, in the current context, some of the tools above applied to single-stage object detectors such as YOLO do not appear to be available, or are sometimes implemented, but their effectiveness is not known. Furthermore, when applied in ad hoc contexts, computer vision solutions often exhibit limited capabilities, not necessarily in terms of computational power but predominantly concerning data availability. The challenge arises from the impracticality of manually curating data sets comprising thousands of images, leading to scenarios where only a few hundred ground truth images are available. The other objective of this study is to investigate the performance of these tools in situations with limited image data sets. The focus is on whether building a robust object detector under resource-limited conditions is possible when only a few hundred images are available for training.

Chapter 3

Investigation on the Effectiveness of Tools for Optimizing YOLOv7 Training Performance

Now, the focus will shift to the execution and implementation of the research outlined in the preceding chapter. First of all, the frameworks and parameters used will be analyzed, and the entire situation of the model and dataset before the actual investigation will be presented, followed by an explanation of how the implementation was carried out at the code level. The results will then be highlighted in the following chapter.

3.1 Frameworks for Development

3.1.1 Robot Operating System (ROS)

ROS (Robot Operating System) [39] is an open-source middleware framework used to develop robot software. Despite the name, ROS is a collection of software tools and libraries that assist programmers in creating intricate robot applications rather than a real operating system. It offers services for package management, device drivers, hardware abstraction, and inter-processor communication, among other things. Fundamentally, ROS functions using a publisher-subscriber architecture, which enables smooth communication between different software components in a robot. The foundation of this communication model is topics, also called buses, across which nodes communicate with one another. Nodes in ROS can publish messages to a designated topic, producing information of any type, including motor commands, sensor readings, and status updates. It is the duty of publishers to compile this information into known message formats and distribute them on related

subjects. On the other hand, nodes can receive messages by subscribing to particular topics. These nodes, referred to as subscribers, have specific data interests and would like to be informed anytime new information relevant to those interests is released. As subscribers decipher the communications, they extract pertinent information for their responsibilities. A subscriber knows which type of information is being read from a particular topic, and it can also append a callback function triggered every time data is subscribed. ROS nodes, acting as black boxes, contain certain robotic system functions. As a result, different nodes could be responsible, for example, for processing sensor data, motor control, and decision-making algorithms. Every node is self-contained and autonomous; it has no knowledge of the inner workings of other nodes. This encapsulation facilitates modularity and streamlines the process of development. Therefore, developers won't have to worry about the complexities of other system components and can concentrate on building specialized nodes. In addition to the contribution it makes in the direction of modularity and application development workflow, ROS has other advantages. First, it promotes reusability by utilizing nodes in various robotic applications, thereby conserving time and resources. Then, collaborative development is encouraged because of its open-source nature, leading to a thriving community of contributors and a rich ecosystem of libraries and tools. Furthermore, ROS is very scalable and can support both basic robots and intricate robotic systems with multiple interconnected parts, making it adaptable for a wide range of scientific and commercial uses. Together, these benefits make ROS desirable, enabling developers to design complex and cutting-edge robotic solutions. The ROS version used in this work is ROS Noetic. Noetic is the latest long-term support of ROS, designed to provide stability and support for an extended period. It is fully compatible with Python3 and contains numerous updates and improvements compared to previous distributions.

3.1.2 DeeBoB

Initially, in the work of this thesis, a set of packages called DeeBoB (Deep learning with Building Blocks for Sensor Data Processing) was used for development, this framework is part of a much larger company program simply called BoB that is used for company purposes and it is not distributed. BoB (Building Block for Sensor Data Processing) is a software framework for sensor data processing from the Institute of Transportation Systems of the German Aerospace Center (DLR). It allows users to create, record, replay, annotate and visualize sensor data in order to extract, evaluate and utilize information for research projects, it is composed of other research projects with different focuses. DeeBoB enables both, modular design of deep learning applications and easy integration of artificial intelligence into BoB,

in other words, this framework has been used to create modular pipelines for training and testing custom neural network. Indeed, it provides a set of Python and C++ ROS nodes that act as building blocks and use standard messages to ensure the compatibility and inter-operability with the eco-system as well as other training and inference systems. The underlying idea is the following: each node encapsulates some of the typical common steps for training neural networks, carrying out the actions required by publishing the minimum information on one or more topics according to a predefined standard. The subsequent node is subscribed to one or more topics published by previous nodes, once it has obtained data, it processes them through callback functions and publishes its own output topics. The published data are standardized and almost always corresponds to images and boxes, but there are some asynchronous data that correspond to events that occur during the training process, such as the end of an epoch, then the subsequent restart of the dataset and saving of the model. In reality, even the previous data are asynchronous, that is, they are not regulated by clock or timer, however there is a backhand node, called Data Flow node, which is responsible for managing the data flow by paying attention to the images loaded and the outputs published by the model.

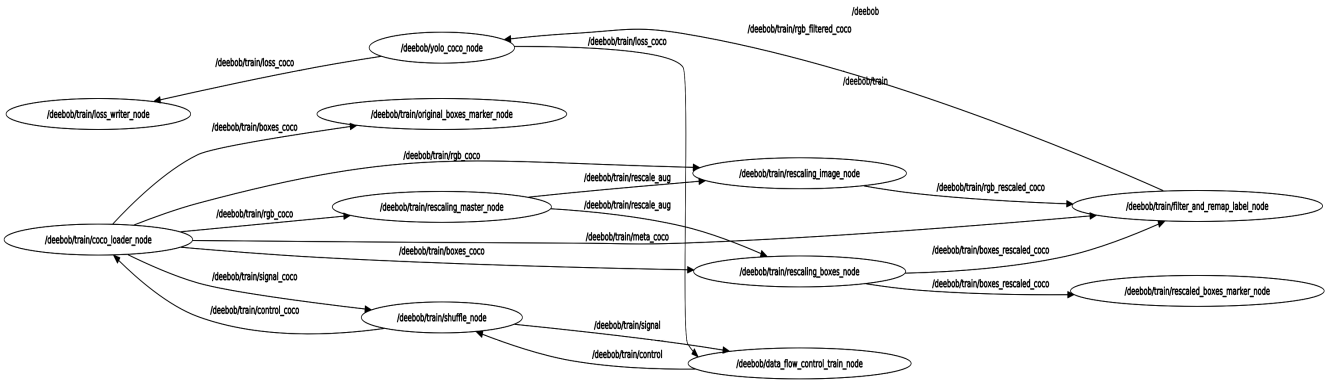


Figure 3.1: View of the DeeBoB training pipeline

Fig 3.1 shows a typical training pipeline with a tool called *rqt*; the rounded shapes represent nodes while the other names are topics. Images are loaded into the pipeline thanks to a loader node, which reads the dataset's images passed as a parameter and standardizes the ground truth boxes and images according to DeeBoB messages. It can also be reset with a signal and started with a message containing a bos (Begin of Stream) inside the body. These data pass through augmentation nodes that change the information depending on their function, such as rescaling or filtering. Then, the most important is the model node. It is the core of the pipeline and encapsulates the main mechanisms of a specific convolutional neural network model. It provides five main functions besides the initialization one: the training, validation, inference functions, and the callbacks to save the model and signal the end of an epoch.

Finally, the node publishes the loss as output in the case of training and validation and the predicted bounding boxes in the case of inference. In order to construct a new model node, the logic of the original model must be replaced with the prior functions, and any necessary adjustments to the input parameters must be made. This results in a new implementation. Since DeeBoB implements functions rather than scripts, the code flow differs from the conventional top-down methodology. As a result, DeeBoB is a technological shift that necessitates a new perspective from the development standpoint. This thesis work created a model node that implements YOLOv7 with the intention of conducting all of the experiments, which will be discussed later.

3.1.3 YOLOv7

What YOLOv7 is and how it works in general terms has already been described previously; furthermore, the hyperparameters used will be described later, but before going further, it is helpful to describe in detail the architecture of YOLOv7 concerning the files of which it is composed since they will be subsequently modified.

The following code sections are taken from the yolov7.yaml file.

```
1   # parameters
2   nc: 80  # number of classes
3   depth_multiple: 1.0  # model depth multiple
4   width_multiple: 1.0  # layer channel multiple
5
6   # anchors
7   anchors:
8     - [12,16, 19,36, 40,28]  # P3/8
9     - [36,75, 76,55, 72,146]  # P4/16
10    - [142,110, 192,243, 459,401]  # P5/32
```

The lines above contain valuable parameters for the model architecture. The number of classes at line 2 represents the classes the model will detect; it is set as 80 because YOLOv7 is trained with COCO, which has that number of classes; however, this parameter is overwritten by the one given within the data configuration file. The number of layers in the model can be scaled using the depth multiple at line 3. The model will have the same number of layers as the original YOLOv7 model if its value is 1.0. A deeper model with more layers will be produced by increasing the value, while a shallower model with fewer layers will be made by decreasing the value. The number of channels in the model is scaled using the width multiple at line 4. The model will have the same channels as the original YOLOv7 model if its value is 1.0. A larger model with more channels will be produced by increasing the value, while a narrower model with fewer channels will be created by decreasing the value. Anchor lists in lines 8,9, and 10 represent the three anchor box types used in the model at different scales. Unlike the first version, which used only two anchor boxes for each cell, YOLOv2 (or YOLO9000) [40] switched to using five anchor boxes, and then nine anchor boxes in YOLOv3 [41] with three different scales, so the model has three different sizes for more extensive and smaller objects in the image. The same pattern remains in YOLOv7. The anchor boxes are defined as pairs of width and height values, and they are used to predict the bounding boxes of objects in an image; they correspond to the three different scales of feature maps in the YOLOv7 model, P3, P4, and P5. Before training, the algorithm looks at whether

objects in ground truth data are smaller than the anchors' dimensions and adapts the list if necessary. This function can be suppressed with an input parameter.

YOLO network consists of three main components that take the image and sequentially process it to the final output: backbone, neck, and head. The first is a convolutional neural network that creates image features, the second is a collection of neural network layers that combines and mixes features to pass it to the next stage for prediction, and the last consumes features from the neck and creates prediction outputs. The backbone consists of a convolutional neural network responsible for extracting features from the input image and creating a feature map that the model can use; furthermore, it is divided into five stages, each with a different spatial resolution.

```
12  # yolov7 backbone
13  backbone:
14  [from, number, module, args]
15  [[-1, 1, Conv, [32, 3, 1]],  # 0
16
17  [-1, 1, Conv, [64, 3, 2]],  # 1-P1/2
18  [-1, 1, Conv, [64, 3, 1]],
19
20  [-1, 1, Conv, [128, 3, 2]],  # 3-P2/4
21  [-1, 1, Conv, [64, 1, 1]],
22  [-2, 1, Conv, [64, 1, 1]],
23  [-1, 1, Conv, [64, 3, 1]],
24  [-1, 1, Conv, [64, 3, 1]],
25  [-1, 1, Conv, [64, 3, 1]],
26  [-1, 1, Conv, [64, 3, 1]],
27  [[-1, -3, -5, -6], 1, Concat, [1]],
28  [-1, 1, Conv, [256, 1, 1]],  # 11
29
30  [-1, 1, MP, []],
31  [-1, 1, Conv, [128, 1, 1]],
32  [-3, 1, Conv, [128, 1, 1]],
33  [-1, 1, Conv, [128, 3, 2]],
34  [[-1, -3], 1, Concat, [1]],  # 16-P3/8
35  [-1, 1, Conv, [128, 1, 1]],
36  [-2, 1, Conv, [128, 1, 1]],
37  [-1, 1, Conv, [128, 3, 1]],
38  [-1, 1, Conv, [128, 3, 1]],
39  [-1, 1, Conv, [128, 3, 1]],
40  [-1, 1, Conv, [128, 3, 1]],
41  [[-1, -3, -5, -6], 1, Concat, [1]],
42  [-1, 1, Conv, [512, 1, 1]],  # 24
```

```
43
44  [-1, 1, MP, []],
45  [-1, 1, Conv, [256, 1, 1]],
46  [-3, 1, Conv, [256, 1, 1]],
47  [-1, 1, Conv, [256, 3, 2]],
48  [[-1, -3], 1, Concat, [1]], # 29-P4/16
49  [-1, 1, Conv, [256, 1, 1]],
50  [-2, 1, Conv, [256, 1, 1]],
51  [-1, 1, Conv, [256, 3, 1]],
52  [-1, 1, Conv, [256, 3, 1]],
53  [-1, 1, Conv, [256, 3, 1]],
54  [-1, 1, Conv, [256, 3, 1]],
55  [[-1, -3, -5, -6], 1, Concat, [1]],
56  [-1, 1, Conv, [1024, 1, 1]],  # 37
57
58  [-1, 1, MP, []],
59  [-1, 1, Conv, [512, 1, 1]],
60  [-3, 1, Conv, [512, 1, 1]],
61  [-1, 1, Conv, [512, 3, 2]],
62  [[-1, -3], 1, Concat, [1]], # 42-P5/32
63  [-1, 1, Conv, [256, 1, 1]],
64  [-2, 1, Conv, [256, 1, 1]],
65  [-1, 1, Conv, [256, 3, 1]],
66  [-1, 1, Conv, [256, 3, 1]],
67  [-1, 1, Conv, [256, 3, 1]],
68  [-1, 1, Conv, [256, 3, 1]],
69  [[-1, -3, -5, -6], 1, Concat, [1]],
70  [-1, 1, Conv, [1024, 1, 1]],  # 50
71  ]
72
73
```

The configuration file syntax makes it possible to build complex architectures. For example, in line 15, the first number expresses the layer to which the convolutional layer is appended, the second how many times it is repeated, and the list is the

input arguments of the layer. The layer's name matches a custom class defined in an internal library; when the program reads the configuration file, it loads the corresponding layer in the internal libraries with the specified rules. For example, Conv layer is a custom combination of a PyTorch Conv2d layer, BatchNorm2d, and SiLU activation function. The MP layer corresponds to the PyTorch implementation MaxPool2d, and Concat concatenates their different outputs.

```
# yolov7 head
head:
[[-1, 1, SPPCSPC, [512]], # 51

[-1, 1, Conv, [256, 1, 1]],
[-1, 1, Upsample, [None,2,'nearest']],
[37, 1, Conv, [256, 1, 1]], #route P4
[[-1, -2], 1, Concat, [1]],

[-1, 1, Conv, [256, 1, 1]],
[-2, 1, Conv, [256, 1, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[[-1,-2,-3,-4,-5,-6], 1, Concat, [1]],
[-1, 1, Conv, [256, 1, 1]], # 63

[-1, 1, Conv, [128, 1, 1]],
[-1, 1, Upsample, [None,2,'nearest']],
[24, 1, Conv, [128, 1, 1]], # route P3
[[-1, -2], 1, Concat, [1]],

[-1, 1, Conv, [128, 1, 1]],
[-2, 1, Conv, [128, 1, 1]],
[-1, 1, Conv, [64, 3, 1]],
[-1, 1, Conv, [64, 3, 1]],
[-1, 1, Conv, [64, 3, 1]],
[-1, 1, Conv, [64, 3, 1]],
[[-1,-2,-3,-4,-5,-6], 1, Concat, [1]],
[-1, 1, Conv, [128, 1, 1]], # 75


[-1, 1, MP, []],
[-1, 1, Conv, [128, 1, 1]],
[-3, 1, Conv, [128, 1, 1]],
[-1, 1, Conv, [128, 3, 2]],
[[-1, -3, 63], 1, Concat, [1]],

[-1, 1, Conv, [256, 1, 1]],
[-2, 1, Conv, [256, 1, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[[-1,-2,-3,-4,-5,-6], 1, Concat, [1]],
[-1, 1, Conv, [256, 1, 1]], # 88

[-1, 1, MP, []],
[-1, 1, Conv, [256, 1, 1]],
[-3, 1, Conv, [256, 1, 1]],
[-1, 1, Conv, [256, 3, 2]],
[[-1, -3, 51], 1, Concat, [1]],

[-1, 1, Conv, [512, 1, 1]],
[-2, 1, Conv, [512, 1, 1]],
[-1, 1, Conv, [256, 3, 1]],
[-1, 1, Conv, [256, 3, 1]],
[-1, 1, Conv, [256, 3, 1]],
[-1, 1, Conv, [256, 3, 1]],
[[-1,-2,-3,-4,-5,-6], 1, Concat, [1]],
[-1, 1, Conv, [512, 1, 1]], # 101
```

The YOLOv7 head is responsible for processing the feature maps generated by the backbone network and making predictions. The part shown would correspond to the neck, as it prepares the features for the final detection part, but since YOLOv7 calls it head, it will be called the same way. The head comprises several layers and operations that process the feature maps and make predictions. Layers are not all connected to the previous one in a top-down view, but some capture the output generated by backbone operations. The head starts with an SPPCSPC layer with 512 channels, which is responsible for capturing context information from the feature maps generated by the backbone network; it is the implementation of the CSPNet

[42] (Cross Stage Partial Network) with an SPP (Spatial Pyramid Pooling Layer) block [43]. Generally, all other layer blocks are primarily convolutional layers; there are also Concat and MP layers. The second and the fourth blocks are connected to P4 and P3 in the backbone, respectively.
Lastly, three RepConv layers are connected to layer 75 in line 103, layer 88 in line 118, and layer 101 in line 133. These layers are an extension of the basic convolutional one; it is an implementation of Represented Convolution [22] (called RepVGG) by Ding et al.

```
135     [75, 1, RepConv, [256, 3, 1]],
136     [88, 1, RepConv, [512, 3, 1]],
137     [101, 1, RepConv, [1024, 3, 1]],
138
139     [[102,103,104], 1, IDetect, [nc, anchors]],   # Detect(P3, P4, P5)
140   ]
```

The last block takes the three RepConv outputs and produces predictions. It is called IDetect, but many other architectures use Detect instead of it. While the latter implements just a convolutional final layer, the first derives from previous studies of YOLO-R [44], which uses tools to manage and use implicit and explicit knowledge to increase model training.

3.2 Methodology: Benchmarking, Dataset and Model Selection

This section goes on to an examination of tools used to maximize the training of object detectors. This stage of the research includes a selection of benchmarking criteria, a thoughtful discussion regarding hyperparameters and model selection, and an overview of the dataset that serves as the primary basis for the investigation. Establishing an appropriate initial setup is crucial because it is a fundamental step that enables comparing results across different models.

Benchmarking

Benchmarking refers to evaluating performance and comparing it to current standards. Within this investigation, the aim is to compare models that use diverse training tools and, sometimes, mix them together. Table 1.1 shows a way to evaluate models, some of which will be considered. Other illustrations in the official paper focus on other metrics, such as speed (in terms of frames per second) or floating point operations (FLOPs), which are not engaging in the ongoing scenario;

in fact, this investigation concerns training performance, as a consequence, metrics that express an external meaning and do not help to understand training performance better will not be considered. Since only one type of model will be used, i.e. the basic YOLOv7 model, and the changes that will be made to its architecture will only concern the addition of drop layers that do not change the number of parameters used, it is unnecessary to report on the number of model parameters in future comparisons. Similarly, the image size will always be 1280. As a result, the metrics used to evaluate each model will be Precision, mAP, and mAP50.

Model and Hyperparameters

The model under consideration is YOLOv7 basic, which will provide a powerful weapon and a fairly general-purpose neural network capable of being used in any context. The associated hyperparameters are the default ones as follows:

```
1  lr0: 0.01  # initial learning rate (SGD=1E-2, Adam=1E-3)
2  lrf: 0.1  # final OneCycleLR learning rate (lr0 * lrf)
3  momentum: 0.937  # SGD momentum/Adam beta1
4  weight_decay: 0.0005  # optimizer weight decay 5e-4
5  warmup_epochs: 3.0  # warmup epochs (fractions ok)
6  warmup_momentum: 0.8  # warmup initial momentum
7  warmup_bias_lr: 0.1  # warmup initial bias lr
8  box: 0.05  # box loss gain
9  cls: 0.3  # cls loss gain
10 cls_pw: 1.0  # cls BCELoss positive_weight
11 obj: 0.7  # obj loss gain (scale with pixels)
12 obj_pw: 1.0  # obj BCELoss positive_weight
13 iou_t: 0.20  # IoU training threshold
14 anchor_t: 4.0  # anchor-multiple threshold
15 fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
16 hsv_h: 0.015  # image HSV-Hue augmentation (fraction)
17 hsv_s: 0.7  # image HSV-Saturation augmentation (fraction)
18 hsv_v: 0.4  # image HSV-Value augmentation (fraction)
19 degrees: 0.0  # image rotation (+/- deg)
20 translate: 0.2  # image translation (+/- fraction)
21 scale: 0.5  # image scale (+/- gain)
22 shear: 0.0  # image shear (+/- deg)
23 perspective: 0.0  # image perspective (+/- fraction), range 0-0.001
24 flipud: 0.0  # image flip up-down (probability)
25 fliplr: 0.5  # image flip left-right (probability)
26 mosaic: 1.0  # image mosaic (probability)
27 mixup: 1.0  # image mixup (probability)
28 copy_paste: 0  # image copy paste (probability)
29 paste_in: 0  # image copy paste (probability), use 0 for faster training
30 loss_ota: 1 # use ComputeLossOTA, use 0 for faster training
```

YOLOv7 allows developers to choose two types of optimizers, namely Adam and SGD; the latter is the one that will be used to which three different parameters are added, specifying three learning rates: lr0, lr1, and lr2. Parameters from lines 1 to

7 concern the behavior of the learning rate and the optimizer, thus the consequent updating of the weights. They define the maximum point of the warmup phase, where the values of the three learning rates align, the initial value for lr2, which corresponds to lrf, and the final minimum point at the end of the epochs, which coincides with lrf*lr0. The initial value of lr0 and lr1 is always 0. Furthermore, the other parameters configure warmup during epochs. Parameters from lines 8 to 15 set gain values for losses and IoU threshold. Then, the ones from lines 16 to 29 determine values for augmentations. The augmentation techniques are described in the comments, and they are respectively: saturation, rotation, translation, scaling, shear, change of perspective, flipping and mosaic, mixup and paste augmentation [45]. These values will be switched on or off following the training process under investigation. When switched on, their value will be as described, and vice versa, it will be 0. The paste augmentation variables will always be left at zero because they are designed for segmentation, not object detection. The last parameter selects the loss calculation function; LossOTA is a YOLOv7 custom loss function, and it is the one that will be used.

Dataset, Training Parameters and Hardware

The research aims not only to verify the improvement that specific tools provide during model training but also to help future developers face the challenges of exploring customized models on proprietary datasets that can sometimes be unbalanced and extremely small. Furthermore, the investigation is approached within the business context of this work, i.e., transport or, more generally, the urban context. Consequently, the choice of dataset is derived from these premises. The data did not come from an official dataset but from images owned by the DRL; it consists of top-view footage of busy urban areas of various places worldwide and contains five types of classes: bike, car, lorry, bus, and pedestrian. It has just under 1500 images with a medium-high resolution, but they will not all be used for training purposes; instead, to align with the intentions described above, only 174 images for the train set and 116 for the validation set will be taken, and used while the other images, the test set, will be adopted to verify the effectiveness of hard example mining methods. The model resizes all images to a square dimension of 1280. Annotations of the dataset were written in XML files, one for each image. The Yolo annotation format requires images to have a matching text file with the corresponding JPG or PNG image name. Each annotation line identifies a detection for the image in question and consists of five values in this order: label, x center, y center, width, and height. As a consequence, many Python scripts were created to manage the data; in particular, they were created for conversion from XML to Yolo format in text files, creation

of datasets according to the hierarchy required by Yolo according to preset ratios, creation of random datasets or according to rankings from a text file containing the uncertainty scores of the test images.

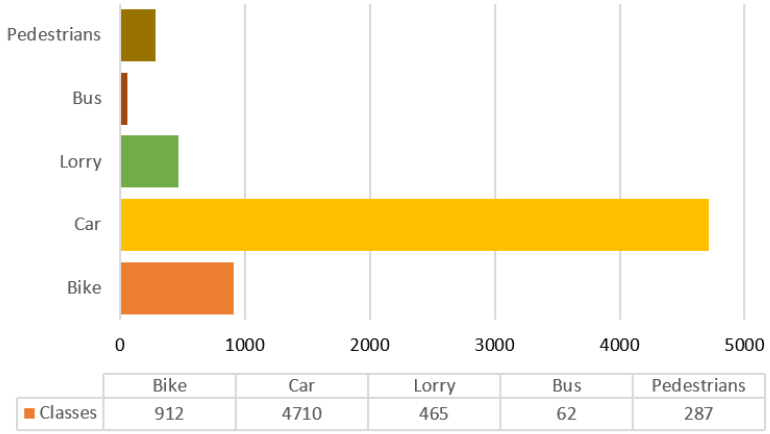


Figure 3.2: Classes sparsity in training set

Fig. 3.2 illustrates the sparse distribution of classes in the training set, which is reflective of real-life scenarios. Cars have a strong predominance over lorries and buses, while bicycles and pedestrians are smaller in size. This means that the model may have difficulty recognizing buses due to their scarcity, while bicycles and pedestrians may pose a challenge due to their size and higher frequency in the images. The viewpoint of the images also varies, with some taken from cameras on tall buildings and others from lower viewpoints like those on lampposts or small buildings. The weather conditions and brightness of the scenes also differ, with some images taken at night (see Fig 3.3). Furthermore, the density of objects in the images varies greatly, with some containing hundreds of objects while others have almost none.

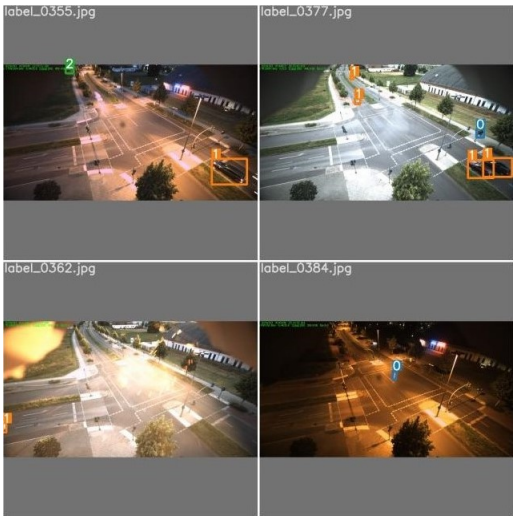


Figure 3.3: Examples of training images

Each model will be fed with training data for 150 epochs; it has been seen that this number of epochs allows for fast training and enables the model to achieve convergence in terms of knowledge. Pretrained weights will be used, in particular the *yolov7_training* model given by the authors. As a result, transfer learning is applied to each model. The computational operations will be carried out on the hardware provided by the DLR; in particular, three NVIDIA GPUs will be used to parallelize and allow more training to be run simultaneously; the GPUs are NVIDIA GeForce RTX 1080 (8GB), NVIDIA GeForce RTX 1080 Ti (12GB) and NVIDIA RTX A6000 (48 GB).

Next steps

Since the aim is the one described before, it is a good practice to have different models trained with diverse tools to compare them. The tools in focus are data augmentations, dropout regularization, and hard example mining. The first should be split into different attempts because YOLOv7 contains many operations; as a consequence, the data augmentation tool will be analyzed in 4 phases: no augmentations, spatial augmentation (rotation, translation, flipping, scaling, shear, change of perspective), color augmentation (hue, saturation, value), spatial and color augmentations and finally all augmentations (including mixup and mosaic). The second will be applied with data augmentation because almost all neural networks adopt data augmentation techniques, so that this model will be compared with its counterpart without dropout. The same goes for the third one, which will, however, be given two attempts, one by adding 100 random images from the test set to the training set and another by adding the 100 most difficult images in terms of uncertainty. This will result in a planning table like the one below:

Table 3.1: Model table plan

Model	**S-AUG**[1]	**C-AUG**[2]	**O-AUG**[3]	**DROP**[4]	**+100R**[5]	**+100U**[6]
M0	no	no	no	no	no	no
M1	yes	no	no	no	no	no
M2	no	yes	no	no	no	no
M3	yes	yes	no	no	no	no
M4	yes	yes	yes	no	no	no
M5	yes	yes	yes	yes	no	no
M6	yes	yes	yes	yes	no	yes
M7	yes	yes	yes	yes	yes	no

M0 will be used to analyze the behavior of YOLOv7 without optimization tools, M1 to verify the improvement of spatial augmentation, M2 for color augmentation, M3 for the combination of color and spatial, and M4 for all data augmentation techniques. M5 will express the performance of a dropout solution, while M6 and M7 will demonstrate the usefulness of applying hard examples. Since the dataset is small, four different training iterations were done for each model to avoid overperforming or underperforming models; the results correspond to the models' average.

[1]Spatial Augmentation
[2]Color Augmentation
[3]Other Augmentations (Mixup, Mosaic, and Paste)
[4]Dropout Layers
[5]Adding 100 Random Images
[6]Adding the First 100 Images with Highest Uncertainty Score

3.3 Implementing Neural Network Optimization Techniques

Let us now enter into the actual implementation of the model and techniques under analysis. As mentioned before, everything was made using Python and PyTorch; many other libraries, such as NumPy or OpenCV, were used but will not be discussed. At the beginning of the work, the original idea was to implement a YOLOv7 solution within DeeBoB and then deploy the required tools to perform the investigation entirely with it, but this path has hit a significant roadblock that will be debated later; therefore, the final implementation and benchmarking is made modifying the original YOLOv7 script. The issues encountered with DeeBoB expose factors that are relevant from the point of view of this research; it is because of this that they will be exposed anyway.

3.3.1 DeeBoB implementation

The detailed DeeBoB pipeline used for a new YOLOv7 implementation is as follows. It all starts with a bash script, which manages other configuration files. The script takes the necessary arguments, such as dataset path, pretrained weights, model architecture, number of epochs, and a relevant file containing all the attributes each node must receive, and launches the first configuration file after creating directories for logging and results. This one renames some parameters and paths and initializes others to default if not present but does not start any nodes, and then it starts another configuration file. The latter creates different absolute paths for training, validation, and, if necessary, inference and starts the pipelines. The training pipeline is shown in 3.1; the validation pipeline is identical, while the inference one is shorter because it does not need augmentation nodes. After setting up the nodes, the life cycle of the process is managed by the initial script, which is responsible for starting and maintaining the loop of epochs and for each epoch to reset the data loaders with the *reset* command, to start data streaming with the *signal* command which contains the *bos* string within it and to cede control to the data flow control node. Once control has been obtained back, it saves the model with the *save_model* command, calls a new epoch with the *epoch_request* command, and starts the cycle again if there are no other active pipelines to start. When the epochs are finished, it then kills its own process.
The first node of the pipeline is the *data_loader_node*; it is activated when receiving a *signal* containing a *bos* message, then, it loads images and boxes into the stream without processing them. Image and box messages are standard; the first refers to

ROS´s design (left), and the second to a custom pattern (right) as described below:

```
# Uncompressed image
# (0, 0) is at top-left corner

Header header # Should be timestamp

uint32 height # image height
uint32 width # image width

string encoding # Encoding of pixels

uint8 is_bigendian # is it bigendian?
uint32 step # Full row length in bytes
uint8[] data # actual matrix data

```

```
# The boundaries in pixel coordinates
int32 xmin
int32 xmax
int32 ymin
int32 ymax

# Covariance matrix
uint8 COV_WIDTH = 4
uint8 COV_HEIGHT = 4
float32[16] covariance

# The label of box
uint64 label
float32 confidence
```

The covariance is set to 0 for the current purpose, while confidence is set to 1 for ground truth data. Since the YOLOv7 dataloader is not used, the images must be scaled to the preset network size of 1280, and the *rescaling_node* is responsible for this process. It is composed of a triad of nodes: the *rescaling_master_node*, the *rescaling_image_node*, and the *rescaling_boxes_node*. The master node configures each sample through a standard message, and the worker nodes perform reshaping. Outputs are rescaled images and boxes modified to maintain the same height/width ratio, and grey padding is added according to the process that YOLOv7 also uses. Then, boxes are filtered and remapped by a *filter_node* which, referring to a file that remaps labels numbers to form a set of successive numbers, reproduces the modified boxes and, if specified, can also act as a filter to prevent certain classes from being learned. These nodes have a C++ implementation. At the end of the pipeline, there is the *model_node*, which is subscribed to the topic published by the filter, and it is at this level the YOLOv7 implementation was made. As it is now called, the *yolo_node* has five main functions. The initialization function is similar to the original code, except for some parameters, it prepares everything needed to start the training, from hyperparameters and pretrained weights to optimizer and loss function, and it publicizes and subscribes to the referred topics, and it offers the *save_model* and *epoch* services. The topics subscribed are: *train_image* and *train_boxes* for training, *val_image* and *val_boxes* for validation, and *infer* for inference, the latter is the image massage without boxes. The topics published are: *train_loss* and *val_loss*, which are the losses connected to training and validation predictions, and *boxes*, which is the bounding box prediction of the image. Some callback functions associated with subscribed topics take the received messages and transform data to a NumPy array. As a result, the *do_train* and *do_val* func-

tions receive well-prepared ground truth data. YOLOv7 works using PyTorch, so information must be converted. Hence, these functions initially encapsulate a pre-processing phase, making obtaining correct data possible. Then, YOLOv7 inference and non-maximum suppression functions are called, losses are computed comparing predictions to ground truth, and the backward pass is made. Finally, the output message is prepared and published on the topic as described below:

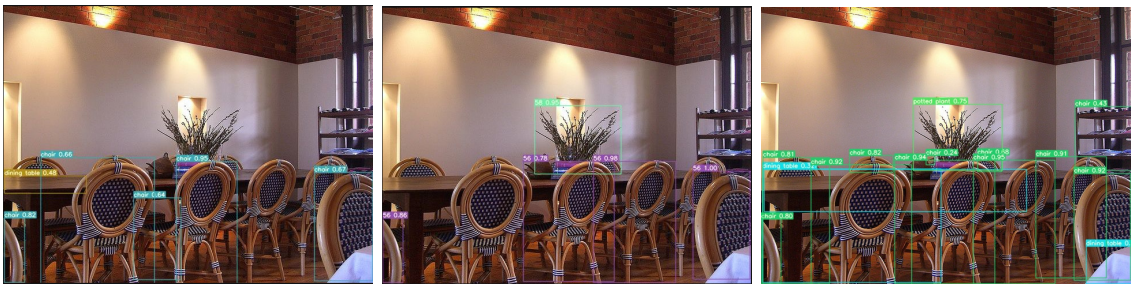
```
std_msgs/Header header

float64[]  loss
uint32 epoch
uint32 iteration
float64[] hyper_parameter
uint32 loop
uint32 current_batch_size
float32[] learning_rate
```

Other DeeBoB nodes can use this message to create markers to plot images and even to draw tensorboard logging to monitor the training and have a better general overview of the whole model. The inference function is similar to the others except for the loss computation. The two service functions enable the bash script file to notify the end of an epoch and order the model to save its weights. These two functions encapsulate some of the YOLOv7 code at the end of the epoch loop. Each node within the pipeline has a queue buffer, called *queue_size*, that is managed by the *data_flow_control_node*. This actor controls the data flow in the application, which means having data sources send the successive samples at the right speed so as not to overflow the pipeline queues but also to keep them busy with work. The general idea is to use message timestamps to take into account the time it takes for data to pass through the pipeline and, consequently, to regulate the input flow to allow nodes to be constantly busy without waiting or having too much data to process with the risk of it being lost due to running out of buffer space.

Unfortunately, a hindering issue was encountered working on DeeBoB. The pipeline framework does not yet have the necessary augmentation tools to train a good model optimally; it holds a couple of nodes for cropping and flipping but which are not sufficient to achieve a similar-yolo augmentation, these nodes were built in C++; as a consequence, there was no possibility in terms of time to create new nodes. The problem was evident when the model could not make a reasonable inference, as it did not recognize all objects in the scene (see Fig 3.4). One might think these tools are not necessary at all and that a model can be trained even without them, but, first of all, the investigation of these tools is an integral part of this thesis research,

and, secondly, the result obtained by not using data augmentation with the official YOLOv7 script underlines the problem above.



(a) YOLOv7 no augm (b) DeeBoB (c) YOLOv7 with augm

Figure 3.4: Inference comparison between YOLOv7 and DeeBoB

Even when training a model with massive data, such as the COCO dataset, it only behaves well with augmentation. This setback led to the first reasonable result, namely the discovery of the importance of data augmentation techniques to increase the variety of images and help the model generalize features by making it more robust. In conclusion, all operations shown will no longer be done through DeeBoB but within the original script.

3.3.2 Augmentations implementation

YOLOv7 implements all kinds of data augmentations within the official code; each transformation can be found in the *utils/dataset.py* library. Those techniques can be divided into four categories: spatial augmentation, color augmentation, mosaic augmentation, and mix-up augmentation. Every method manipulates the image differently and with a different contribution to efficiency. Nevertheless, they can be combined together (see Fig 3.5g). Their algorithms will be listed in order of activation and commented on below.

Algorithm 1 Mosaic Algorithm

Input: image index, border

Mosaic center $yc, xc \leftarrow random(-border, 2 * size + border)$
Select 3 other random *indices*
for *index* in *indices* **do**
    $img, h, w = load\_image(index)$
    if $i_{index} == 0$ **then**
        Create a larger canvas with dimensions twice the size of individual images.
        Compute mosaic region
        $x1a, y1a, x2a, y2a = max(xc - w, 0), max(yc - h, 0), xc, yc$
        Compute image region
        $x1b, y1b, x2b, y2b = w - (x2a - x1a), h - (y2a - y1a), w, h$
    else if $i_{index} == 1$ **then**
        Compute mosaic and image regions
    else if $i_{index} == 2$ **then**
        Compute mosaic and image regions
    else if $i_{index} == 3$ **then**
        Compute mosaic and image regions
    end if
    Copy image region onto the mosaic region at the specified position
    Compute padding $padw, padh$
    Convert and resize $labels_{index} \leftarrow xywhn2xyxy(labels, w, h, padw, padh)$
end for
Concat/clip mosaic together

Mosaic augmentation selects four random images (3 in addition to the already taken). It combines them to form a single image (see Fig 3.5e) 4 times larger than the base one; following the positioning of the image, it also moves the labels. Then, it takes a random cutout, selecting a random center within the mosaic and excluding the image from the given border. The cutout is made after all other transformations are applied.

Algorithm 2 Spatial Augmentation Algorithm

Input: img, degrees, translate, scale, shear, perspective

1: Create 3x3 identity matrix C ▷ Center
2: $C[0,2] \leftarrow -width/2$
3: $C[1,2] \leftarrow -height/2$
4: Create 3x3 identity matrix P ▷ Perspective
5: $P[2,0] \leftarrow random(-perspective, perspective)$
6: $P[2,1] \leftarrow random(-perspective, perspective)$
7: Create 3x3 identity matrix R ▷ Rotation and Scale
8: $angle \leftarrow random(-degrees, degrees)$
9: $scale \leftarrow random(1 - scale, 1.1 + scale)$
10: $R[:2] \leftarrow roation\_matrix(angle, center = (0,0), scale)$
11: Create 3x3 identity matrix S ▷ Shear
12: $S[0,1] \leftarrow tan(random(-shear, shear) * \pi/180)$
13: $S[1,0] \leftarrow tan(random(-shear, shear) * \pi/180)$
14: Create 3x3 identity matrix T ▷ Translation
15: $T[0,2] \leftarrow random(0.5 - translate, 0.5 + translate) * width$
16: $T[1,2] \leftarrow random(0.5 - translate, 0.5 + translate) * height$
17: $M = T * S * R * P * C$ ▷ Final Operation Matrix
18: $img \leftarrow apply\_matrix(img, M)$

Spatial augmentations are achieved by utilizing transformation matrices, each serving a specific purpose in shaping the final distorted image (see Fig 3.5b). The process involves five distinctive matrices: the center matrix, responsible for defining the transformation's central point; the perspective matrix, introducing alterations simulating changes in perspective, the rotation and scaling matrix, facilitating rotations and scaling operations, the shear matrix, enabling shear transformations and the translation matrix, for applying spatial shifts. These individual matrices are systematically combined into the final transformation matrix in line 17. It is important to note that the order of multiplication from right to left is pivotal, as altering the sequence leads to diverse outcomes. The transformation matrix encapsulates the cumulative effects.

Algorithm 3 Mix up Algorithm

Input: Mixup ratio, img1, img2, mix up threshold, mosaic

1: **if** $random() < thresh$ AND $mosaic$ **then**
2: $ratio \leftarrow random(8.0, 8.0)$
3: $img1 = (img1 * ratio + img2 * (1 - ratio))$
4: **end if**

Mixup blends two images (see Fig 3.5f) using a random variable *ratio*. Note that this algorithm is not always applied, unlike the others. Furthermore, it does not work if the mosaic is deactivated.

Algorithm 4 Color Augmentation Algorithm

Input: img, hue_gain, saturation_gain, value_gain

$ratio \leftarrow random(-1, 1, 3) * [hue\_gain, saturation\_gain, value\_gain] + 1$
Extract $hue, saturation, value$ from img
Create look up tables
x = [0, 1, ... 255]
$lut\_hue \leftarrow (x * ratio[0])\%180$
$lut\_sat \leftarrow clip(x * ratio[1], 0, 255)$
$lut\_val \leftarrow clip(x * ratio[2], 0, 255)$
$img\_hsv \leftarrow merge(LUT(hue, lut\_hue), LUT(sat, lut\_sat), LUT(val, lut\_val))$
Convert img_hsv to RGB

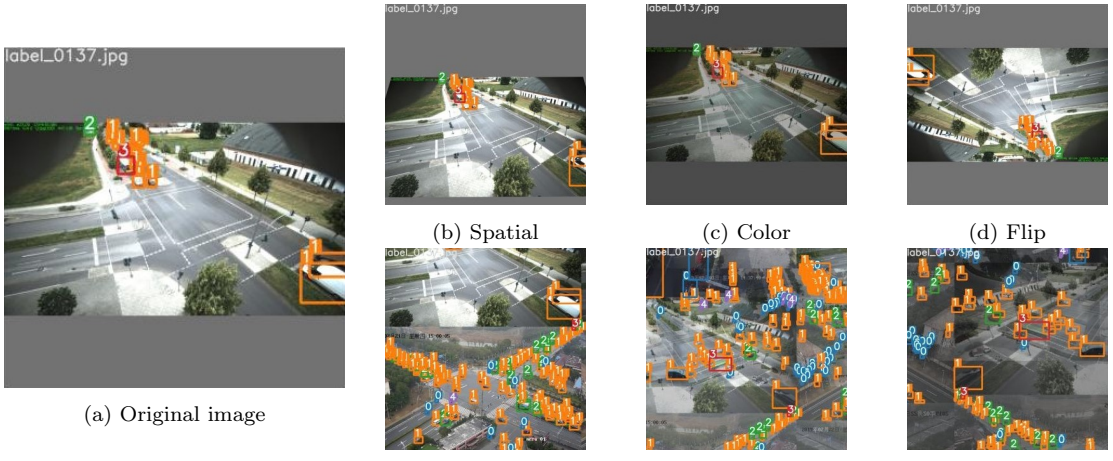
Color augmentation makes use of look-up tables. These tables incorporate the transformations for all values from 0 to 255 for all three channels except hue, which has a range from 0 to 179 in OpenCV, so having a specific color value, one only has to look at the corresponding value in the table to determine the final result (see Fig 3.5c).

Algorithm 5 Flipping Algorithm

Input: Img, flipping_ud thresh, flipping_lr thresh

if $random() < ud\_thresh$ **then**
    Reflect image horizontal
end if
if $random() < lr\_thresh$ **then**
    Reflect image vertical
end if

Finally, flipping reflects the image vertically and horizontally (see Fig 3.5d). This process might not seem very incisive, but for the model, having one reflected image means having two distinct images, thus doubling the variety of images available. As mix-up, there is no absolute certainty that this transformation will be made.



(a) Original image (b) Spatial (c) Color (d) Flip (e) Mosaic (f) Mixup (g) All

Figure 3.5: Data Augmentation applied to an image

3.4 Uncertainty Quantification

Confidence in object detection is the output percentage from the model's output layer; it is not the certainty a model attributes to a discovered object belonging to a specific class; in other words, high confidence does not mean the model prediction is confident [46]. It is crucial to understand the difference between absolute certainty and confidence scores. The hidden contrast between absolute certainty and confidence scores may seem minor in some fields, such as less crucial applications. However, when accuracy is critical, it becomes essential to recognize and deal with this distinction. Misinterpretations in medical diagnostics, for example, might have grave repercussions, emphasizing the necessity for detailed evaluations of prediction reliability. Similarly, when assuring the dependability and robustness of the deployed models is crucial in autonomous driving since pedestrian and passenger safety is at risk, factoring in uncertainty becomes essential. Therefore, although confidence scores indicate a model's level of assurance, identifying and measuring uncertainty is especially important in high-stakes scenarios when forecast reliability is a non-negotiable. Numerous research [47] works have highlighted the significance of quantifying the uncertainty linked to forecasts in machine learning models. In order to measure it, different metrics have been investigated. These metrics offer insightful information on the consistency and dependability of a model's predictions, enabling a more sophisticated comprehension of the degree of trust attached to items that have been spotted.

Metrics

Generally, there are two main types of uncertainty [48]: aleatory uncertainty and epistemic uncertainty. Aleatory uncertainty results from a process or phenomenon's inherent variability, whereas epistemic uncertainty results from ignorance or insufficient information. The latter will be investigated, and the term uncertainty will refer to it from now on. No metric can fully capture uncertainty complexity; rather, a variety of metrics are essential for measuring and comprehending distinct aspects of uncertainty. Different metrics, including entropy, variance [49], and standard deviation [50], provide distinct insights into the degree of uncertainty present in a particular system or model. For example, entropy describes the disorder and unpredictability inside a set of outcomes, whereas variance and standard deviation concentrate on the distribution of data points around the mean.

Entropy

There are different ways to compute uncertainty. One first way is to calculate the entropy of a prediction, as Wu et al. [51] described. Their research is based on the formulas proposed by the American mathematician Claude Shannon [52] in 1948. Entropy, at its core, is a measure of dissimilarity between two probability distributions. It quantifies how different one probability distribution is from another, and it is expressed as:

$$\text{Entropy} = -1 * \sum_{i=0}^{NC-1} (p_i * \ln(p_i)) \tag{3.1}$$

Where NC is the number of classes and p_i is the probability associated with the class i. Usually, the final value of the output layer is passed through a *softmax* activation function, which results in probability values so that the sum of the probability of each class concerning an object is always similar or approximately equal to one. This formula was more extensive in its original version since there was another term calculating the entropy generated by the background, which many two-stage models collect as a class; in the case of YOLOv7, this part has been omitted since it is not detected. This method is simple and applicable to any model.

Monte Carlo Dropout

Gal and Ghahramani [53] describe another way of expressing uncertainty using dropout layers to obtain a Bayesian-like model. In machine learning, Bayesian approximation allows for incorporating prior knowledge and quantifying uncertainty in predictions. Unlike the proposed dropout solution, this is powerful but requires more computational power.

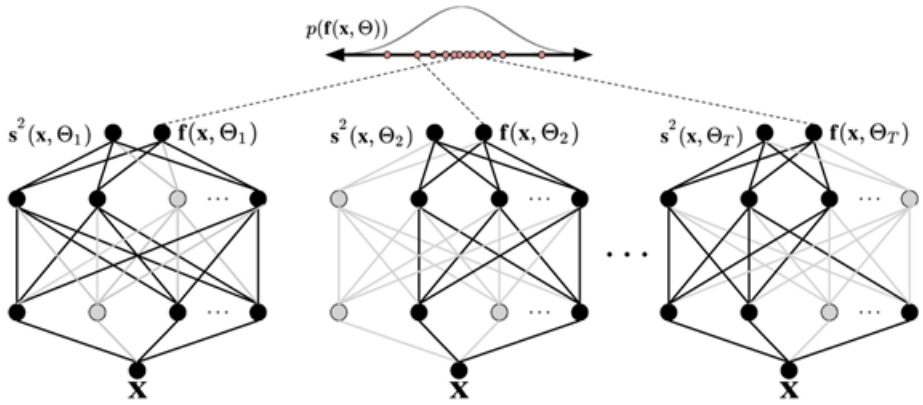


Figure 3.6: Dropout as Bayesian approximation

These layers randomly select neurons in the neural network's feature maps and mask those neurons during training and inference. Thus, applying the Monte Carlo method allows one to model and determine the uncertainty in deep neural networks. Since in every training iteration, the model randomly samples the neurons to be dropped out in each layer (according to that layer's dropout rate), a different set of neurons are being dropped out each time. Hence, the model's architecture is slightly different for each iteration, and the result can be understood as an ensemble average of numerous neural networks trained on a single batch of data (see Fig 3.6). In statistics, Monte Carlo refers to a class of computational algorithms that rely on repeated random sampling to obtain a distribution of some numerical quantity. The underlying concept is to use randomness to solve problems that might be deterministic in principle. In deep learning, all the necessary is to apply dropout at test time. Then, instead of one prediction, there will be many, one by each model, and then it is possible to average them or to analyze their distributions.

DropBlock

Ghiasi et al. [54] introduced DropBlock, a structured dropout method introduced as a regularization technique for convolutional neural networks. DropBlock lowers units collectively in a continuous area of a feature map, unlike typical dropout, which drops features at random. By addressing the spatial correlation of features in convolutional layers, this structured method stops information from spreading via the neighboring pixels of the discarded features. The authors discovered that improving accuracy by using DropBlock in skip connections with convolution layers and improving accuracy and robustness to hyperparameter decisions may be achieved by gradually increasing the number of dropped units during training. Based on experimental findings using the ResNet-50 architecture, it enhanced the Average Precision of RetinaNet for COCO detection and improved accuracy on ImageNet by more than 1.6%. The technique has been demonstrated to perform better in regularising convolutional networks than classical dropout. Additionally, Deepshikha et al. [55] demonstrate how DropBlock can effectively compute and improve uncertainty quantification.

3.4.1 Implementation of Uncertainty Quantification Techniques within YOLOv7

The implementation occurred on three parallel paths, not all of which were subsequently used. The core of developments lies in the calculation of entropy according to the formula mentioned above; in some circumstances, its calculation has been moved to other parts of the code, but the process's substance has remained the same. It was decided to use entropy as the reference measure of uncertainty both because it had already been used successfully by other projects [56] and because, after a few examinations, it turned out to be a sufficiently "high" measure, i.e. one from which values were not so small that they were almost incomparable. The first path is the simple calculation of the entropy of a prediction.

```
def non_max_suppression_and_entropy(pred, conf_thres, iou_thres):
    ... # Settings
    xc = prediction[..., 4] > conf_thresh
    entropies_raw = torch.zeros((prediction.shape[0], max_num_detection)) #1280*1000
    entropies = []
    output_pred = [torch.zeros((0, 6), device=prediction.device)] * prediction.shape[0]
    output_ent = [torch.zeros((0), device=prediction.device)] * prediction.shape[0]

    for xi, x in enumerate(predictions): # image index, image inference
        # Apply costraints
        x = x[xc[xi]]
        cls_confidences = x[:, :5]
        obj_confidences = x[:, 4]
        # Delete rows if conf < conf_tres
        mask = max_conf_values >= conf_thresh
        x_ent = x[mask]
        cls_prob = x_ent[:, 5:]
        # Compute entropy for each prediction
        for j, each_cls_prob in enumerate(cls_prob):
            entropies_raw[xi][j] = -torch.sum(each_cls_prob*torch.log(each_cls_prob))
        # Delete unused rows
        neg_mask = entropies_raw[xi] == 0
        entropies.append(entropies_raw[xi][~neg_mask]
        ...
        # Compute good predictions indices
        i = torchvision.ops.nms(boxes, scores, iou_thres)
        ...
        output_pred[xi] = x[i]
        output_ent[xi] = entropies[xi][i]

    return output_pred, output_ent
```

YOLOv7 uses a custom non-maximum suppression (NMS) function. This function is essential to eliminate duplicates and keep only the most accurate detections. After the model has generated a series of bounding boxes with their associated confidences for different objects, NMS is applied to reduce overlaps. It was necessary to modify

this function, as it returns by default a tensor containing as many rows as the detections of an image and six columns, which are respectively: x of the center of the bounding box, y of the center of the bounding box, height and width of the bounding box, prediction confidence and label. The confidence value is the product of the object score, which is the probability that the selected image part is an object and the label score with the highest probability. Consequently, through NMS, all class probabilities are lost. The code was refactored, adding an output list containing the detection entropies. Nevertheless, more is needed. Since it is necessary to obtain the result of several predictions to apply the Monte Carlo method, the NMS function must be modified by returning class scores to manipulate them out of the function.

```
def non_max_suppression_and_output_all_pred(pred, conf_thres, iou_thres):
    ... # return list of predictions (n,6) and predictions with all cls scores (n, 6 + num_cls)
    return output_pred, output_all
```

As described previously, Dropout or DropBlock layers must be present within the network to use the Monte Carlo method. This has been achieved but will be shown in the next section, so they are assumed to be present within the architecture.

```
def detect():
    ...
    MC_prediction = []
    for i in range(opt.num_samples): # collect inferences for Monte Carlo
        with torch.no_grad(): # Calculating gradients would cause GPU memory leak
            pred = model(img, augment=opt.augment)[0] # Inference
        # Apply NMS and obtain prediction with all cls_scores
        pred, pred_all = non_max_suppression_and_output_all_pred(pred, opt.conf_thres,
        ↪  opt.iou_thresh)
        MC_prediction.append(pred_all)
    # data structure from [samples, batch, Tensor[n, 6 + num_cls]] to [batch, samples, Tensor]
    list_batch = []
    for img_idx in range(len(MC_prediction[0])):
        list_image = []
        for sample_idx in range(len(MC_prediction)):
            pred_tensor = MC_prediction[sample_idx][img_idx]
            list_image.append(pred_tensor)
        list_batch.append(list_image)
    mean_detections = merge_similar_predictions(list_batch, threshold=15) # Average predictions
    entropies = [] # Compute entropy
    for avg_prediction in mean_detections:
        cls_scores = avg_prediction[:, 6:]
        entropy = -torch.sum(cls_scores * torch.log(cls_scores + epsilon), dim=1)
        entropies.append(entropy)
    for i, (det, ent) in enumerate(zip(pred, entropies)): # Process detections
        ...
        uncertainty = torch.mean(ent)
        ...
```

The new *detect* function performs as many multiple inferences as indicated by the *num_samples* variable. This variable can be set from 10 to 100 according to how many inferences must be made to apply the Monte Carlo method. Then, the structure of prediction data is changed; at the beginning, there is a structure formed by a list of size *num_samples*, which contains as many lists as the size of the inference batch. Finally, each sublist contains tensors of *num_detections* rows and 11 columns, 6 of the prediction, and the 5 class scores of labels specified for the dataset. This change in structure is made to allow the use of PyTorch functions to calculate the average used in the *merge_similar_predictions* function. First, the entropy of each detection is calculated, and then the entropy of the entire inference is calculated by averaging all the detections. This value was used in two different ways. The first was plotting it inside the resulting image, as was done with the fundamental entropy without the Monte Carlo method. The second was saved within a data structure list containing the path to the image and the uncertainty value ordered by the latter (see Fig 3.7). The list was then written to an external text file, which was used to find the images employed to obtain the M6 model, resulting in the so-called hard samples.


```
/dataset/img_0496.jpg 0.48801
/dataset/img_0416.jpg 0.42380
/dataset/img_0201.jpg 0.41796
/dataset/img_0407.jpg 0.41022
/dataset/img_0891.jpg 0.40626
/dataset/img_0376.jpg 0.40421
/dataset/img_0715.jpg 0.40324
/dataset/img_0291.jpg 0.40137
/dataset/img_0086.jpg 0.39697
/dataset/img_0517.jpg 0.38901
/dataset/img_0532.jpg 0.37582
/dataset/img_0381.jpg 0.36521
/dataset/img_0832.jpg 0.36416
/dataset/img_1146.jpg 0.36392
/dataset/img_0644.jpg 0.36226
/dataset/img_0665.jpg 0.36113
```

Figure 3.7: Uncertainty score plotted (left) and written as list (right)

The *merge_similar_predictions* associates the predictions with each other according to their distance, averages them, and returns a single final prediction as if it were a single inference. When the function is called, the data structure is a list of batches, each containing as many lists as Monte Carlo samples, which consist of as many tensors as there is detection in the image.

```python
def merge_similar_predictions(batch_predictions, device, threshold = 25):
    # Compute distances between predictions
    output = []
    for samples_predictions in batch_predictions: # For each batch
        average_list = []
        # Take the sample with most prediction as reference
        num_detections = max(len(tensor) for tensor in samples_predictions)
        num_samples = len(samples_predictions)
        # For each detection compute a distance matrix with all others detections
        distance_matrix, best_sample = compute_distance_matrix(samples_predictions, num_samples,
        ↪  num_detections, device)
        if best_sample != -1: # If there are detections
            for j in range(num_detections):
                same_detections_index = []
                for m in range(num_samples):
                    for n in range(num_detections):
                        if (distance_matrix[best_sample][j][m][n] != -1) &
                        ↪  (distance_matrix[best_sample][j][m][n] < threshold): # If distance
                        ↪  is less than threshold it is considered good
                            same_detections_index.append((m, n))
                if samples_predictions: # If list is empty the prediction is an outlier
                    same_detections_index.append((best_sample, j))
                values = [samples_predictions[h][k] for h, k in same_detections_index]
                # Average predictions
                average_tensor = torch.mean(torch.stack(values), dim=0)
                average_list.append(average_tensor)
            # Append to output
            if average_list:
                output.append(torch.stack(average_list))
    return output
```

First, the function collects the number of samples and the maximum number of detections between tensors. This is done so that the association algorithm does not have the problem of determining which prediction is random and which is to be taken, thus having to discriminate which rows to delete. By using the maximum number of detections as a standard, it is sufficient to fill in missing rows with other tensors with fewer detections with negative values and then let the algorithm handle distances by excluding them. After the for loop on line 4, the data structure resembles an array of *num_ samples* rows and *num_ detections* columns. For each of these values, a distance matrix is calculated with all the other detections of the other samples, resulting in a matrix of matrices. Generally, the strategy adopted was to mark with negative values all associations to be excluded a priori, such as detections with different labels, associations with missing detections, and detections of the same sample among them. Once the negative values have been excluded, the algorithm proceeds to select the predictions that appear to be the same. The fundamental problem is recognizing which predictions look alike but which are always different. For similar predictions, the distance range is always less than 25 pixels, so detections that meet this requirement are considered equal and merged. It should be

noted that this solution relates to the dataset used; datasets with very large or tiny images may require a variation of this threshold. Similar detections are collected in a list, and their corresponding indices are extrapolated into the distance matrix associated with the detection under investigation. Once all of them have been collected, their average is calculated about both bounding boxes and confidence and label. Due to the very design of the algorithm, the labels of two correctly matched predictions are always the same, so averaging them is no problem.

```
def compute_distance(prediction1, prediction2):
    # distance is computed as a difference between bounding box coordinates
    if prediction1[0] == -1 and prediction2[0] == -1: # if i receive two added prediction -1
        return -1
    # if label is different set -1
    if prediction1[5] != prediction2[5]:
        return -1
    return np.linalg.norm(prediction1[:4].to("cpu") - prediction2[:4].to("cpu"), axis=0)

def compute_distance_matrix(samples_predictions, num_samples, num_detections, device):
    if not samples_predictions: # If list is empty return -1
        return -1, -1
    # Create distance matrix [num_samples, num_det, num_samples , num_det]
    distance_matrix = np.zeros((num_samples, num_detections, num_samples, num_detections))
    for i in range(num_samples):
        for j in range(num_detections):
            # For each detection in the matrix a proper distance matrix is built,
            # If a detection is missing or if it is compared to detections on the same sample,
            ↪  value set to -1
            for m in range(num_samples):
                for n in range(num_detections):
                    if len(samples_predictions[m])<num_detections:
                        missing_detections = num_detections-len(samples_predictions[m])
                        samples_predictions[m] = torch.cat((samples_predictions[m],
                        ↪  torch.full(...)))
                    if i==m:
                        distance_matrix[i, j, m, :] = -1
                    else:
                        distance_matrix[i, j, m, n] =
                        ↪  compute_distance(samples_predictions[i][j],
                        ↪  samples_predictions[m][n])
    # Return the matrix and the index with most detections
    return distance_matrix, np.argmax([torch.sum(torch.all(tensor != -1, axis=1)).to("cpu") for
    ↪  tensor in samples_predictions])
```

At the beginning of the work, it was planned to use the Hungarian Algorithm [57] to obtain a similarity projection. However, this hypothesis was not adopted, as the algorithm returns the maximum optimization of a matrix; since detections are likely to be missing rows, it was decided to construct a custom association function. The *compute_ distance_ matrix* function creates the distance matrix for a single detection. Initially, it initializes the structure, and then, for each association, it calculates the distance following the negative value strategy described above. The auxiliary

function *compute_distance* assigns a similarity score between two predictions. If the predictions are assignable, then the score is represented by the linear distance between the first four positions of the array, which are the center coordinates, the height, and the width.

Architecture changes

```
# yolov7 head
head:
[[-1, 1, SPPCSPC, [512]], # 51

[-1, 1, Conv, [256, 1, 1]],
[-1, 1, Upsample, [None,2,'nearest']],
[37, 1, Conv, [256, 1, 1]],
[[-1, -2], 1, Concat, [1]],

[-1, 1, Conv, [256, 1, 1]],
[-2, 1, Conv, [256, 1, 1]],
[ -1, 1, nn.Dropout, [ 0.25 ] ],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[[-1,-2,-3,-4,-5,-6], 1, Concat, [1]],
[-1, 1, Conv, [256, 1, 1]], # 64

[-1, 1, Conv, [128, 1, 1]],
[-1, 1, Upsample, [None,2,'nearest']],
[24, 1, Conv, [128, 1, 1]],
[[-1, -2], 1, Concat, [1]],

[-1, 1, Conv, [128, 1, 1]],
[-2, 1, Conv, [128, 1, 1]],
[ -1, 1, nn.Dropout, [ 0.25 ] ],
[-1, 1, Conv, [64, 3, 1]],
[-1, 1, Conv, [64, 3, 1]],
[-1, 1, Conv, [64, 3, 1]],
[-1, 1, Conv, [64, 3, 1]],
[[-1,-2,-3,-4,-5,-6], 1, Concat, [1]],
[-1, 1, Conv, [128, 1, 1]], # 77

-1, 1, MP, []],
-1, 1, Conv, [128, 1, 1]],
-3, 1, Conv, [128, 1, 1]],
-1, 1, Conv, [128, 3, 2]],
[-1, -3, 64], 1, Concat, [1]],
```

```
[-1, 1, Conv, [256, 1, 1]],
[-2, 1, Conv, [256, 1, 1]],
[ -1, 1, nn.Dropout, [ 0.25 ] ],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[-1, 1, Conv, [128, 3, 1]],
[[-1,-2,-3,-4,-5,-6], 1, Concat, [1]],
[-1, 1, Conv, [256, 1, 1]], # 91

[-1, 1, MP, []],
[-1, 1, Conv, [256, 1, 1]],
[-3, 1, Conv, [256, 1, 1]],
[-1, 1, Conv, [256, 3, 2]],
[[-1, -3, 51], 1, Concat, [1]],

[-1, 1, Conv, [512, 1, 1]],
[-2, 1, Conv, [512, 1, 1]],
[ -1, 1, nn.Dropout, [ 0.25 ] ],
[-1, 1, Conv, [256, 3, 1]],
[-1, 1, Conv, [256, 3, 1]],
[-1, 1, Conv, [256, 3, 1]],
[-1, 1, Conv, [256, 3, 1]],
[[-1,-2,-3,-4,-5,-6], 1, Concat, [1]],
[-1, 1, Conv, [512, 1, 1]], # 105

[77, 1, RepConv, [256, 3, 1]],
[91, 1, RepConv, [512, 3, 1]],
[105, 1, RepConv, [1024, 3, 1]],

[[106,107,108], 1, IDetect,
[nc, anchors]],
]
```

Figure 3.8: Positioning of dropout layers (DropBlock were applied in the same position)

As already stated, it is necessary to implement blocks that randomly release neurons

or regions of neurons to achieve Bayesian behavior and perform the Monte Carlo method, which has been done through dropout and DropBlock layers. First, the position where the layers must be placed must be identified. In the original paper that proposed dropout layers, by Hinton [58], dropout was used on each of the fully connected (dense) layers before the output; it was not used on the convolutional layers, but more recent research [59] has shown some value in applying dropout also to convolutional layers, although at much lower levels; in other words, there is no specific rule but only good practices. Therefore, the layers were applied on four different levels and placed after the largest convolutional layers of the head with a relatively low probability (p = 0.25), as shown in figure 3.8. YOLOv7 creates the model architecture via a configuration file. Within the file, each line corresponds to a PyTorch block or custom blocks in the internal libraries. The PyTorch *nn.Dropout* library was used to implement the dropout block.
Instead, a custom block was created to realize the DropBlock. The paper provides the algorithm for its implementation:

Algorithm 6 DropBlock

Input: output activations of a layer (A), $block\_size$, γ, $mode$

1: **if** $mode == Inference$ **then** then
2: return A
3: **end if**
4: Randomly sample mask M: $M_{i,j} \sim Bernoulli(\gamma)$
5: For each zero position $M_{i,j}$, create a spatial square mask with the center being $M_{i,j}$, the width, height being $block\_size$ and set all the values of M in the square to be zero (see Figure 2).
6: Apply the mask: $A = A \times M$
7: Normalize the features: $A = A \times \mathbf{count}(M)/\mathbf{count\_ones}(M)$

The algorithm takes as input the output activations of a previous layer A. If the model is in inference mode, the algorithm returns the original activations without any modifications. For training mode, the algorithm randomly samples a binary mask M based on a Bernoulli distribution with parameter γ, which formula is:

$$\gamma = \frac{(1 - \text{keep\_prob})}{\text{block\_size}^2} \cdot \frac{\text{feat\_size}^2}{(\text{feat\_size} - \text{block\_size} + 1)^2} \tag{3.2}$$

Subsequently, for each zero position in M, a spatial square mask is created, centered at the position and with dimensions defined by $block\_size$, and it is applied to A. Finally, the modified activations are normalized by dividing by the count of zeros in M, ensuring consistent feature scaling. Consequently, the PyTorch implementation:

```
1  class DropBlock(nn.Module):
2      def __init__(self, block_size: int, p: float = 0.5):
3          super().__init__()
4          self.block_size = block_size
5          self.p = p
6
7      def calculate_gamma(self, x: Tensor) -> float:
8          """Compute gamma, eq (1) in the paper
9          Args: x (Tensor): Input tensor
10         Returns: Tensor: gamma
11         """
12         invalid = (1 - self.p) / (self.block_size ** 2)
13         valid = (x.shape[-1] ** 2) / ((x.shape[-1] - self.block_size + 1) ** 2)
14         return invalid * valid
15
16     def forward(self, x: Tensor) -> Tensor:
17         if self.training:
18             gamma = self.calculate_gamma(x)
19             mask = torch.bernoulli(torch.ones_like(x) * gamma)
20             mask_block = 1 - F.max_pool2d(
21                 mask,
22                 kernel_size=(self.block_size, self.block_size),
23                 stride=(1, 1),
24                 padding=(self.block_size // 2, self.block_size // 2),
25             )
26             x = mask_block * x * (mask_block.numel() / mask_block.sum())
27         return x
```

The input parameters are the block size and the keep probability; unlike dropout, the probability given to the block means the one not to be dropped. The *calculate_gamma* function computes the γ variable. Then, the max pool in line 20 was adopted to create zero regions of *block_size* using a *kernel_size* set to its value and one-pixel stride. In line 26, it is normalized. The result is shown in Fig 3.9:

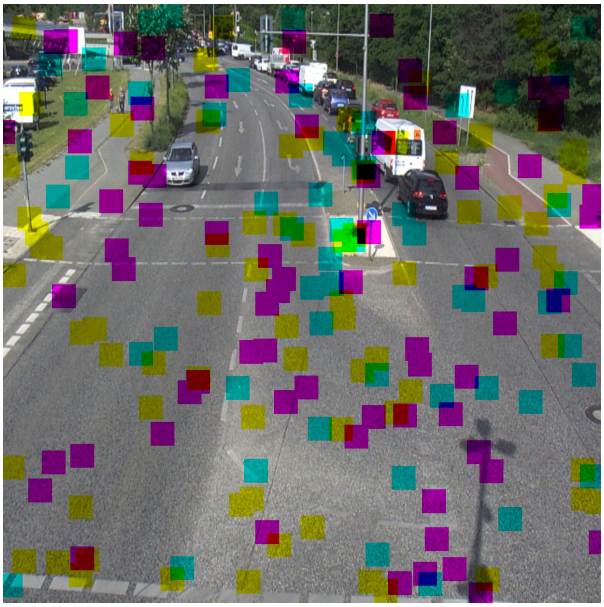


Figure 3.9: DropBlock applied to an image of the dataset

Each dropped region afflicts a single color channel; each image comprises three colors, usually Red, Green, and Blue, which is why the result of the layer applied to the image appears speckled. Before going on, the drop layers must be set in training mode so that the model behaves like a Bayesian network. To do this, they must be modulated by setting them in training mode so that even during the inference phase, the neurons will be switched off randomly.

```
def detect():
    ...
    model = attempt_load(weights, map_location=device) # set to model.eval() mode
    ...
    # set DropBlock or Dropout layers to train() to perform Monte Carlo method
    for m in model.model.modules():
        if m.__class__.__name__.startswith('DropBlock'):
            m.train()
        if m.__class__.__name__.startswith('Dropout'):
            m.train()
```

Chapter 4

Analysis of Results

Before proceeding with the analysis of the results of the models described in the 3.1 table, it is worth clarifying a few concepts. As already stated in section 2.3, the focus of this work is twofold:

1. Compare specific optimization tools' contribution to a one-stage model such as YOLOv7. The tools in focus are spatial augmentation, color augmentation, other augmentations, dropout regularization, and hard examples.

2. Attempt to achieve a workable and usable model in terms of accuracy with reduced datasets, simulating the typical use case of a developer who wants to build a situation-specific solution.

In order to perform a proper comparison, all tools are implemented and used with the same model type, YOLOv7 base, and with the same hyperparameters as described in section 3.2. The data augmentation parameters are switched off if the method is not used. The model architecture remains the same except for the dropout layers inserted according to Fig 3.8. Finally, the dataset comprises 174 training images containing five classes: Bike, Car, Lorry, Bus, and Pedestrian. Each training setup was run four times, and the result plotted is the model's average.
During the collection of the results, new challenges emerged that opened up avenues for more excellent knowledge or performance of the investigation. Consequently, the planning table was evolved to contain not only the previous models but also the best model-tools schemes, in addition, another table containing other variations will be shown. It should be noted that the latter does not bring any improvements to the preplanned models but offers insights for analysis.

Table 4.1: Results table

Model	Model setup	Precision Bike	Precision Car	Precision Lorry	Precision Bus	Precision Ped	mAP@.5	mAP@.5:.95
M0	No augmentations	0.555	0.908	0.768	0.740	0.278	0.652	0.396
M1	Augmentations: - Spatial	0.675	0.923	0.800	0.790	0.375	0.677	0.410
M2	Augmentations: - Color	0.563	0.900	0.755	0.575	0.290	0.627	0.373
M3	Augmentations: - Color, Spatial	0.643	0.915	0.785	0.715	0.315	0.655	0.392
M4	Augmentations: - Color, Spatial, - Mosaic	0.740	0.940	0.885	0.860	0.460	0.762	0.475
M4.1	Augmentations: - Spatial, Mosaic	0.735	0.940	0.890	0.888	0.473	0.767	0.475
M5	Tools: - M4.1 setup - Dropout	0.710	0.935	0.850	0.775	0.408	0.732	0.446
M6	Tools: - M5 setup - 100 HS	0.748	0.950	0.888	0.883	0.558	0.790	0.492
M6.1	Tools: - M4.1 setup - 100 HS [1]	0.773	0.950	0.893	0.858	0.598	**0.803**	**0.507**
M7	Tools: - M5 setup - 100 images [2]	0.708	0.948	0.885	0.860	0.460	0.770	0.488

[1] 100 images with highest uncertainty score
[2] 100 random images

Table 4.2: Other trained models

Model	Model setup	Precision Bike	Precision Car	Precision Lorry	Precision Bus	Precision Ped	mAP@.5	mAP@.5:.95
M4.2	Augmentations: - Mosaic	0.710	0.930	0.870	0.890	0.460	0.747	0.460
M4.3	Augmentations: - M4.1 setup - Mixup	0.750	0.940	0.870	0.840	0.480	0.768	0.474
M5.1	Tools: - M4.1 setup - DropBlock	0.695	0.930	0.820	0.855	0.395	0.723	0.439
M6.2	Tools: - M4.1 setup - 100 MAX HS [3]	0.788	0.950	0.905	0.885	0.540	0.798	0.510
M6.3	Tools: - M4.1 setup - 200 HS	0.798	0.953	0.890	0.890	0.620	0.809	0.509
M6.4	Tools: - M4.1 setup - 100 ENT [4]	0.780	0.950	0.878	0.883	0.593	0.803	0.507
M8	Tools: - M4.1 setup - All images [5]	0.870	0.960	0.920	0.890	0.690	0.878	0.582

[3]Hard samples are sorted taking the maximum uncertainty among all classes instead of averaging them all
[4]Images are sorted by entropy instead of using Monte Carlo uncertainty
[5]1084 training images, 219 validation images and just 220 test images

The tables show the results of the planned models in table 3.1, each row shows the results of the M_i model training, and each column the accuracies obtained on each class and the accuracy with IoU threshold set to 0.5 (mAP@.5) and the average mAP on the incremental thresholds from 0.5 to 0.95 (mAP@.5:.95). In order to simplify, the last two metrics will be called mAP50 and mAP95 respectively. It is good to clarify that mAP50 is not the average of previously obtained accuracies, although it is very similar. This is because the mAP50 score is calculated as the area subtended by the Precision-Recall curve. Finally, the following pages will show some important confusion matrices to fully understand how models work; since the tables were written as an average of 4 different models with the same setup, the confusion matrices correspond to those of the best model among the four attempts.

Basic model: M0

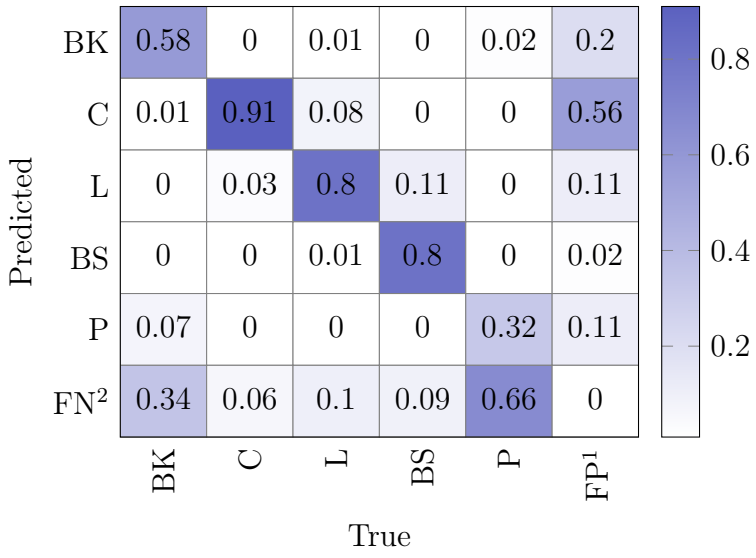
The M0 model is the starting point for all the research; it consists of the basic model of YOLOv7 trained on the dataset without any tool. It is on this that all subsequent comparisons will be made, so it is essential to understand how it performs thoroughly. The accuracy marked by the mAP50 score shows that M0 can all predict correctly at 65.2%, but the results are low for mAP95; the model achieves a score of 39.6%. Analyzing the individual classes, it can be seen that only the Car class achieves a good result with a total score of 90.8%; similarly, for the Lorry and Bus classes, it achieves 76.8% and 74.0%, while for Bike and Pedestrian, results are low, with a score of 55.5% and 27.8%. These results already show some peculiarities.

Referring to 3.2, the result obtained on the Car class is predictable since there are numerous occurrences of the same. However, it is enough to dwell on the others to understand some peculiarities of the dataset. The second class with the highest number of occurrences is Bike, but it is not the one with the second highest accuracy; it is in the second to last position, coming after both Car and Lorry.

Similarly, for the Pedestrian class, although quantitatively superior to Bus, M0 has a worse score than Bus. This characteristic is mainly due to the size of the objects in the scene. The bus class is made up of instances that occupy a more significant part of the image than the others, so M0 easily recognizes and extracts features belonging to that class; furthermore, there are no objects in the scene similar to it, for example, an instance of Car can be misidentified with an instance of Lorry. There can be misclassifications between Pedestrian and Bike, but nothing similar to Bus. However, Bus's size and color are different, and it has widely distinguishable features. This makes Bus a class that is easily recognizable even though there are fewer occurrences in the dataset, as in the real world. On the contrary, Pedestrian instances are misclassified because they are small objects in the scene and because they are very similar to Bike instances, making it difficult

for M0 to correctly distinguish an instance of one from an instance of the other. In fact, when analyzing the confusion matrix of the best M0 model, the following result is obtained:

Figure 4.1: M0 Confusion Matrix



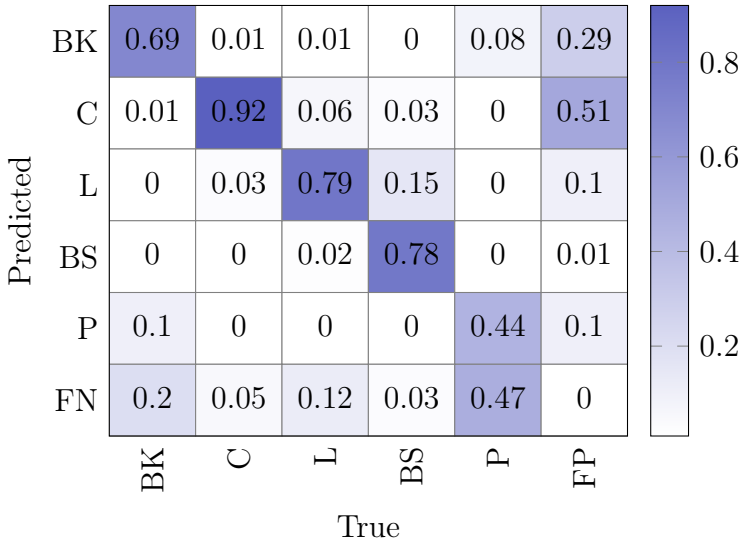
False negatives for the Car and Bus classes are low but still significant, and even higher for the Lorry class since false negatives are at 10%. M0 misidentifies the Bus as Lorry with a probability of 11%, and the latter is misinterpreted as Car with a probability of 8%. It is a different matter for the classes on the fringes, which are hardly mistaken for each other but often go undetected; in fact, about 7% of instances of the Bike class are mistaken for Pedestrian, and 34% are not recognized, on the other hand only 2% of instances of Pedestrian are mistaken for Bike but about 66% of them are false negatives. Concerning false positives, the model hardly mispredicts a Bus instance, whereas all other classes exceed 10%, especially Car, which has a score of 56%. However, the False Positives column remains the easiest to limit since, often, it is enough to increase the confidence threshold to lower them significantly. To summarize, from the confusion matrix, it can be seen that M0's performance is significantly linked to the internal structure of the dataset; in particular, M0 struggles to recognize small objects such as Car and Pedestrian. It also presents some difficulties for the other classes, but all in all on the result obtained is good.

[1]Background False Positive: the model predicts background as an object
[2]Background False Negative: the model predicts an object as background

Data augmentation: M1, M2, M3

Let us now consider models incorporating data augmentation tools. M1 implements spatial augmentation, M2 color augmentation, and M3 combines the two. A different and specific analysis regarding M4, M4.1, M4.2, and M4.3 will follow. In the development stages of the M4 model, it was seen that the contribution of the up-and-down flipping transformation was negative, decreasing performance. It is no coincidence that in the default hyperparameter configuration file of YOLOv7, this parameter was kept at 0, so in subsequent models, the *flipud* variable will also be set at 0. The same goes for the *shear* and *perspective* parameters. Spatial augmentation adds significantly to M1's performance; mAP50 is increased by 2.5% and mAP95 by 1.4%, but it is by analyzing the individual classes' precision that the tool's actual contribution is revealed. The middle classes grow in their score, which was already high previously, in particular, they grow by 1.5% for Car, 3.2% for Lorry and 5% for Bus respectively, but the most surprising result is for the other two whose increase settles at 12% for Bike and 9.7% for Pedestrian. Going to check the confusion matrix:

Figure 4.2: M1 Confusion Matrix



It is evident that the techniques applied cause an increase in misclassifications between classes. The clearest example is the Bus column, which shows that its instances are mistaken for Lorry and Car. However, it also shows a large contribution to small classes and, in general, a decrease in false negatives; part of this decrease is spread over erroneous detections. In conclusion, M1 has learned to recognize objects better than M0, but it tends slightly more to confuse them with each other.

M2's results are remarkably interesting. The model unexpectedly worsens its performance, with a decrease of 2.5% in mAP50 and 2.3% in mAP95. Although the accuracy trend on

the other middle classes remains about the same, the Bus column sinks, and likewise also the Bike and Pedestrian columns. This fact has a root. Going to retrieve the training set images after color augmentation has been applied it gives the following result for some images:

Figure 4.3: Training image after color augmentation

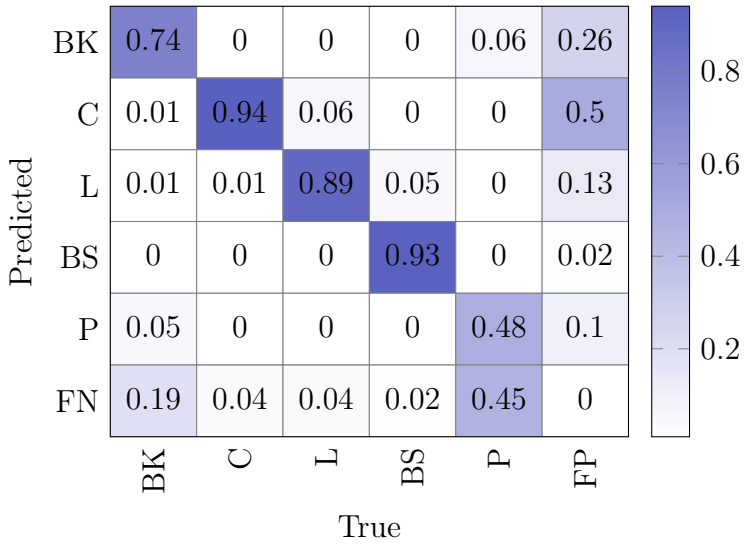
The image in Fig 4.3 is distorted in color and there is now a kind of flash in the center of the image that makes object detection impossible, or at least greatly hindered. This phenomenon has occurred for multiple images, even in night contexts made even darker. As a result, object features are sometimes hidden while its label still appears. Thus, the dataset now has images that are difficult to recognize and real poisoned images, which in such a small dataset decreases the performance of M2 even if present in a modest amount. This is why, after making the final verification by going to see the results of the M3 model, it was decided to remove color augmentation from subsequent studies. As can be seen from the table, M3 incorporates, on the one hand, the improvement of spatial augmentation and, on the other hand, the obstacle of color augmentation, still resulting in a worse model than M1; its score decreases by 2.2% in mAP50 and 1.8% in mAP95.

Mosaic augmentation: M4, M4.1, M4.2 and M4.3

After noting the failure of color augmentation, the models inherent in mosaic augmentation will be analyzed. M4 still uses color augmentation, but there is already a glimpse of the remarkable contribution of mosaic augmentation to the model. M4 outperforms the base model by 11 percentage points in mAP50 and by 7.9 percentage points in mAP95; the same offset is maintained with M3, which had almost identical results to M0; in addition, all classes increased their score. This already testifies to the importance of this tool. By implementing it stand-alone, M4.2 achieves a score of 74.7% in mAP50 and 46% in mAP95, with class accuracies of 71%, 93%, 87%, 89%, and 46%, respectively. The single contribution of mosaic augmentation significantly surpasses that offered by the other data

augmentation techniques; M4.2 outperforms M1 by 7 points and M2 by 12 points, the latter result being irrelevant since M2´s contribution was negative. Furthermore, model M4.3 showed that the contribution of mixup augmentation is almost irrelevant, so it was excluded from further study. Therefore, it is clear that the best solution for combining data augmentation tools is the combination of spatial and mosaic; this setup was added to the main table as it proved to be the best base model case. Let us analyze M4.1. This model represents the most important version of the entire table; although it is not the best in accuracy, it provides the best possible configuration among data augmentation techniques. Therefore, it is used as the basis for all the models to follow. M4.1 achieves a total accuracy that is already high enough to be used in real scenarios; the model has a mAP50 of 76.7% and a mAP95 of 47.5%, which increases the performance of M0 by 11.5 and 7.9 points. The confusion matrix is as follows:

Figure 4.4: M4.1 Confusion Matrix



M4.1 surpasses M0 in virtually all results, each class increasing by 16%, 3%, 9%, 13%, and 16%, respectively. These increases are noteworthy because they benefit from the decrease in false negatives, meaning the model better recognizes class instances. The model predicts the middle classes very well, exchanging only the instances of Car with Lorry at 1%, Lorry with Car at 6%, and Bus with Lorry at 5%. At the same time, their respective false negatives settle at 4%, 4%, and 2%. The instances of Bike are exchanged with Car and Lorry at 1% and with Pedestrian at 5%, while those of Pedestrian with Bike at 6%. However, false negatives still remain problematic for these classes, making up 19% for Bike and 45% for Pedestrian, respectively. To summarize, M4.1 achieves significant improvements and turns out to be the best model in the scenario of data augmentation techniques alone. Significant issues remain for the Bike and Pedestrian classes, whose main

problem is false negatives. Finally, the model will be used as the basic setup for subsequent ones, forming a solid foundation with excellent results.

Drop models: M5 and M5.1

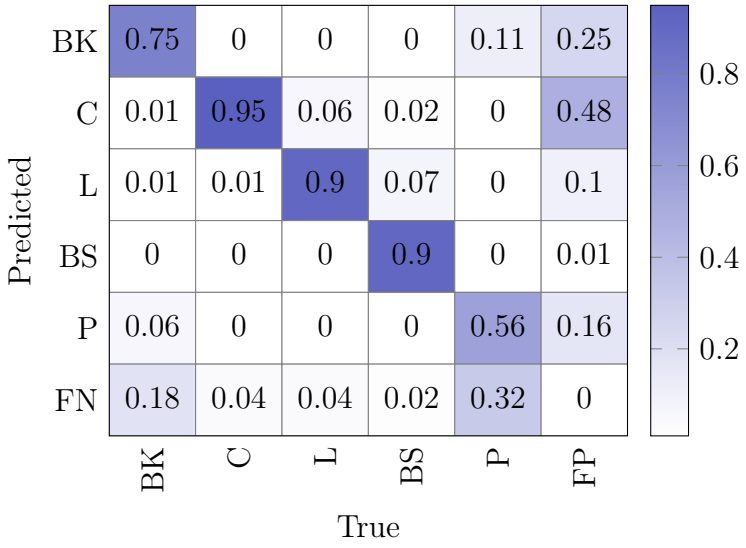
The M5 and M5.1 models correspond to those with different architecture because of the addition of Dropouts or DropBlock layers. The results of M5 show that by applying Dropout layers to the model, it suffers in terms of accuracy; in fact, comparing it to its counterpart without Dropout (M4.1), the mAP50 decreases by 3.5% and the mAP95 by 2.9%, this shows that the layers do not offer a positive contribution. This result is not unexpected. Indeed, they are helpful when it comes to reducing overfitting, which in a well-built model like YOLOv7 is not an issue, or at least not with the dataset in use. If there is no problem of overfitting, then the contribution of the tool is null as far as training performance is concerned; it was instead valuable in the calculation of hard mining samples. The same goes for the implementation of DropBlock layers; the performance of M5.1 even worsens that of M5 by 0.9% and 0.7%. As a result, the DropBlock layer will no longer be used in later models, but Dropout will be employed where needed. It is possible that, for both M5 and M5.1, an ideal configuration has not been found. The factors that come into play in these models are the position of the layers, the number of layers, the associated drop percentage, and the size of the visual blocks for DropBlocks. Since there are no specific rules, as written in section 3.4.1, it would have taken too long to look for an optimal solution empirically.

Contribution of Hard Samples: M6 and M7

How the hard samples were obtained has already been described in section 3.4.1. At the beginning of the work, a problem was encountered: when analyzing the images with the highest uncertainty, they were found to be without objects. This happened because some of the images in the initial dataset were frames of a video shot by a UAV drone, therefore, it also had examples without objects. The images had not all been supervised since there were about 1600 of them, so after the uncertainty quantification, the images were removed, and the dataset and the models that had been trained up to that point were trained again. This is important because it shows how the uncertainty calculation highlights the images that are most difficult to recognize and also allows us to check whether there are inadequate images. Thus, once the list of images with the associated uncertainty had been recalculated, finding night images or images with poor visibility among the first images in the list was standard, as these constitute the set of corner cases in the dataset. Various approaches regarding the computation of uncertainty to select hard samples will be explored later. Then, the first 100 images with the highest uncertainty were added to the dataset, and as

a result, the training set is now made up of 274 images. A parallel dataset was also made with 100 random images that were added to the original ones to have an excellent cue for comparing the models and thus analyze the effective contribution of the hard samples. M6 achieves an increase of 13.8% and 9.6% percent over M0 and 2.3% percent and 1.7% percent over M4.1, which until now was the best-performing model. The comparison with M7 is of remarkable importance; the two models are the only ones with a larger dataset than the others. M6 overtakes M7 in all scores, the mAP50 is 2 percentage points higher, while the results obtained for the Bike and Pedestrian classes are significant, increasing by 4% and 9.8% respectively, the latter result brings M6´s score on the Pedestrian class to 55.8%, the first time a model has achieved a result largely above the 50% threshold.

Figure 4.5: M6 Confusion Matrix

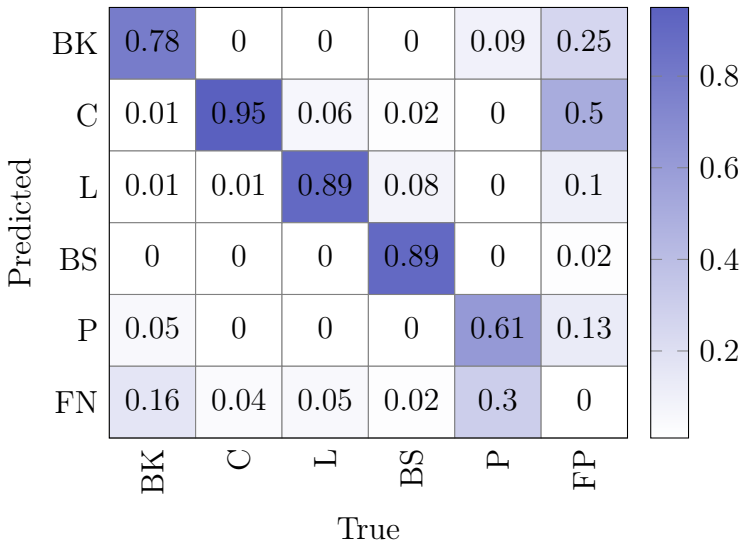


However, going to compare the matrix of M6 with that of M4.1 shows that the situation has mostly stayed the same. It is important to note that M4.1´s results were already good, so the more a model reaches a ceiling of accuracy, the harder it is to improve it. There are positive changes in general; Bike, Car, and Lorry precision increases by 1 point percentage, but the Bus precision decreases by 3 points. The latter decrease is actual but accentuated by randomness; comparing the average of the accuracies of the four attempts of each model, the accuracy of M4.1 for the Bus class is only 0.5% higher than M6. The accuracy of the Pedestrian class is noticeably improved with an increase of 8% even though it is more mistaken for Bike.

The best solution: M6.1

Considering all that has been revealed through the past few models has led to the definition of the best solution. Specifically, data augmentation models highlighted that the combination of spatial and mosaic without color augmentation was the best; drop models showed that drop layers worsen network performance in this case study. Finally, the M6 model demonstrated the relevance of hard samples. As a result, the M6.1 takes up what worked successfully in previous models. M6.1 achieves the highest possible score; subsequently, better scores will be achieved in later models. However, this result will only be reached by increasing the number of dataset images; therefore, within this case study, the model has achieved maximum performance. The model broke the wall of 80% in mAP50 and 50% in mAP95, achieving a total accuracy of 80.3% and 50.7%. It improves the performance of M0 by 15.1 and 11.1 points and by 1.3 and 1.5 points over the previous M6. The average precision for each class are 77.3% for Bike, 95% for Car, 89.3% for Lorry, 85.8% for Bus, and 59.8% for Pedestrian; the latter achieves an extraordinary score considering that M0´s accuracy in the class was 27.8%.

Figure 4.6: M6.1 Confusion Matrix



The trend in the confusion matrix is the same as seen in the previous ones: the false negatives of the intermediate classes have reached an upper bound, so if they improve by a few percentage points, it is due more to the randomness than to the model itself. On the other hand, false negatives of the side classes decrease by a few percentage points; in particular, now the Bike class has an accuracy close to 80%, and the Pedestrian class has twice the accuracy of the corresponding false negative.

Experiments on Hard Samples: M6.2, M6.3 and M6.4

M6.1 has been hailed as the best model due to its composition of the optimal configuration using available tools, resulting in the highest average accuracy and performance across individual classes. Afterward, further experiments were conducted to identify the most valuable hard samples—those most meaningful and, consequently, most effective during the training phase. This research is crucial as it enables the identification of images, leading to the most substantial performance improvement, which is the most challenging to recognize. Indeed, correctly detecting these hard samples can provide a reduced set of examples that are significantly more impactful for the model than other easy and standard instances. This approach increases the model's accuracy with a modest number of examples, which are also easier and quicker to document for a study tailored to a specific situation. So far, hard samples have been collected by calculating and averaging the entropies of each prediction derived from the mean of inferences obtained through the Monte Carlo method. This way, images are identified based on their predictions with a high average entropy. There is no discrimination between an image containing hundreds of objects and one containing only one; what matters is whether the average entropy of the inference is high or not. The models M6.2 and M6.4 investigate this topic; moreover, M6.3 represents an evolution of M6.1, added 100 ulterior hard samples to the model for a total of 200 to the aim to verify how much the precision increases by adding another unit of samples.

M6.3 improves minimally compared to M6.1, with an increase of 0.6% on mAP50 and 0.2% on mAP95; in individual classes, the increases are slight, the most clear are 2.5 and 2.2 points for Bike and Pedestrian. This result can have two explanations; the first is that the dataset used does not have so many hard samples, and therefore, the addition of other 100 images among the most uncertain does not necessarily mean adding hard samples containing corner cases; the second is that the model has already learned the best from the first 100 samples that adding the new ones hardly increases the result obtained with M6.1. Both explanations can be possible even together.

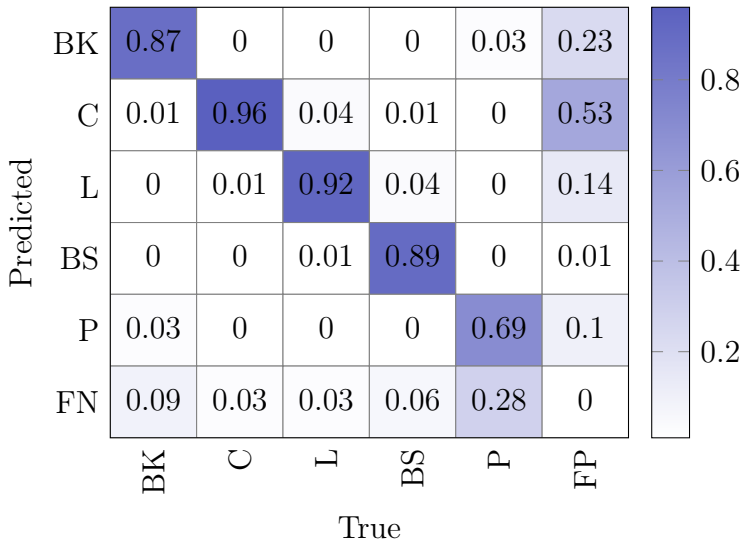
By analyzing the models M6.2 and M6.3, the practices used look for different techniques for calculating uncertainty. M6.2 calculates uncertainty differently. After averaging predictions by the Monte Carlo method, rather than calculating uncertainty as the mean of entropies, it computes the score by taking the maximum value between the entropies of the prediction. Therefore, if there are many simple cases in an image, but one of them is remarkably uncertain, the image score is taken as the final value of the latter. This technique focuses on individual objects in the image rather than the image in general. The result needs to clarify whether this approach is better than the standard one; in fact, M6.2 gets a lower score of 0.5% in mAP50 and higher than 0.2% compared to M6.1. The accuracy on individual classes is slightly increasing except for the Pedestrian class, which drops by 5.8 percentage points. Since the models have generally reached a maximum threshold for

the accuracy of the Car, Lorry and Bus classes, although M6.3 increases their accuracy by a little, the decrease obtained on the Pedestrian class does not suggest this method.
M6.4 focuses on the possible removal of the Monte Carlo method in favor of using simple entropy. Calculating uncertainty with the Monte Carlo method is time-consuming; each image must be processed as often as indicated by the number of samples, which can vary from 10 to 100. For example, in this case study, the model took about a day to process just over a thousand images. In addition, Dropout or DropBlock layers must be implemented to use the method, and this research showed that they worsen performance; furthermore, changing the architecture of the model denies the use of transfer learning and the loading of pretrained weights.
Thus, simple entropy was used to calculate the uncertainty of images. The results showed that 92 out of 100 images corresponded to those generated by the Monte Carlo method. Analyzing the performance, M6.3 achieves the same results as M6.1 in absolute precision terms. There are some fluctuations in the precision of classes that are due to randomness. This result is of great importance because it shows that hard samples can be extracted by calculating entropy; this simplifies the work, both in terms of time, since the calculation of entropy is much faster than the application of the Monte Carlo method, and in terms of the complexity of the model, since drop layers do not have to be implemented.

The upper bound: M8

The last model that has been trained is M8; it constitutes the maximum peak of performance; in fact, it has been trained with 80% of the whole dataset for a total of 1084 training images. M8 identifies the maximum threshold beyond which none of the models under consideration can go and allows us to observe how far the results obtained are from the model. The model achieves a total score of 87.8% in mAP50 and 58.2% in mAP95, and a score by class of 87% on Bike, 96% on Car, 92% on Lorry, 89% on Bus and 69% on Pedestrian. It surpasses the best model, M6.1, by 7.5 percentage points on both mAP50 and mAP95. Looking at the differences in the individual classes, we can see that the central classes are not far from the optimum; indeed, for the Car class, the offset is only one percentage point, while for Lorry, it is 2.7%, and for Bus, it is 3.2%. The biggest distance is undoubtedly for the other two classes, the one on Bike is 9.7% and on Pedestrian is 9.2%. This difference in precision levels remains remarkable but simultaneously makes it clear that M6.1, with only 274 training images, can compete remotely with M8.

Figure 4.7: M8 Confusion Matrix



The confusion matrix improves the one of M6.1, decreasing the number of false negatives and increasing the general precision. In general, it is a great effort to bring down below the threshold of 10% the number of false negatives for Bike and under that of 30% the false negatives for Pedestrian. The latter is obviously the class with the most problems, and it is clear that these issues are also present as the dataset increases.

Chapter 5

Conclusion

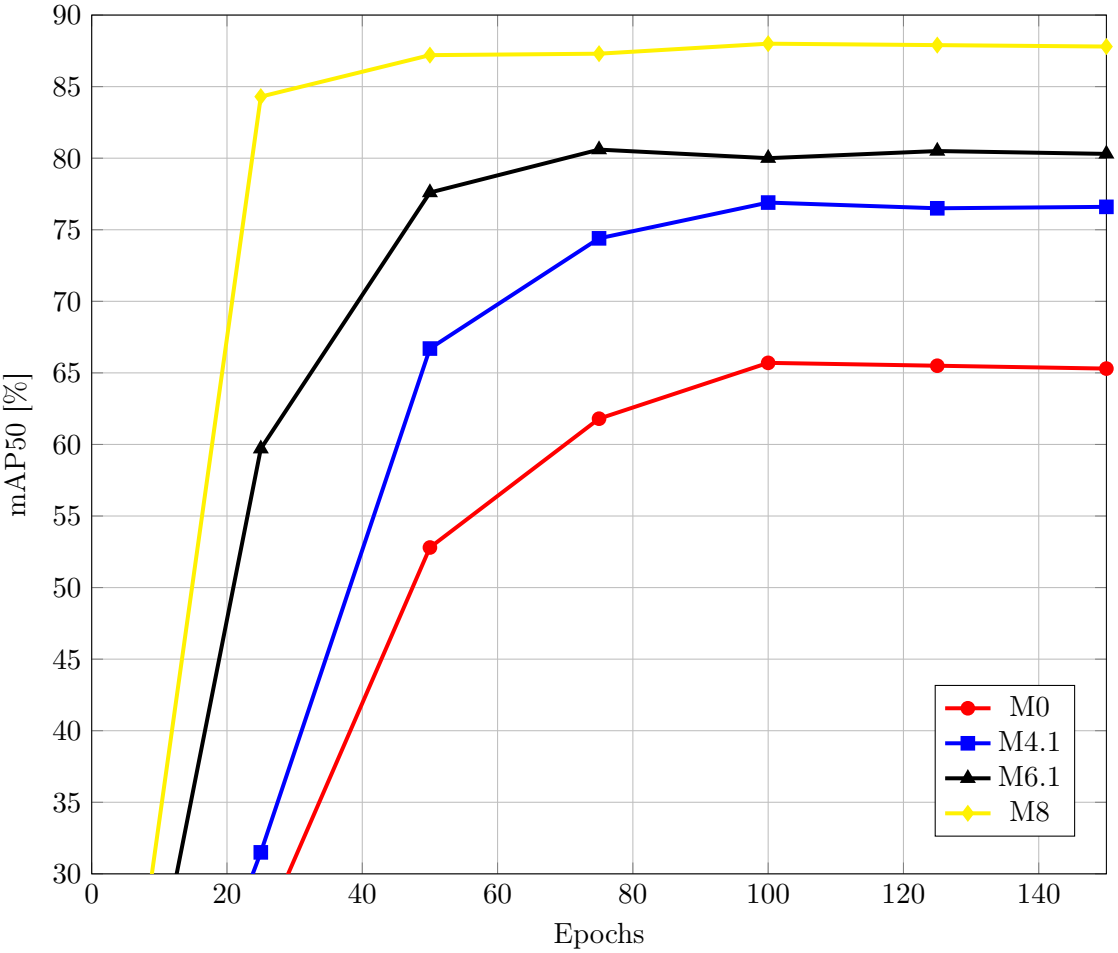
The world of artificial intelligence has revolutionized human lives, not only from an industrial point of view. In the context of computer vision, convolutional neural networks are leading the way; through their ability to extract features and generalize them, they are able to outperform any other type of feature extraction algorithm, becoming the main solution for object detection. Over the past decade, several solutions have been investigated, each time surpassing the other, starting with the simplest two-stage models and arriving at the development of YOLOv7, a one-stage model capable of delivering impressive performance in precision and speed and even if nets development is still growing, this model represents a cornerstone. Parallel to the development of neural networks, solutions have been discovered to improve their performance; among these are data augmentation, dropout regularization, and hard samples mining. Their contribution to training performance was analyzed in this thesis work, applying them to YOLOv7 in order to measure their contribution to model accuracy. In addition, the dataset used was intended to simulate the scenario of a developer who wants to build a solution tailored to a specific situation, therefore, with a small number of images of the dataset in different contexts. The ground truth data contained five classes with different frequencies; the classes were Bike, Car, Lorry, Bus, and Pedestrian, with a large number of occurrences of Car and Bike, a small number of occurrences of Bus, and medium for the other classes. A total of 174 training images were used, to which a further 100 images were added for hard samples experiments. Initially, the work was to be done entirely on a company framework called DeeBob. Unfortunately, the lack of custom tools for data augmentation made it impossible to continue the work, so an implementation entirely with the official YOLOv7 script was adopted. The work resulted in the production of 17 models, each of which implements different tools, often combining them; the results can be seen in the two 4.1 and 4.2 tables. The first table shows the main results, which were planned in table 3.1, and the second encapsulates ulterior experiments conducted to gain a better knowledge of the tools. The implementation of data augmentations was already internal to YOLOv7; therefore, those were used for the drop layers PyTorch Dropout layers

and also DropBlock layers were implemented; the models containing them have been used for the collection of hard examples through the Monte Carlo method. Among the most significant models, there are those represented below:

Model	mAP@.5	mAP@.5:.95		
M0	65.2%	39.6%		-22.6%
M4.1	76.7%	47.5%	+11.5%	-11.1%
M6.1	80.3%	50.7%	+15.1%	-7.5%
M8	87.8%	58.2%	+22.6%	

M0 is the basic model, has no active tool, and is the lower bound for all subsequent models. M4.1 is the best possible configuration for data augmentation, while M6.1 is the best model. It has the same setup as M4.1 with the addition of 100 of the best hard samples. Finally, M8 is the best model ever; it has the same configuration as M4.1 but uses all the dataset images that are just under 1100; in other words, it constitutes the upper bound of all the models. The offsets between the table models and the upper and lower limits are marked in red and green.
In addition, in the development of the solution presented, other important information emerged, they are:

- Color augmentation makes some images spurious and decreases performance.
- Flip up-down, perspective, and mixup augmentations are either performance or ineffective.
- The Dropout and DropBlock layers decrease the model's effectiveness, DropBlock is even worse than Dropout.
- 100 hard examples are more effective than 100 random images.
- Adding another 100 hard samples to the dataset does not get the same increase but an increase of only 16% compared to the first.
- Calculating uncertainty by taking the maximum uncertainty between the individual predictions of an inference rather than calculating it by averaging the uncertainties does not help to find better hard samples or increase the model's effectiveness.
- Calculating the entropy of the prediction is a good way to calculate the uncertainty; in fact, without using the Monte Carlo method, the first 100 hard samples are 92% the same ones.



To conclude, this thesis work has demonstrated the effectiveness or ineffectiveness of the tools to optimize the training performance of YOLOv7; in addition, it was made clear that there are solutions to achieve adequate performance even with an extremely small number of images for training. The M4.1 model shows an 11.5% improvement over its tool-less counterpart M0, while M6.1, which has 100 additional hard samples, surpasses the 80% threshold in mAP50, achieving a score of 80.3%. These findings indicate that these models can be effectively used in real-world scenarios within the dataset domain.

Bibliography

[1] Jiageng Mao, Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. 3d object detection for autonomous driving: A comprehensive survey, 2023.

[2] Omer Wosner, Guy Farjon, and Aharon Bar-Hillel. Object detection in agricultural contexts: A multiple resolution benchmark and comparison to human. *Computers and Electronics in Agriculture*, 189:106404, 2021.

[3] Pu Li and Wangda Zhao. Image fire detection algorithms based on convolutional neural networks. *Case Studies in Thermal Engineering*, 19:100625, 03 2020.

[4] Shaoxiong Ji, Shirui Pan, Xue Li, Erik Cambria, Guodong Long, and Zi Huang. Suicidal ideation detection: A review of machine learning methods and applications. *CoRR*, abs/1910.12611, 2019.

[5] Xiangrong Zhou, Kuzuma Yamada, Ryosuke Takayama, Xinxin Zhou, Takeshi Hara, Hiroshi Fujita, Song Wang, and Takuya Kojima. Performance evaluation of 2d and 3d deep learning approaches for automatic segmentation of multiple organs on ct images. In *Medical Imaging 2018: Computer-Aided Diagnosis*, page 83, 02 2018.

[6] Sangeeta Rani, Bhupesh Kumar Singh, Deepika Koundal, and Vijay Anant Athavale. Localization of stroke lesion in mri images using object detection techniques: A comprehensive review. *Neuroscience Informatics*, 2(3):100070, 2022.

[7] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[8] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346–359, 2008.

[9] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, 2005.

[10] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3), 1995.

[11] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[12] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

[13] Jasper Uijlings, K. Sande, T. Gevers, and A.W.M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104:154–171, 09 2013.

[14] Ross Girshick. Fast R-CNN. *IEEE International Conference on Computer Vision (ICCV)*, 2015.

[15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Trans Pattern Anal Mach Intell*, 2016.

[16] Ross B. Girshick Joseph Redmon, Santosh Kumar Divvala and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[17] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[18] ultralytics. Yolov8. `https://github.com/ultralytics/ultralytics`, 2022.

[19] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2022.

[20] Chien-Yao Wang, Hong-Yuan Mark Liao, and I-Hau Yeh. Designing network design strategies through gradient path analysis, 2022.

[21] Zhi Zhang, Tong He, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of freebies for training object detection neural networks. *CoRR*, abs/1902.04103, 2019.

[22] Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding and Jian Sun. Repvgg: Making vgg-style convnets great again. *CoRR*, 2021.

[23] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[25] Aram Galstyan and Paul R. Cohen. Empirical comparison of "hard" and "soft" label propagation for relational classification. In Hendrik Blockeel, Jan Ramon, Jude Shavlik, and Prasad Tadepalli, editors, *Inductive Logic Programming*. Springer Berlin Heidelberg, 2008.

[26] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. `https://github.com/WongKinYiu/yolov7`, 2022.

[27] Paszke et al. Pytorch. In *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, 2019.

[28] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[29] Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *CoRR*, abs/1710.09412, 2017.

[30] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014.

[31] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of tricks for image classification with convolutional neural networks. *CoRR*, 2018.

[32] Abhinav Shrivastava, Abhinav Gupta, and Ross B. Girshick. Training region-based object detectors with online hard example mining. *CoRR*, 2016.

[33] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6, 2019.

[34] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *CoRR*, 2017.

[35] Luke Taylor and Geoff Nitschke. Improving deep learning using generic data augmentation. *CoRR*, 2017.

[36] Brahnam S. Nanni L., Paci M. and Lumini A. Comparison of different image data augmentation approaches. *Journal of Imaging*, 2021.

[37] Sungheon Park and Nojun Kwak. Analysis on the dropout effect in convolutional neural networks. *Asian Conference on Computer Vision*, 2016.

[38] Aybora Koksal, Onder Tuzcuoglu, Kutalmis Gokalp Ince, Yoldas Ataseven, and A. Aydin Alatan. Improved hard example mining approach for single shot object detectors. *Computer Vision and Pattern Recognition*, 2022.

[39] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

[40] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[41] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[42] Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, and Jun-Wei Hsieh. Cspnet: A new backbone that can enhance learning capability of CNN. *CoRR*, 2019.

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *CoRR*, 2014.

[44] Chien-Yao Wang, I-Hau Yeh, and Hong-Yuan Mark Liao. You only learn one representation: Unified network for multiple tasks. *CoRR*, abs/2105.04206, 2021.

[45] Terrance Devries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout. *CoRR*, abs/1708.04552, 2017.

[46] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. *CoRR*, abs/1412.1897, 2014.

[47] Moloud Abdar, Farhad Pourpanah, Sadiq Hussain, Dana Rezazadegan, Li Liu, Mohammad Ghavamzadeh, Paul W. Fieguth, Xiaochun Cao, Abbas Khosravi, U. Rajendra Acharya, Vladimir Makarenkov, and Saeid Nahavandi. A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *CoRR*, abs/2011.06225, 2020.

[48] Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? *CoRR*, abs/1703.04977, 2017.

[49] Weizhu Qian, Dalin Zhang, Yan Zhao, Kai Zheng, and James J. Q. Yu. Uncertainty quantification for traffic forecasting: A unified approach, 2022.

[50] Liang Peng, Hong Wang, and Jun Li. Uncertainty evaluation of object detection algorithms for autonomous vehicles. *Automotive Innovation*, 4:241 – 252, 2021.

[51] Jiaxi Wu, Jiaxin Chen, and Di Huang. Entropy-based active learning for object detection with progressive diversity constraint. *Computer Vision and Pattern Recognition*, 2022.

[52] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 1948.

[53] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. *International Conference on Machine Learning*, 2016.

[54] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V. Le. Dropblock: A regularization method for convolutional networks. *CoRR*, abs/1810.12890, 2018.

[55] Kumari Deepshikha, Sai Harsha Yelleni, P. K. Srijith, and C. Krishna Mohan. Monte carlo dropblock for modelling uncertainty in object detection. *CoRR*, abs/2108.03614, 2021.

[56] Joshua Niemeijer, Federico Battistella, Gurucharan Srinivas, and Andreas Leich. An approach for fusing two training-datasets with partially overlapping classes. In *2023 IEEE 17th International Conference on Semantic Computing (ICSC)*, 2023.

[57] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[58] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[59] Sungheon Park and Nojun Kwak. Analysis on the dropout effect in convolutional neural networks. In *Asian Conference on Computer Vision*, pages 189–204, 03 2017.