PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

ANDREY DE AGUIAR SALVI

CONVOLUTIONAL NEURAL NETWORKS COMPRESSION FOR OBJECT DETECTION

Porto Alegre
2021

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

CONVOLUTIONAL NEURAL NETWORKS COMPRESSION FOR OBJECT DETECTION

ANDREY DE AGUIAR SALVI

Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Rodrigo Coelho Barros

Porto Alegre
2021

Ficha Catalográfica

S184c Salvi, Andrey de Aguiar

Convolutional Neural Networks Compression for Object Detection / Andrey de Aguiar Salvi. – 2021.
106.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Rodrigo Coelho Barros.

1. Deep Learning. 2. Object Detection. 3. Model Compression. 4. YOLOv3. I. Barros, Rodrigo Coelho. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

Andrey de Aguiar Salvi

Convolutional Neural Networks Compression for Object Detection

This Master Thesis/Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Doctor/Master of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on March 22, 2021.

COMMITTEE MEMBERS:

Prof. Dr. Mauro Roisenberg (UFSC)

Prof. Dr. Duncan Dubugras Alcoba Ruiz (PPGCC/PUCRS)

Prof. Dr. Rodrigo Coelho Barros (PPGCC/PUCRS - Advisor)

Dedicate this work to my parents.

"As it says on the book, we are blessed, and cursed.
(...) Same things make us laugh, make us cry."
(Big Smoke)

ACKNOWLEDGMENTS

First, I thank all my undergraduate teachers which helped me to reach this stage, especially: Vilson Heck Júnior, who introduces me to the Computer Vision area and had great patience being my advisor on our research project; Ailton Durigon, who was always enthusiastic about my doubts and interests in Calculus and gave me a small part of his immense knowledge, which was necessary for me to get where I am; and Wilson Castello Branco Neto, who was my AI teacher, mentor on my first steps on Neural Networks, and who guided me to chose and start a master's degree.

Second, I thank the special persons from GPIN, especially: Rodrigo Coelho Barros, my advisor, for giving me the opportunity to be here, learning about Deep Learning (a term that I did not even know before I started my master's). His Deep Neural Networks discipline were the best classes I have ever had; Nathan Schneider Gavenski, my friend and master's colleague, who I always admired for his skills, and who taught me what is a strong dedication to achieve a purpose; Eduardo Henrique Pais Pooch, my friend and master's colleague, for the jokes, especially in the most work stressful moments, and also for the help in writing my first works, which I had some difficulty; Felipe Roque Tasionero, my friend and master's colleague, also for the jokes, for hearing my outbursts, and for the advice on my master's degree and especially on life; and Alessandra Helena Jandrey, for your advice, the jokes, your care with me, and by the more than special person who you are.

Third, I thank for HP Inc. that gave us all the necessary resources to this project could be carried out. Also for my colleagues from LIS, especially Luana Müller and Anderson da Silva, for your support reaching my first job, for the moments of laughter and relaxation, and most importantly, for friendship.

And by the last, but not the least, I thank my family, for the unconditional support in all steps of life. There is a long time ago I started this journey, and you helped me at each stage, regardless of my choice made. Part of this achievement is yours.

COMPRESSÃO DE REDES NEURAIS CONVOLUCIONAIS PARA DETECÇÃO DE OBJETOS

RESUMO

Aprendizado Profundo é o estado da arte em tarefas de Visão Computacional, tais como Classificação de Imagens, Detecção de Objetos, Segmentação de Instâncias, Geração de Conteúdo, entre outros. Ao longo do tempo, os modelos se tornaram maiores, mais profundos, e de maior acurácia, mas também super-parametrizados, pesados e lentos, dificultando o uso de tais modelos em automação de processos em dispositivos limitados, com poder de processamento reduzido, memória, ou energia. Consequentemente, a Compressão de Modelos emergiu na literatura para reduzir o tamanho do modelo e o custo de processamento o máximo possível, impactando o mínimo possível na performance do modelo na tarefa alvo. Embora existam muitos estudos de compressão de modelos na literatura versando sobre diferentes abordagens, existem poucos estudos trazendo comparações práticas entre diferentes abordagens, e nenhum deles com o foco em Detecção de Objetos. Portanto, este trabalho contribui à literatura ao comparar e explorar os *trade-offs* existentes entre *Pruning*, *Knowledge Distillation (KD)*, *Neural Architecture Search (NAS)*, e uma reconstrução de modelo baseada em convoluções eficientes. Para alcançar tal objetivo, modelos baseados na YOLOv3 foram treinados com a mesma estratégia de *data-augmentation* em dois conjuntos de dados, PASCAL VOC e *Exclusively Dark Images*, e avaliados de acordo com *Mean Average Precision*, número de parâmetros, tamanho de armazenamento, e *Multiply-Accumulate Operation (MAC)*. Os resultados mostram que um *Pruning* mais agressivo foi capaz de gerar o melhor *trade-off*, onde o seu mAP ultrapassou a abordagem de NAS + KD, além de produzir um modelo com o menor número de parâmetros e com a maior redução efetiva em MACs.

Palavras-Chave: Aprendizado Profundo, Detecção de Objetos, YOLOv3, Compressão de Modelos, Poda, Destilação de Conhecimento, Pesquisa por Arquiteturas Neurais, Convoluções Eficientes.

CONVOLUTIONAL NEURAL NETWORKS COMPRESSION FOR OBJECT DETECTION

ABSTRACT

Deep Learning (DL) is the state-of-the-art in Computer Vision tasks, such as Image Classification, Object Detection, Instance Segmentation, Content Generation, among others. Over time, the models have become broader, deeper, and more accurate, but also hyper-parameterized, heavier, and slower, making their use harder for automating tasks based on constrained devices, such as those with reduced processing power, or with memory or energy consumption constraints. Consequently, Model Compression emerges in the literature to reduce the model's size and processing cost as much as possible, while impacting as little as possible in the model's performance within its target task. Although there are many model compression studies in the literature exploring several different approaches, there are few studies in the literature bringing practical comparisons between different approaches and none of those focusing on Object Detection. Therefore, this work contributes to the literature by comparing and exploring the existing trade-offs between Pruning, Knowledge Distillation (KD), Neural Architecture Search (NAS), and a model reconstruction based on efficient convolutions. To achieve this goal, we train models based on YOLOv3 with the same data augmentation on two datasets, PASCAL VOC and Exclusively Dark Images, and we evaluate them according to Mean Average Precision, number of parameters, storage size, and Multiply-Accumulate Operations (MACs). Results show that a more aggressive Pruning was capable of generating the best trade-off: its mAP surpassed a NAS + KD approach, in addition to producing a model with the smallest number of parameters and with a most effective reduction in MACs.

Keywords: Deep Learning, Object Detection, YOLOv3, Model Compression, Pruning, Knowledge Distillation, Neural Architecture Search, Efficient Convolutions.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS

ANN – Artificial Neural Network

CNN – Convolutional Neural Network

CS – Continuous Sparsification

DSC – Depth-wise Separable Convolution

FCA – Fully-Connected Attention

GAN – Generative Adversarial Network

IMP – Iterative Magnitude Pruning

KD – Knowledge Distillation

KD GAN – Knowledge Distillation based on GAN

LRF – Low-Rank Factorization

LTH – Lothery Tickets Hypothesis

ML – Machine Learning

MLP – Multi Layer Perceptron

NAS – Neural Architecture Search

PE – Projection-Expansion Module

PEP – Projection-Expansion-Projection Module

LIST OF SYMBOLS

CONTENTS

1. INTRODUCTION

Object detection is the task of recognizing objects and retrieving their localization in a given image, being useful for automation, *e.g.,* , in video surveillance, to control the flow of people in and out of a location with restricted access; in self-driving cars, to perceive and understand the objects that are near or far from the car or that are related to traffic signs; in medical image analysis, to locate tumors on diagnostic images; or in robotics, where a robotic arm can pack different objects in their respective boxes.

Deep Learning (DL) has become the state-of-the-art in computer vision tasks. For instance, after Krizhevsky *et al.* [47] winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 with AlexNet, all subsequent models that won the contest were also Convolutional Neural Networks (CNN). Roughly speaking, the deeper these networks are, the better their performance, albeit at the expanse of overparameterization [5].

Theoretically, models without unimportant parameters may provide better generalization, requiring few training samples to learn the function that maps the input to the desired output and providing improved learning speed. The various possible ways of combining convolutional filters with different sizes and many other details allow the creation of models with fewer weights and less computation, like the work of He *et al.* [33], Szegedy *et al.* [73], and Sandler *et al.* [68]. Even so, none of those models have less than 10 million weights or performs less than 2.5 Giga FLoating-point Operations Per Second (FLOPS).

The high computational cost of these models requires the use of graphics cards to accelerate the processing, which is not available in many automation scenarios that require the use of constrained hardware. In video surveillance and other object detection tasks, it is common to find hardware constraints like limited memory, battery restriction, energy consumption, or no GPU acceleration. Consequently, there is a growing area in the literature aiming to reduce these models in different aspects, such as the reduction of the number of operations, storage size, or energy consumption, depending on the need of the problem at hand.

There are many different forms of model compression with different purposes in the literature. In pruning, for example, a large model is created and different techniques are used to remove unnecessary parameters [40, 30, 20, 76]. To optimize the storage size reduction, it is also possible to share parameters, grouping them with non-supervised learning [30]. In knowledge distillation, some studies create a smaller version of a good model and try to use the large model to augment the smaller model performance [6, 86, 81]. In Neural Architecture Search (NAS), the goal is to automate the architecture creation, aiming at a certain number of parameters, inference time, and latency reduction [84, 37, 74, 75], allowing the deployment on mobile devices.

1.1 Problem and General Objective

Although there is quite a lot of work in model compression literature, it remains inconclusive which methods provide good trade-offs regarding model complexity and predictive performance. The literature is not uniform in terms of datasets, models, and data augmentation techniques. Even more, a given study that presents a novel technique for improving its respective approach never compares itself with a technique from another approach [21, 6, 84, 63, 35, 82, 81, 30, 37].

There are studies that only compare techniques from a specific approach, as in [28, 36, 50, 14, 16], which helps the reader to choose a technique but only for a specific compression approach. Recent studies have compared techniques from different approaches, however, with some limitations: presenting evaluation/metrics limited to some approaches, and not evaluating all the approaches with the same metric; presenting the results reported from the original authors without isolating the training scheme; and focusing only on image classification.

Thus, to the best of our knowledge, no work to date attempts at comparing substantially different strategies for model compression in a controlled scenario (say, a NAS strategy *vs* pruning *vs* KD). For that reason, the general objective of this work is to provide a comprehensive comparative study over the performance, compression ratio, and effective computational cost reduction of some of the most relevant works in the literature, focusing on object detection and software-based optimizations.

1.2 Specific Objectives

To achieve the general objective, the following specific objectives were outlined:

- Defining a model to perform model compression.
- Defining a dataset to train/evaluate the models.
- Defining a training scheme for all approaches.
- Evaluating the results with fair metrics for all approaches.

These specific objectives allow a fair assessment of the results, isolating model, dataset, and training scheme, and ensuring that the obtained performance is consequence of the approaches themselves, and not of other external variables.

1.3 Contributions

We list below the main contributions of this work:

- To the best of our knowledge, this is the first work to execut the Lottery Tickets Hypothesis [21] (LTH) and Continuous Sparsification [69] (CS) for object detection.
- This is the first work to study the effective computational cost reduction from LTH and CS.
- For an effective and not just theoretical effort reduction, we propose a technique for reconstructing pruned convolutions that is easy to implement and is hardware/architecture independent, while allowing real measurable gains.
- To the best of our knowledge, this is the first work to evaluate pruning, KD, and NAS approaches within the same experimental protocol.
- This is also the first work to perform this kind of comparison for the object detection task.

1.4 Outline

The remainder of this dissertation is organized as follows:

Chapter 2 explains the basics to understand this work: Deep Learning; training with Stochastic Gradient Descendent; and the object detection task.

Chapter 3 depicts related work and explains the approaches we have evaluated in this dissertation.

Chapter 4 explains this work's methodology, such as the datasets, the training details, and the evaluation metrics.

Chapters 5 and 6 present the results of the experiments and a thorough discussion on the achieved results.

2. BACKGROUND

2.1 Introduction

Machine Learning is a multidisciplinary field of study whose goal is to give computers the ability to learn based on artificial intelligence, statistics, control theory, psychology, neurobiology, and other fields [55]. It makes use of statistics with increased emphasis on estimating complicated functions and decreased emphasis on proving confidence intervals around these functions [26].

Artificial Neural Network (ANN) is an ML approach designed to mimic how the brain performs a particular task or interest function. The motivation for the development of ANNs is the recognition that the human brain processes information differently from the conventional digital computer because the brain is a highly-complex, nonlinear, and parallel computer (information processing system) [32].

Generally speaking, an ANN consists of numerous simple processing units, called neurons, which work in parallel on a large scale and have the ability to learn. One benefit of the ANN learning process is its characteristic of correlating data and generalizing patterns, even in the presence of noise [45].

2.1.1 Neurons

The neurons are the basic units of an ANN, and each of them is an information processing unit. The neuron has three essential elements: a set of synapses that connect neurons, a summation to perform a linear transformation over the input set, and an activation function to restrict the output amplitude of a neuron. We present these three fundamental elements in Figure 2.1.

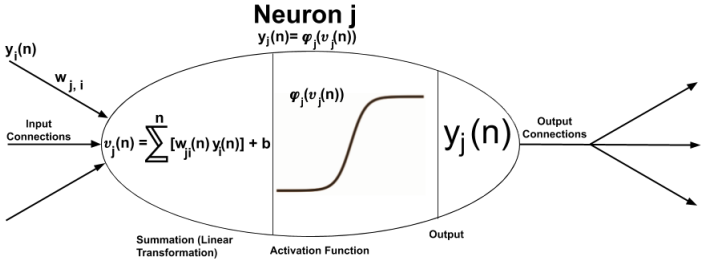


Figure 2.1 – Example of an artificial neuron. Source: Adapted from Russel and Norvig [65].

The neuron has a set of input synapses, which link neuron i to neuron j. These synapses propagate the activation value y_i to j and are associated with weight w_{ij}, which determines the connection signal strength. The neuron computes the receptive field υ_j over a linear transformation on the input set, given by Equation 2.1.

$$\upsilon_j = \sum_{i=0}^{n} [w_{ij} \times y_i] + b_j \tag{2.1}$$

The $\varphi_j(.)$ activation function uses the receptive field υ_j as input and outputs the value (activation) of the neuron j, or y_j, given by Equation 2.2.

$$y_j = \varphi_j(\upsilon_j) = \varphi_j \left(\sum_{i=0}^{n} [w_{ij} \times y_i] + b_j \right) \tag{2.2}$$

Haykin [32] cites examples of activation functions for a neuron, as follows:

Linear activation:

$$\varphi_j(x) = x. \tag{2.3}$$

Logistic, unipolar or sigmoidal activation:

$$\varphi_j(x) = \frac{1}{1 + e^{-x}}. \tag{2.4}$$

Hiperbolic tangent:

$$\varphi_j(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{2.5}$$

Finally, the activation value y_j is propagated to the next neurons through the set of output synapses. Grouping multiple neurons in layers, and connecting all the neurons from a given layer to all the neurons of the next layer (a topology called *fully-connected layer*), we have a Multi-Layer Perceptron (MLP), presented in Figure 2.2.

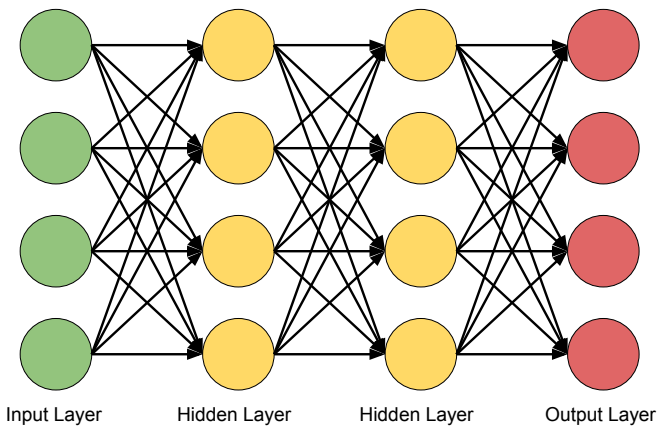


Figure 2.2 – Example of an MLP with an input layer, two hidden layers, and an output layer.

2.1.2 Backpropagation and Stochastic Gradient Descent

To train a neural network, first, we need a loss function $\mathcal{L}$, which measures how wrong is the network output based on the disagreement of the ground-truth $\mathbb{Y}$ with the model output $\hat{\mathbb{Y}}$. A simple example of loss function is the Mean Square Error, $\mathcal{L} = (\mathbb{Y} - \hat{\mathbb{Y}})^2$.

Backpropagation is the procedure for computing the derivatives of the loss function with respect to the model parameters. It gives the gradient direction to update the weights and decrease the disagreement between model output and the ground-truth. Deriving the loss function generates a gradient for a given last-layer neuron, δ_j,

$$\delta_j = \varphi'_j(v_j) \times \mathcal{L}'_j, \tag{2.6}$$

where j indexes the neurons of the final layer. Backpropagation uses the chain rule of derivatives to computes the gradients for the remaining layers, starting from the penultimate layer until the first layer, with

$$\delta_i = \varphi'_i(v_i) \times \sum_{j=1}^{N} w_{ij} \times \delta_j, \tag{2.7}$$

where i indexes the neurons from the current layer l and j indexes the connections between the neuron i and the neurons from layer $l+1$. Finally, we update the weights using an optimization strategy such as Stochastic Gradient Descent (SGD):

$$w_{ij} = w_{ij} - \eta \times \delta_j \times y_i + \alpha \times \Delta w_{ij}, \tag{2.8}$$

where η is the learning rate, which ponders the step size in the weight update, Δ is the w_{ij} update from the last step, and α is the momentum constant, which ponders Δ. The term $\alpha \times \Delta$ is used to accelerate convergence.

2.2 Deep Learning and Convolutional Neural Networks

Convolutional Neural Networks (CNN) is a specific kind of neural network for processing data on a grid-like topology, and that stands out as an example of the neuroscientific principles influencing deep learning [26]. Instead of using the linear transformation over all the input as in Equation 2.1, it performs the linear transformation within the input grid-view. The convolutional weight matrix, also called as kernel, slides over all the input generating a grid-like output, performing the activation function element-wise. We show the operation of a convolutional filter in Figure 2.3.

For CNNs, we need to rewrite the receptive field equation:

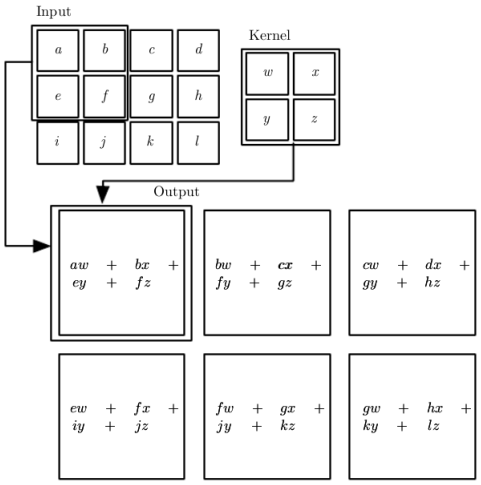


Figure 2.3 – Operation of a convolutional kernel. Source: Goodfellow *et al.* [26].

$$\varphi_{xy} = \sum_{i=1}^{K_w} \left[\sum_{j=1}^{K_h} \left[\sum_{k=1}^{K_d} [\mathrm{X}_{x+1,y+j,k} \times \mathcal{W}_{ijk}] \right] \right] + b, \tag{2.9}$$

where φ_{xy} is the receptive field at the (x, y) coordinate, K_w, K_h, and K_d are the convolutional kernel size regarding width, height, and depth, respectively, X is the input and $\mathcal{W}$ is the convolutional kernel weight matrix. For instance, converting an RGB image to grayscale and applying a convolution with kernel size $3 \times 3 \times 1$, we can build a feature map as in Figure 2.4, which extracts vertical borders from an input image using a convolutional kernel known as Sobel.

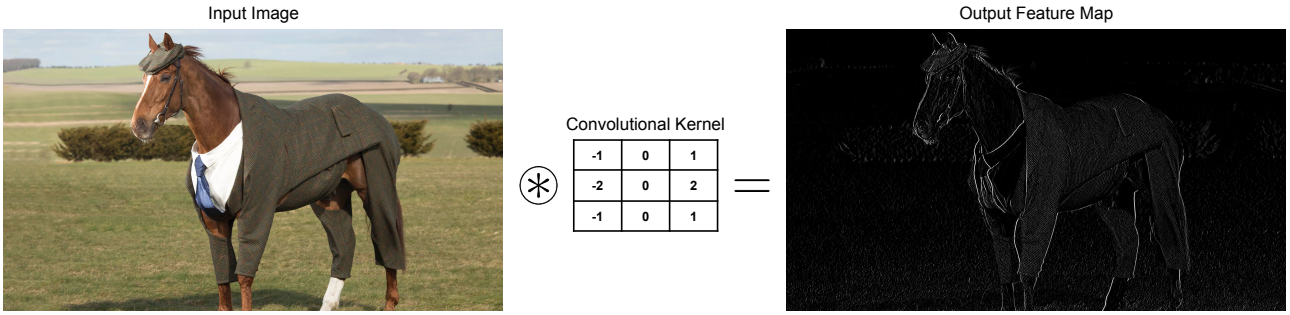


Figure 2.4 – Example of feature extraction by building vertical borders with a Sobel kernel.

In deep learning, the idea is to use Backpropagation to automatically learn each kernel's values instead of using a manually-designed kernel as in Figure 2.4. Thus, CNNs automatically extract features from raw data, instead of the classical methods where the algorithms trains to perform some task based on hand-crafted features. The preprocessing step is also minimal because the idea is to let the model learn what types of variability it must learn to become invariant to [26]. With Figure 2.3 as an example, the backpropagation algorithm learns values w, x, y, and z from the convolutional kernel. These values represent the features learned by the model, and the kernel works as template-matching: after the convolutional operation, each cell from the output will be the input of an activation function. If the activation is high, it means

the neuron found the kernel features within the respective input region. Stacking multiple kernels to build a layer and using multiple layers, we have a CNN, pictured in Figure 2.5.

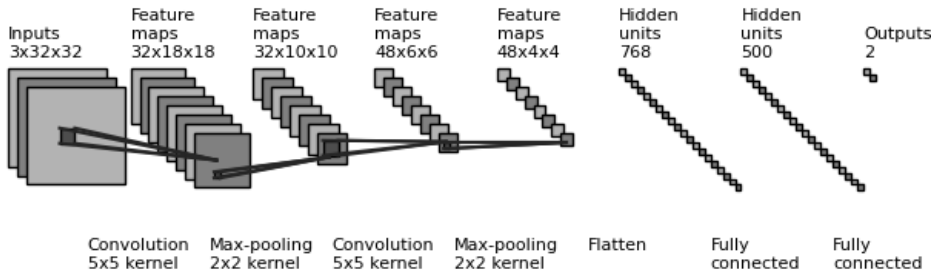


Figure 2.5 – Example of a convolutional neural network architecture.

CNNs learn in a supervised manner how to approximate non-linear functions with the following characteristics [32]:

- Feature extraction: the inputs to a neuron come from the previous layer's local receptive field, forcing it to extract local features. The exact location from a feature becomes less important, as long as it preserves its position relative to other features.
- Feature mapping: each feature map comes from neurons constrained to share the same set of synaptic weights. This reduces the number of free parameters compared to an MLP and allows the creation of feature maps that are shift-invariant.
- Subsampling: after each convolutional layer, we can apply a pooling layer, which subsamples the feature map, reducing the sensitivity of the feature map output to shifts and other forms of distortion.

2.2.1 Pooling Layer

Convolutions can reduce the input resolution when not zero-padded and the kernel is larger than 1×1. A pooling layer is another mechanism for resolution reduction: it compacts a subset of feature values, based *e.g.,* on the average or the maximum value at a given feature subset. This approach is commonly used in many CNN architectures. Since the pooling layer has no parameters, this becomes an advantage over convolution when the objective is resolution reduction. An example of Average Pooling and Max Pooling layers is presented in Figure 2.6.

2.2.2 Object Detection

Object detection is a task that combines two problems: regression, to create a bounding box limiting each object within the scene, and classification, to classify each bounding box

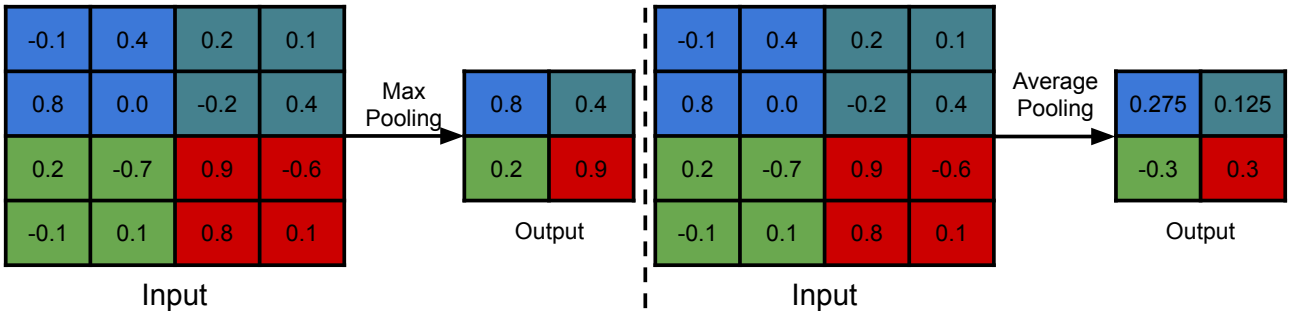


Figure 2.6 – Examples of Average and Max Pooling, with kernel size 2×2 and stride 2, depicted by the distinct colors.

into one of the known classes. Detecting vehicles in a highway video is an example of the object detection task.

Object detection is a task harder than classification. In Figure 2.7, while classification is limited in labeling the image to the main object, which varies from task to task, in object detection there are two objects from different classes, which can be located anywhere in the image, including with overlapping regions, as is the case of this example.

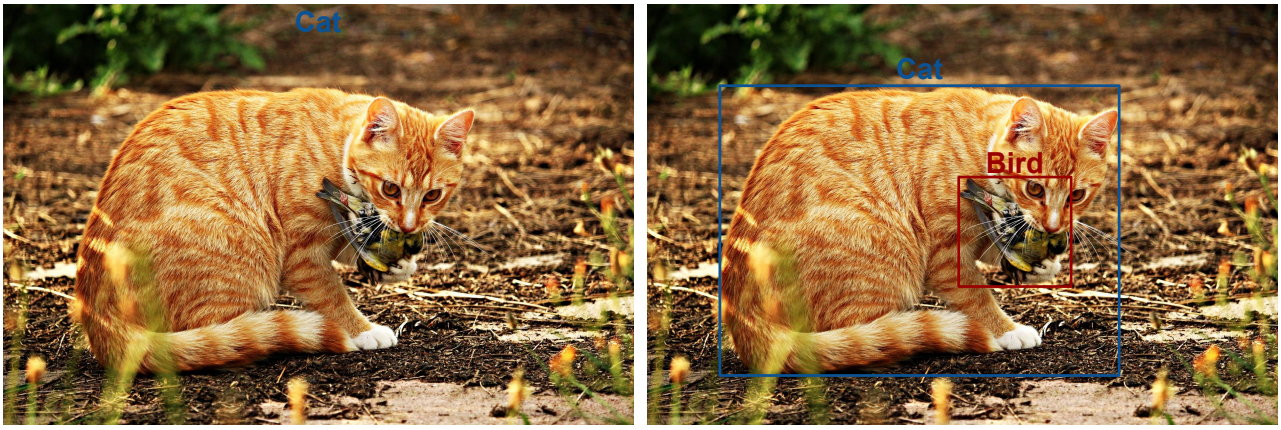


Figure 2.7 – Example of image classification and object detection.

The representation of a bounding box can vary from task to task. For instance, in the PASCAL VOC dataset [17], a bounding box is a set containing the (x, y) coordinates of upper-left and bottom-right corners. In MS COCO [11], there is a (x, y) coordinate of the upper-left corner and the bounding box width and height. This representation can change from model to model, *e.g.,* , in YOLO [60], instead of the upper-left corner, there is the center coordinate of the bounding box.

2.3 YOLO: You Only Look Once

You Only Look Once (YOLO) is a CNN for object detection developed by Redmon *et al.* [58], which classifies objects with high accuracy and reaches up to 45 frames per second. YOLO can reach such results due to its design: it is trained in an end-to-end manner seeing the input only once time, unlike its competitors, which need to see the input many times in many different aspect ratios to perform an ensemble of the outputs, as in R-CNN [24], Fast R-CNN [23], and Faster R-CNN [61].

YOLO works as follows:

- It divides the input image into a $S \times S$ grid, where each cell of the grid is responsible for predicting $\mathcal{B}$ bounding boxes. Figure 2.8 shows an example of the YOLO inference and the $S \times S$ grid.
- For each bounding box, YOLO predicts one confidence score (how confident in the existence of an object inside the grid), computed as $\Pr(Object) \times IOU^{gt}_{pred}$, and more four coordinates: the pair (x, y) with object center position relative to the bounds of the grid cell, and the width w and high h of the object, relative to the whole image.
- For each grid cell, YOLO also predicts $\mathcal{C}$ conditional class probabilities $\Pr(Class_i \mid Object)$. It applies a softmax function for these class probabilities.

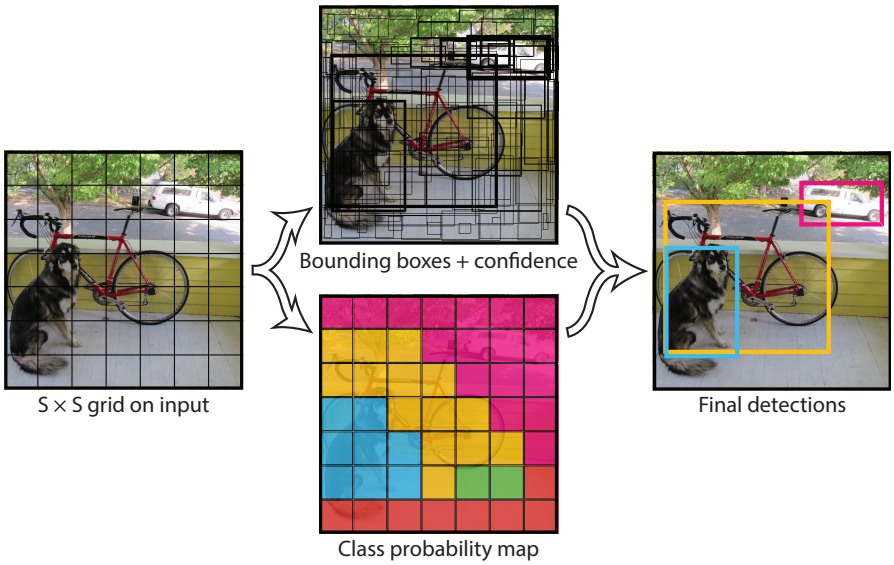


Figure 2.8 – YOLO inference. Source: Redmon *et al.* [58].

The model is a combination of 7×7, 3×3, and 1×1 convolutional layers, in a total of 24 convolutional layers, followed by 2 fully-connected (FC) layers. The model was first created with 20 convolutional layers and trained on the ImageNet dataset with a $224 \times 224 \times 3$ resolution. After this pretraining, the authors added four convolutional layers and two FC

layers with random initialization. Then, the authors finetune the model on the PASCAL VOC dataset [18] with inputs of resolution $448 \times 448 \times 3$. Since the model creates a 7×7 grid, each cell grid containing the 20 classes from PASCAL ($\mathcal{C} = 20$), with two bounding boxes ($\mathcal{B} = 2$) and its respective confidences, YOLO outputs a final volume of size $7 \times 7 \times 30$. The last FC layer has a linear activation function, while the other layers have a Leaky ReLU activation function, given by Equation 2.10.

$$LeakyReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.1 \times x, & \text{otherwise} \end{cases} \tag{2.10}$$

2.3.1 YOLO9000

YOLO9000 is an improvement over the original YOLO performed by Redmon and Farhadi [59]. The authors removed the FC layers, making the model fully convolutional. They also remove one pooling layer to get a higher resolution feature map and introduce batch-normalization layers to remove dropout without overfitting the data. Finally, they introduce a pass-through layer, bringing early features with high resolution to the end of the network, and concatenating it with the low-resolution features, stacking adjacent features into different channels.

To simplify the learning process, YOLO9000 makes use of anchor boxes (predefined bounding boxes) to predict the bounding boxes. Instead of computing the width and the height coordinates of the predicted object relative to the whole image, YOLO9000 computes these coordinates relative to the grid cell size. The YOLO9000 architecture is listed in Table 2.1.

2.3.2 YOLOv3

YOLOv3 is the third version of YOLO. Some differences of YOLOv3 over previous versions are listed below:

- YOLOv3 computes each bounding box's objectness score using logistic regression, unlike previous versions, which use linear activation on the last layer.
- Unlike other models such as Faster R-CNN [61], YOLOv3 assigns only one bounding box to each ground-truth object, meaning that if the model does not assign the bounding box to a ground-truth object, there is no loss of bounding box regression or class predictions, but only for the objectness.
- The model performs predictions with different scales. For example, configuring the model to predict bounding boxes at three scales, the output from YOLOv3 is three volumes

Table 2.1 – Architecture from YOLO9000. Source: adapted from Redmon and Farhadi [59].

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

with dimensions $S \times S \times [3 * (4 + 1 + N)]$, where 3 is the total number of bounding boxes ($\mathcal{B} = 3$), 4 is from the four bounding boxes offsets, 1 is from the objectness prediction, and N is the number of classes. To compute this prediction with different scales, YOLOv3 has ramifications in the model, where some branches use a pass-through layer, as in YOLO9000, bringing early features to the end of the network. As the early and current features have different resolutions, the model performs a 2D interpolation to upsample the low-resolution ones.

The architecture of YOLOv3 is slightly different. It has 53 convolutional layers, with many residual connections, pass-through layers, and three main branches, which perform the detection at three different scales, each with a different $S \times S$: 13×13, 26×26, and 52×52, respectively. We can see the architecture of YOLOv3 in Figure 2.9.

With these improvements, YOLOv3 achieves an mAP-50 of 57.9 in the COCO dataset running on 51 milliseconds per image, beating some similar competitors, belonging to the same category of one-stage inference such as SSD (45.1 mAP, 61 ms) and RetinaNet-101 (57.5 mAP, 198 ms). YOLOv3 narrowly loses to Faster R-CNN on mAP (59.1) by just 1.6 mAP points. However, YOLOv3 is more than three times faster than Faster R-CNN, which runs in 172 milliseconds per image [60].

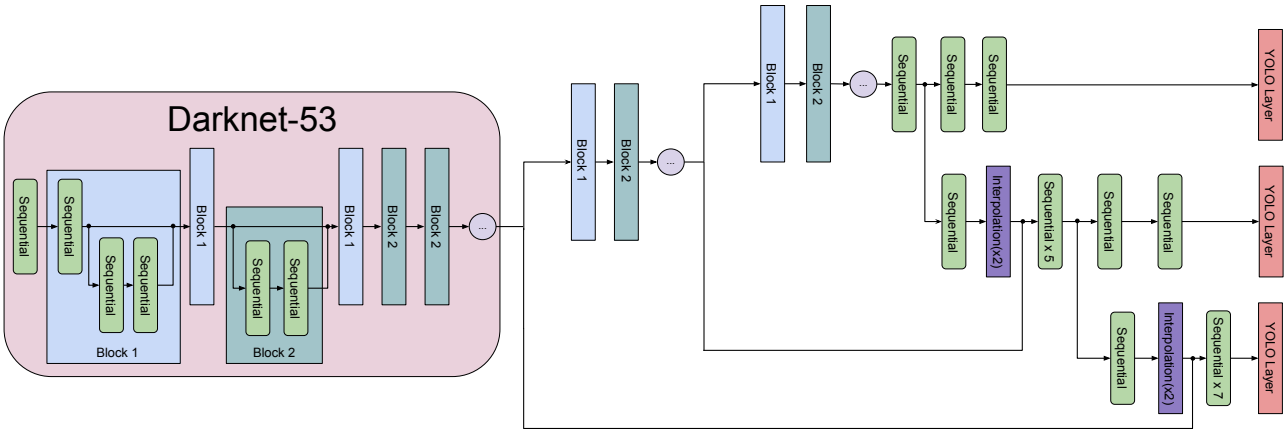


Figure 2.9 – YOLOv3 macro architecture.

2.3.3 Original Loss Function

YOLO loss generally has three main elements: the bounding box regression loss; the object confidence regression loss; and the classification loss. The first element from the loss function is equal in all YOLO versions.

Bounding Box Regression Loss

In the bounding box regression loss, for each bounding box $\mathcal{B}$ predicted at each grid cell S, the loss is computed with the Mean Squared Error, with Equation 2.11,

$$\begin{aligned}\mathcal{L}_{coord} = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2\right] + \\ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2\right]\end{aligned} \tag{2.11}$$

where x_i and y_i are the ground-truth bounding box center coordinates, $\hat{x}_i$ and $\hat{y}_i$ are the model inference of the center coordinates, w_i and h_i are the width and height ground-truth of the bounding box and $\hat{w}_i$ and $\hat{h}_i$ are the model inference of the width and height. The $\mathbb{1}_{ij}^{obj}$ means the loss computation if the i-th grid cell S is responsible to predict the j-th bounding box (*i.e.*, has the highest IOU of any predictor in that grid cell). The λ_{coord} term is a weight to balance the importance of the term in the final loss function and is equal to 5 in the work of Redmon and Farhadi [58].

Confidence Loss

In the confidence loss, for each bounding box $\mathcal{B}$ predicted at each grid cell S, the loss is computed using Logistic Regression as in Equation 2.12,

$$\mathcal{L}_{conf} = \sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{b=0}^{B} [c_i \log(p(c_{ib})) + \lambda_{noobj}(1 - c_i) \log(1 - p(c_{ib}))] \tag{2.12}$$

where c_i is the ground-truth object confidence, that is, if there exists an object inside the grid cell i. $p(c_{ib})$ is the model object confidence inference. The term $\mathbb{1}_i^{obj}$ indicates whether there is an object in the grid-cell. Only one bounding box is assigned to the ground-truth. If a bounding has an IoU with the ground-truth higher than the threshold 0.5, $c_i = 1$. If more than one bounding has the IoU larger the threshold, only the higher one has $c_i = 1$, while the others have $c_i = 0$, and consequently, there is no bounding box regression loss, neither classification loss, only the confidence loss. λ_{noobj} is a weight to balance the error when $c_i = 0$, used as 0.5 in the work of Redmon and Farhadi [58].

Classification Loss

For object classification, YOLOv3 uses Logistic Regression for each prediction, as in the Equation 2.13,

$$\sum_{i=0}^{S^2} \mathbb{1}_{ij}^{obj} \sum_{b=0}^{B} [c_i \log(p(c_{ib})) + (1 - c_i) \log(1 - p(c_{ib}))] \tag{2.13}$$

where c_{ij} is the ground-truth probability class at cell ij and $p(c_{ij})$ is the probability predicted by the model of the ij-th grid cell with an object belonging to the c-th class.

Finally, we can write the total loss of YOLO in the Equation 2.14:

$$\mathcal{L}_{total} = \mathcal{L}_{coord} + \mathcal{L}_{conf} + \mathcal{L}_{class}. \tag{2.14}$$

2.4 Intersection Over Union

Intersection Over Union (IoU) is the resulting ratio between the intersection area of two bounding boxes and the union area of these two bounding boxes. We can see the computation of IoU between a prediction and the ground-truth in Figure 2.10 and its formula in Equation 2.15.

$$IoU = \frac{|A \cup B|}{A \cap B} \tag{2.15}$$

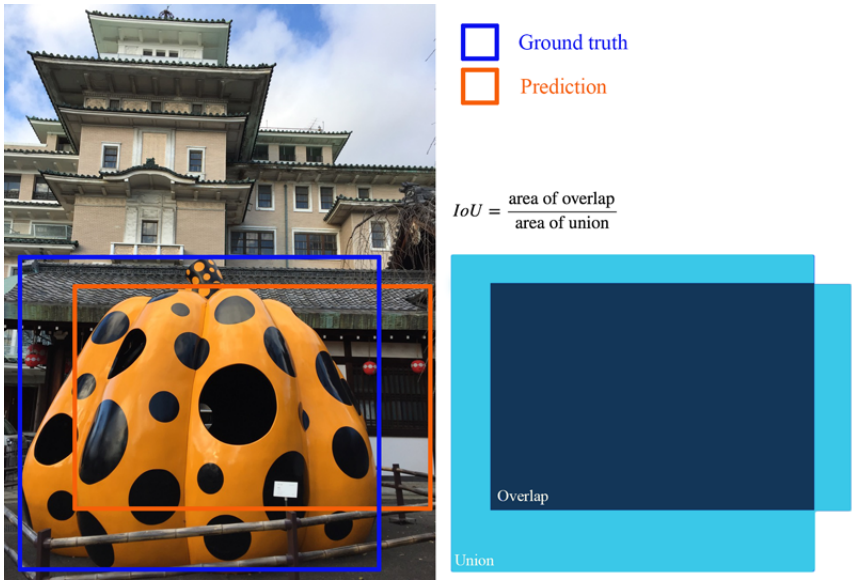


Figure 2.10 – Intersection Over Union example. Source: [39]

In this example, IoU is 0 when the model inference has no intersection with the ground-truth, and is 1 when it is precisely equal to the ground-truth. Hence, we want an IoU as close to 1 as possible. The IoU has a desirable property of scale invariance because it is a ratio between areas, no matter their size.

2.5 Generalized IoU

Although the IoU can be used as a metric to evaluate models, there is a gap between the IoU and the traditional loss functions of bounding box regression, such as the ℓ_n distances. Different predictions can have the same losses but different IoU, as shown in Figure 2.11. Also, IoU has a second problem. With no intersection between inference and ground-truth, the IoU is zero, independently of the distance between the boxes, a plateau that makes it infeasible to optimize in the case of non-overlapping bounding boxes. Obviously, between two inferences, both with no intersection, the best is the nearest one.

To address these problems, Rezatofighi *et al.* [62] created the Generalized Intersection Over Union (GIoU), a metric both for evaluation and loss function, which also avoids the problem of non-overlapping bounding boxes. The GIoU also uses the smallest convex hull that encloses both bounding boxes. We can see the GIoU in Figure 2.12 and its respective formula in Equation 2.16:

$$GIoU = IoU - \frac{C \setminus |A \cap B|}{C}, \tag{2.16}$$

where C is the convex hull (dashed circle in Figure 2.12). Note that a corner of one bounding box needs to be the center of C, while a corner of the other bounding needs to overlap C. In other words, the GIoU is a ratio between the C area, excluding A and B, and dividing by

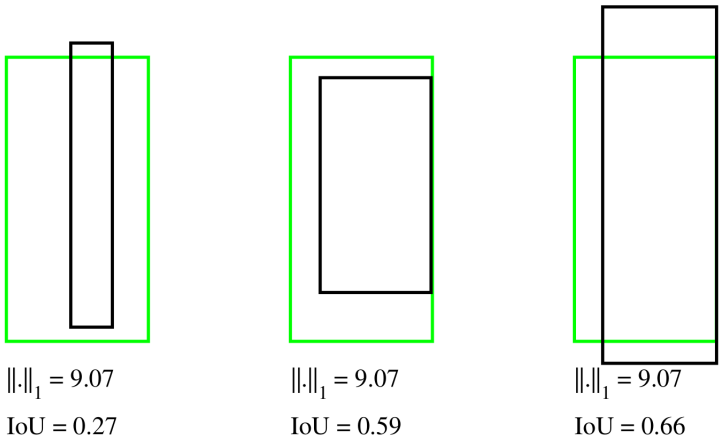


Figure 2.11 – Bouding boxes with same ℓ_1 loss and different IoU. Source: Rezatofighi *et al.* [62].

the total C area. Thus, for the example of the two bounding boxes with non-overlapping, the smaller is the convex hull, the larger is the GIoU, ensuring that the models do not fall on a plateau and thus can learn to optimize the model.

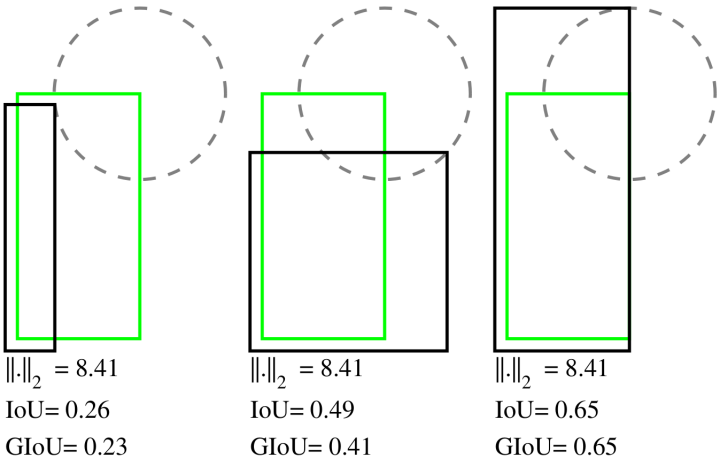


Figure 2.12 – Generalized Intersection Over Union graphical example. Source: [62].

Finally, we can transform the GIoU to a loss function with Equation 2.17, since we want to minimize the loss and increase the GIoU. In this manner, the larger the GIoU, the smaller the loss:

$$\mathcal{L}_{GIoU} = 1 - GIoU. \tag{2.17}$$

2.6 Non-Max Supression

It is common in object detection models to generate an excessive amount of bounding boxes. For instance, in YOLO, YOLO9000, or YOLOv3, which work by dividing the image into a grid, when the object is large enough to fall into more than one grid-cell, it is common

Figure 2.13 – Example of Non-Max Suppression.

to have multiple detections on the same object, coming from different grid-cells. A common approach in the literature to eliminate the excessive amount of bounding boxes is the Non-Max Suppression (NMS) procedure. NMS computes the IoU between each pair of bounding boxes belonging to the same class. If the IoU is larger than a threshold, the approach removes the bounding boxes with less confidence. We can see an example of NMS in Figure 2.13.

The expectation is that NMS filters the predictions and keeps the more accurate bounding boxes. In [58], Redom *et al.* reveal that NMS improves between $2-3\%$ in Mean Average Precision (mAP), a metric for object detection which we will explain in Section 4.2.1.

3. RELATED WORK

In this chapter, we first give a general description of the existing approaches for model compression in deep learning. Then, we comment on surveys and comparisons that exist on the matter in the specialized literature. Finally, we review the approaches that will be experimentally evaluated in this work.

3.1 Existing Approaches

There are many kinds of model compression methods in the literature, each of which with its goals and peculiarities. Based on the work of Cheng *et al.* [8], Wang *et al.* [80], Choudhary *et al.* [10], Salvi and Barros [67], and Agarwal *et al.* [1], we group the existing model compression work into the following categories:

- Parameter Pruning and Sharing: removal of redundant and uncritical parameters, either in a structured manner (*e.g.,* removing neurons or layers), or in an unstructured manner (removing connections). It can be performed both in convolutional and FC layers. Support training from scratch and pre-trained models, depending on the technique.

- Quantization: reduction of the model storage size by reducing the precision of the machine representation for the model parameters.

- Low-Rank Factorization (LRF): decomposition of matrices to estimate the informative parameters of the neural network. It can be performed both at convolutional and FC layers, and support both training from scratch and pre-trained models.

- Transferred/Compact Convolutional Filters: designing special structures of convolutional filters to reduce the parameter space. This method can be applied only on convolutional layers, and the algorithms are application-dependent. Supports only training from scratch.

- Neural Architecture Search (NAS): automation of the architecture engineering for deep neural networks. A search strategy generates and trains a network, whose performance guides the search strategy to create another model. The search can be performed by a variety of methods, such as random search, Bayesian Optimization, Evolutionary Algorithms, gradient-based search, or even Reinforcement Learning [16].

- Knowledge Distillation (KD): distillation of a model by training a more compact neural network to reproduce the output of a larger model. Supports convolutional and FC layers, and the network structure only supports training from scratch.

Pruning is a post-processing step after typical training that removes non-significant/ redundant connections/neurons based on gradients or other heuristics. Pruning-based approaches rely on the concept that it is necessary to train large networks to find smaller subnetworks with a performance that tends to be similar to the original model. The Optimal Brain Damage (OBD) from LeCun *et al.* [48] and Optimal Brain Surgeon (OBS) from Hassibi *et al.* [31] are both methods to reduce the number of connections based on the Hessian Matrix of the Loss Function $\mathcal{L}$. With Weight Elimination from Weigend *et al.* [83] and Approximate Smoother from Moody and Rögnvaldsson [56], all of these methods described here are based on the idea of complementing the Loss Function $\mathcal{L}$ with terms that punish some of the weights $\mathcal{W}$ in order to reduce the number of connections as well as overfitting. In these cited examples, backpropagation learns which weights $\mathcal{W}$ should be eliminated.

In another strategy, Han *et al.* [30] reuse the weights (weight sharing) to reduce the memory space. To find which weights to keep and which to remove, they use k-Means to cluster the parameters. To reduce the memory space, the authors employ Huffman coding in the weights. Frankle *et al.* [21] iteratively train and remove the weights based on the magnitude of the values and reinitialize them to an intermediate state of the training process. The Savarese *et al.* [69] approach continuously learns which weights to remove based on backpropagation.

Despite the fact that pruning approaches present, in general, the smallest models with the best performance, they generally generate sparse matrices, which means one needs to remodel its structure to benefit from real model reduction, since operations on sparse matrices within the available frameworks/hardware are not typically optimized.

In **quantization**, the goal is to reduce weight storage. For instance, Gupta *et al.* [29] make use of a 16-bit fixed-point representation of the weights. Vanhoucke *et al.* [78] and Han *et al.* [30] use an 8-bit floating-point representation. Some studies use even a 1-bit weight representation, which is the case of BinaryConnect [12], BinaryNet [13], and XNORNet [57]. Even though these methods can considerably reduce the storage space of ANNs, the accuracy is also considerably reduced when applied on large CNNs [8]. Thus, it is common to find studies using quantization combined with other approaches, as in Han *et al.* [30], Gong *et al.* [25] and Tung and Mori [77].

In the **Low-Rank Factorization** approach (LFR), the idea is to remove the redundant parameters, decrease the amount of computation, and maintain the model's performance. One example is the separable filters created by Rigamonti *et al.* [63]. The authors replace a 3×3 convolutional layer by a 1×3 followed by a 3×1 convolutional layer. Both approaches have 9 and 6 parameters, respectively, and the same receptive field φ. Denton *et al.* [15] propose using bi-clustering to cluster the rows and columns of the convolutional kernels, while the filters with high redundancy of features will be a group. This operation breaks the weight matrix $\mathcal{W}$ in sub-tensors, which is easier to fit with a low-rank approximation. These low-rank factorization methods involve the factorization operation, which is computationally expensive and requires extensive model retraining, making the whole process slow [8].

LRF tries to decompose a large learned weight matrix $\mathbb{A}$ into small ones, *e.g.,* $\mathbb{B}$ and $\mathbb{C}$, and tries to learn the values in $\mathbb{B}$ and $\mathbb{C}$ which, combining the inference of both, results in an output that mimics the feature map originally generated by $\mathbb{A}$. In the **Transferred Convolutional Filters** approach, instead of decomposing this learned matrix into small ones, it creates and trains the combination of matrices, *e.g.,* $\mathbb{B}$ and $\mathbb{C}$, from scratch. For instance, Li *et al.* [49] create a multi-biased activation function, changing a convolutional layer with N channels by a convolution of N/b channels. Then, they concatenate the obtained feature maps summed with b different biases, reducing the redundancy of parameters. Iandola *et al.* [41] propose the squeeze layer. It performs a sequence of 1×1 convolutional layers, and its goal is to reduce the number of input channels of the following 3×3 convolutional layers. These approaches surely reduce the storage space and the inference cost and can be easily introduced in any architecture, but also decreasing model performance.

Shang *et al.* [71] find that layers at the beginning of the network create filters forming pairs in "positive" and "negative" phases. This means that only half of these parameters are needed. Using a convolution with half size, they reconstruct the other half by replicating the filters, reversing the signal. They also create the Concatenated ReLU (CReLU), which alleviates the redundancy among the convolutional filters. CReLU copies the linear transformations after the convolution, invert its signs values, and concatenates both activation parts, applying then a ReLU altogether.

The **Neural Architecture Search** approach aims at developing an automated way to build an efficient architecture. Instead of human-crafted architectures, with handcraft factorized convolutions based on trial and error, this approach replaces the human factor by reinforcement learning or evolutionary programming. Looking in the literature, there seems to be a convergence in using an RNN, called a controller, to sample a child architecture. After this, the approach trains the child architecture in the goal task, to compute its performance, and uses the performance to train the controller with reinforcement learning. For example, Zoph and Le [90] use the mentioned description to generate an efficient convolutional network. To speed up the training, they train multiple children networks in parallel and perform asynchronous updates on the controller. Training 800 children networks concurrently in 800 GPU, in a total of $12,800$ children networks by 28 days, the authors create architectures with comparable error rates on CIFAR-10 — 6.01% of error rate with $2.5M$ of parameters and 20 layers. However, this performance is comparable to works with no NAS, such as the DenseNet with 40 layers, that achieves a smaller error rate of 5.24% and has fewer parameters, with $1M$.

Tan *et al.* [74] create the MnasNet as a multi-objective search to find models with high accuracy and low inference latency. They formulate the problem as multiple Pareto optimal problems. In other words, there exist solutions with high accuracy without increasing latency or with the lowest latency without decreasing accuracy. In their work, the controller searches by basic blocks, the operations, and connections to link these basic blocks. The approach also groups the basic blocks into a predefined skeleton and learns to repeat these skeletons as many

times as needed. With this approach, the authors outperform the MobileNetV2 tradeoff between latency and ImageNet classification accuracy. It is important to highlight that this approach is expensive since the authors use 64 Tensor Processing Units (TPU), a device explicitly designed for deep learning. The training lasted 4.5 days over $50K$ training images.

Similarly, Zoph *et al.* [91] use a controller to sample a generic convolutional cell. Their contribution is the design of a new search space. Their controller searches for the best cell based on CIFAR-10. Grouping these basic cells as Inception blocks and stacking them, results in the so-called NASNet, which outperforms the tradeoff between multiply and add operations versus accuracy on ImageNet, when comparing with models as MobileNetV1, Xception, or ResNeXt-101. However, it is important to highlight that, even reducing the searching space, the authors use 500 GPUs that need to be trained for four days.

Based on the assumption that designing tiny networks from scratch is not sufficient to achieve performance comparable to larger/wider networks, the **Knowledge Distillation** (KD) area aims at reducing the gap between tiny and large models. KD trains the reduced model to mimic the behavior of a larger model. The reduced model is called a student, and it shifts knowledge from the larger model, the teacher, learning its class distribution. Ba and Rich [3] create the Shallow Neural Net (SNN), a single-layer FC network trained with a Norm ℓ_2 Distance Loss to mimic the behavior of an ensemble of CNN trained on CIFAR-10 [46]. Hinton *et al.* [35] propose a hint layer to guide the intermediate student feature maps. Romero *et al.* [64] propose to distill the knowledge of an ensemble that comprises many specialist models. As in [35], the student model mimics the teacher model's outputs and the feature maps from the intermediate layers. Guobin *et al.* [6] creates a student of an object detection model that has no bounding box regression loss when its outputs outperform the teacher. These approaches can build student models with a reduced performance gap between teacher and student. Although KD approaches can reduce the discrepancy between student and teacher models, sometimes both models must have the same architecture family since the two models represent the features maps in different domains and sometimes learn very different features [8].

3.2 Existing Comparisons and Surveys

We can find existing surveys in the literature related to object detection or model compression. For object detection as a whole, we can find, *e.g.,* the work of Zou *et al.* [92], which gives an historical description of the object detection literature, from classical ML models using handcraft features to DL models using features learned by gradient descent. They also provide some model compression approaches, but this is not the focus of the work, and the authors give only a description of the areas. Another example is the work of Agarwal *et al.* [1], which explains the DL loss functions, the data augmentation most used for object detection, the difference between one-stage and two-stage object detectors, etc. Related to the

area constraints, they provide a discussion on the tradeo-ff between performance and speed, briefly discussing some models such as YOLOv2 and other one-shot detectors.

With a focus on model compression, the work of Cheng *et al.* [8] is a survey that presents explanations and discussions on pruning, quantization, LRF, transfered convolutional filters, and KD. Similarly, the work of Ge [22] details pruning, quantization, network approximation, KD, and network densifying. Both studies do not provide an experimental analysis involving the approaches. An interesting point is that Ge [22] concludes that there is little work on object detection, tracking, and other visual tasks. Neither of them consider NAS as a tool for model compression. In [2], A'r âbi and Schwarz report general constraints and tradeoffs in embedded ML and DL, such as reducing latency, increasing reliability, power versus cost, privacy, and security. They divide the existing approaches into two families: software optimization, such as efficient architectures, dimensionality reduction of input data, or cache-based approaches to avoid memory access; and hardware optimization, such as reducing the floating-point precision, additional resources as cloud computation, and FPGAs. They explain and recommend whether to use CPU based on RISC or CISC architectures, but they also lack any experimental comparison among the methods.

In [10], the authors survey over LRF, KD, quantization, pruning, and efficient architectures. They provide practical comparisons between the approaches, however, in restricted scenarios. For instance, there is an analysis showing the parameter reduction, the top-1 and top-5 classification errors of an AlexNet in a comparison of pruning, pruning with quantization, and LRF, as well as with a VGG-16, comparing pruning, pruning with quantization, quantization, and LRF. In both cases, they do not analyze KD nor efficient architectures. Moreover, their comparison only reported the original results of the referenced authors. Thus, there is no way of ensuring that the same training scheme was actually performed, damaging the confidence of the reader on the validity of the results. Then, they compared the number of parameters, latency, and top-1 classification error on ImageNet of models with efficient architectures, such as SqueezeNet and MobileNet, but without mentioning approaches such as pruning or LRF. In the end of their work, they presented a comparison of their own results with an AlexNet on CIFAR10 and MNIST, but only evaluating pruning and quantization. There is an additional experiment, but only evaluating AlexNet with two quantization approaches on CIFAR10, one quantization approach on ImageNet, and one quantization with ResNet18 on ImageNet. Hence, the reader cannot tell which method (or set of methods) provides the best trade-off between compression and accuracy.

In [80], Wang *et al.* surveyed over model compression approaches and analyzed quantization, pruning, parameter sharing, LRF, and activation approximation. One of the experiments show the throughput (inferences produced per second or classification rate) of an AlexNet over ImageNet versus weight precision versus activation precision in two different hardware with fixed-point representation for the weights and activation function. This is an interesting comparison to evaluate the tradeo-ff between precision and throughput. However, it

is useful only for quantization approaches. Another comparison shows the top-one error rates *vs* compression rate for implementations of AlexNet over ImageNet. Again, these are the original authors' results, which do not implement the same data augmentation strategy, for instance, impacting the validity of the comparison. This comparison analyzes quantization, pruning, weight sharing, and LRF, but not NAS or KD. Finally, there is a trade-off comparison between compression and accuracy, but only for quantization approaches in the classification task.

3.3 Lottery Tickets Hypothesis

3.3.1 Original Algorithm

The Lottery Tickets Hypothesis (LTH), also called Iterative Magnitude Pruning (IMP), is a state-of-the-art algorithm for model compression, aiming to prune as many connections as possible while maintaining the model's performance. The rationale is that in dense, randomly-initialized networks, we can find subnetworks (winning tickets) able to reach an accuracy comparable to the original model.

In Equation 3.1, we can denote a neural layer inference as:

$$\hat{\mathbb{Y}} \leftarrow \phi(\mathcal{W}, \mathbb{X}), \tag{3.1}$$

where $\mathbb{X}$ is the input vector, $\mathcal{W}$ is the weight matrix, $\phi()$ is the activation function and $\hat{\mathbb{Y}}$ is the inference output. In lottery tickets [20], we can rewrite the forward pass with Equation 3.2:

$$\hat{\mathbb{Y}} \leftarrow \phi(\mathcal{W} \odot \mathcal{M}, \mathbb{X}), \tag{3.2}$$

where $\mathcal{M}$ is a binary mask with same shape as $\mathcal{W}$, holding or pruning the weights, and $\odot$ is the Hadamard Product. Given a pruning rate ρ, each iteration in LTH prunes the remaining weights with lower magnitude, meaning $\mathcal{M}_{i,j} = 0$, and the iterations are performed until the number of remaining weights reaches the desired compression rate.

Algorithm 3.1 shows the pseudocode for LTH, where c is the number of desired remaining weights at the final of compression, $\mathcal{W}_0$ is a backup of the initialization, $count\_parameters(.)$ is a function to count the number of weights in the weight matrix $\mathcal{W}$ that was not pruned (in the same indexes, $\mathcal{M} = 1$), $remaining\_weights$ is the number of weights not pruned in $\mathcal{W}$, and $\min_{\mathcal{W}} \mathcal{L}(\mathbb{Y}, \hat{\mathbb{Y}})$ indicates the parameter update with Stochastic Gradient Descent. Function $find\_smallest\_values(.)$ will find the ρ smallest values from the input matrix, receiving $|\mathcal{W}|$ as input to reach the smallest weights in absolute values. This function works both in FC and convolutional layers, and can find the values locally, searching layer by layer, or globally, searching all layers. We can see the difference between the local and global search in Figure 3.1.

Algorithm 3.1 Lottery Tickets Hypothesis. Source: adapted from [20].

Require: weight matrix $\mathcal{W}$, mask $\mathcal{M}$, pruning rate ρ, number of steps n_steps, desired compression c, input $\mathbb{X}$, labels $\mathbb{Y}$

```
random_initialization(W)
initialize_with_ones(M)
remaining_weights ← count_parameters(M)
W_0 ← W
while remaining_weights > c do
    for epoch ∈ n_epochs do
        W ← W ⊙ M
        Ŷ ← φ(W, X)
        min_W L(Y, Ŷ)
    end for
    indexes ← find_smallest_values(|W|, ρ)
    M[indexes] ← 0
    W ← W_0
    remaining_weights ← count_parameters(M)
end while
W ← W ⊙ M
train()
```

In Figure 3.1, the subscribed numbers, separated by commas, indicate the index of the connection's input neuron and the index of the connection's destination neuron, respectively. While in IMP with local pruning two connections are pruned per weight matrix, with IMP global one connection is removed from the first weight matrix and three from the second.

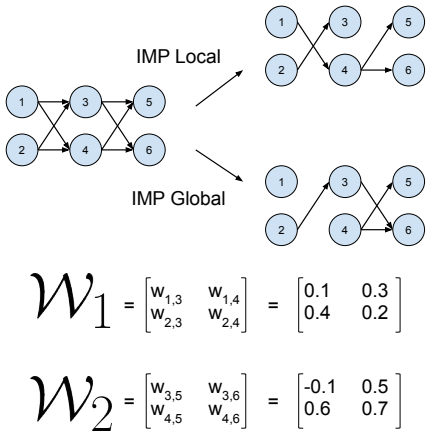


Figure 3.1 – Difference of IMP searching parameters globaly and localy, using $\rho = 0.5$.

Note that backpropagation does not work over mask $\mathcal{M}$, so the mask is updated only by command $\mathcal{M}[indexes] \leftarrow 0$. Backpropagation will update all weights in $\mathcal{W}$, even the pruned ones. Since the algorithm applies the Hadamard between $\mathcal{W}$ and $\mathcal{M}$ before each forward pass, it overrides the update on the pruned weights and re-prune them.

With this algorithm, Frankle and Carbin [20] prune the LeNet 300-100, an FC with 300 neurons in the first hidden layer and 100 neurons in the second hidden layer, trained over the MNIST dataset. They apply a pruning rate $\rho = 20\%$ iteratively until only 7.0% of the parameters remain, and they are capable of outperforming the original model. The results obtained by the authors also showed that the gap between training accuracy and test accuracy is smaller for the winning tickets. For CNNs, the authors prune scaled-down variants of the VGG. They train on the CIFAR10 dataset and reach a model with 26.4% the size of the original model and test accuracy 3.3% better [20].

3.3.2 Late Reseting

Frankle *et al.* [21] updated their method achieving better results for CNNs [21]. The older LTH version reinitializes the weight matrix $\mathcal{W}$ to the initialization values. In this new LTH version, it reinitializes the weights to a k-th training step, generally from 0.1% to 7% of the total epochs of the iteration. We can see this new method in Algorithm 3.2, where $\mathcal{W}_k$ is the backup of the weights in the late resetting state. The remaining pseudocode remains similar to Algorithm 3.1.

Algorithm 3.2 Iterative Magnitude Pruning (IMP) with rewinding to iteration k. Source: adapted from [21].

Require: weight matrix $\mathcal{W}$, mask $\mathcal{M}$, pruning rate ρ, backup epoch k, total of epochs n_epochs, desired compression c, input $\mathbb{X}$, labels $\mathbb{Y}$

$random\_initialization(\mathcal{W})$
$initialize\_with\_ones(\mathcal{M})$
$remaining\_weights \leftarrow count\_parameters(\mathcal{M})$
while $remaining\_weights > c$ **do**
  for $epoch \in n\_epochs$ **do**
    if $epoch = k$ **then**
      $\mathcal{W}_k \leftarrow \mathcal{W}$
    end if
    $\mathcal{W} \leftarrow \mathcal{W} \odot \mathcal{M}$
    $\hat{\mathbb{Y}} \leftarrow \varphi(\mathcal{W}, \mathbb{X})$
    $\min_{\mathcal{W}} \mathcal{L}(\mathbb{Y}, \hat{\mathbb{Y}})$
  end for
  $indexes \leftarrow find\_smallest\_values(|\mathcal{W}|, \rho)$
  $\mathcal{M}[indexes] \leftarrow 0$
  $\mathcal{W} \leftarrow \mathcal{W}_k$
  $remaining\_weights \leftarrow count\_parameters(\mathcal{M})$
end while
$\mathcal{W} \leftarrow \mathcal{W} \odot \mathcal{M}$
$train()$

With this improvement, Frankle *et al.* [21] are capable of generating a Resnet-50 trained on Imagenet dataset with 80% of the weights pruned and with the same accuracy of the original model. Late resetting improves performance by introducing two kinds of stability [21]:

- Stability to pruning: it is a kind of stability measured by the distance between the weights of a subnetwork trained in isolation (for example, the final pruned model) and the weights of the same subnetwork when trained within the larger network (for example, the equivalent subnetwork inside a non-pruned model). This kind of stability captures a subnetwork's ability to train in isolation and still reach the same destination as the larger network.
- Stability to data order: it is the stability measured by the distance between the weights of two copies of a subnetwork trained with different data order. This captures a subnetwork's intrinsic ability to consistently reach the same destination despite Stochastic Gradient Descendent's gradient noise.

The authors perform experiments comparing the results increasing the value of k and discovering that rewinding later causes gradual improvements in stability and finding the lottery winners.

Increasing k also achieves better stability and accuracy than a network with a new initialization (line 15 of Algorithm 3.2).

3.4 Continuous Sparsification

Continuous Sparsification (CS) is an alternative approach to the LTH method developed by Savarese *et al.* [69]. In this method, SGD is used to continuously learn a deterministic mask as an ℓ_1 regularization problem. Since gradient descendent does not create binary values, in the original approach the mask $\mathcal{M}$ is re-parameterized over the soft-mask $\mathcal{S} : \mathbb{R}^D$, as in Equation 3.3:

$$\hat{\mathcal{M}} \leftarrow \sigma(\beta\mathcal{S}), \tag{3.3}$$

where σ is the sigmoid function and β is a temperature value. The higher the β, the steeper is the sigmoid function and closer to binary values when $\beta \to \infty$. However, the gradient vanishes and the approach cannot learn which parameters on $\mathcal{W}$ to remove. To mitigate this problem, β is annealed slowly during training and is reinitialized after each iteration. Additionally, one resets the soft-mask of the remaining weights with $\mathcal{S} \leftarrow \min(\beta\mathcal{S}, \mathcal{S}_0)$, which does not interfere in the removed ones. This re-initialization gives the approach a moment to "rethink" which parameters to keep or not.

After computing $\mathcal{M}$ with Equation 3.3, the mask is applied before each forward pass as in Equation 3.2, and the **backward** pass is computed to minimize the following equation:

$$\min_{\substack{\mathcal{W}\in\mathbb{R}^D \\ \mathcal{S}\in\mathbb{R}^D}} \mathcal{L}(\mathbb{Y}, \phi(\mathcal{W} \odot \sigma(\beta\mathcal{S}), \mathbb{X})) + \lambda \cdot \|\hat{\mathcal{M}}\|_1, \tag{3.4}$$

where $\mathcal{L}$ is the traditional loss function used to solve the problem, $\mathbb{X}$ is the input set and the weight decay λ to balance the ℓ_1 regularization. $\phi(\mathcal{W} \odot \sigma(\beta\mathcal{S}), \mathbb{X})$ denotes a model inference step applying the learned mask on the weights. Algorithm 3.3 shows the proposed approach.

Here, β is annealed after the end of each epoch, and the function $binarize(.)$ is applied after the process learned the soft-mask, setting to 1 the positive values and 0 otherwise. In the end, the approach calls $train(.)$ to fine-tune the model weights, freezing the soft-mask $\mathcal{S}$.

3.5 YOLO Nano

Wong *et al.* [84] propose YOLO Nano, a reduced version of YOLOv3 as a compressed model for object detection in constrained scenarios such as embedded devices. They create YOLO Nano with machine-driven exploration over human design prototyping and YOLO-family design. YOLO Nano has a macro-architecture similar to that shown in Figure 2.9: it

Algorithm 3.3 Continuous Sparsification. Source: [69].

Require: weight matrix $\mathcal{W}$, soft-mask $\mathcal{S}$, initial value $\mathcal{S}_0$, β increment,initial temperature β_0, backup epoch k, total of epochs n_epochs, total of iterations $n\_iterations$, input $\mathbb{X}$, labels $\mathbb{Y}$

$random\_initialization(\mathcal{W})$
$initialize\_with\_constant(\mathcal{S}, \mathcal{S}_0)$
for iteration $\in n\_iterations$ **do**
  for $epoch \in n\_epochs$ **do**
    if $epoch = k$ **and** $iteration = 0$ **then**
      $\mathcal{W}_k \leftarrow \mathcal{W}$
    end if
    $\hat{\mathcal{M}} \leftarrow \sigma(\beta \times \mathcal{S})$
    $\mathcal{W} \leftarrow \mathcal{W} \odot \hat{\mathcal{M}}$
    $\hat{\mathbb{Y}} \leftarrow \varphi(\mathcal{W}, \mathbb{X})$
    $\min_{\substack{\mathcal{W} \\ \mathcal{S}}} \mathcal{L}(\mathbb{Y}, \hat{\mathbb{Y}}) + \lambda \cdot \|\hat{\mathcal{M}}\|_1$
    $\beta \leftarrow \beta + i$
  end for
  $\mathcal{S} \leftarrow \min(\beta\mathcal{S}, \mathcal{S}_0); \beta \leftarrow \beta_0; \mathcal{W} \leftarrow \mathcal{W}_k$
end for
$\mathcal{W} \leftarrow \mathcal{W} \odot binarize(\mathcal{S})$
$train()$

initially extracts features, which are passed through three branches and two interpolations, detecting objects at three different scales. The main differences on YOLO Nano from YOLOv3 are the replacement of the traditional *Sequential* blocks by Projection-Expansion (PE) modules and Projection-Expansion-Projection (PEP) modules, also adding a Fully-Connected Attention (FCA) module.

One of the optimizations introduced by YOLO Nano are the Depth-Wise Convolutions (DW_C) with a kernel size of 3×3. For instance, in a traditional convolution over an RGB image, a single convolutional filter has three channels, each convolving its respective input channel, and the outputs are summed, resulting in a feature map with one channel. The depth-wise convolution stacks the outputs, resulting in a feature map with three channels. Therefore, a depth-wise convolutional layer has 1/3 the number of parameters of a traditional convolution. We can see the difference between a traditional convolution and a depth-wise convolution in Figure 3.2.

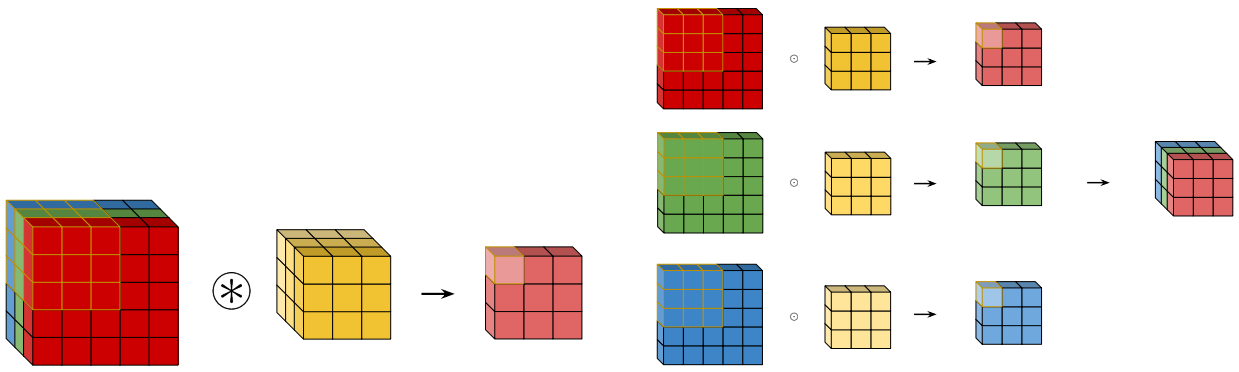


Figure 3.2 – Traditional convolutional operator (left) *vs* depth-wise convolutional operator (right).

Both PE and PEP modules augment the number of feature channels with the first 1×1 convolutional layer. In PE, there is a 3×3 DW convolution, which keeps the number of channels, and a final 1×1 convolution changing the number of channels to the desired

output depth. In PEP, there is one more 1×1 convolution before the 3×3 DW convolution, both keeping the number of channels. Finally, there is a 1×1 convolution with no activation, changing the number of channels to the input depth, and a residual connection summing the input with the features. Both PE and PEP are depicted in Figure 3.3.

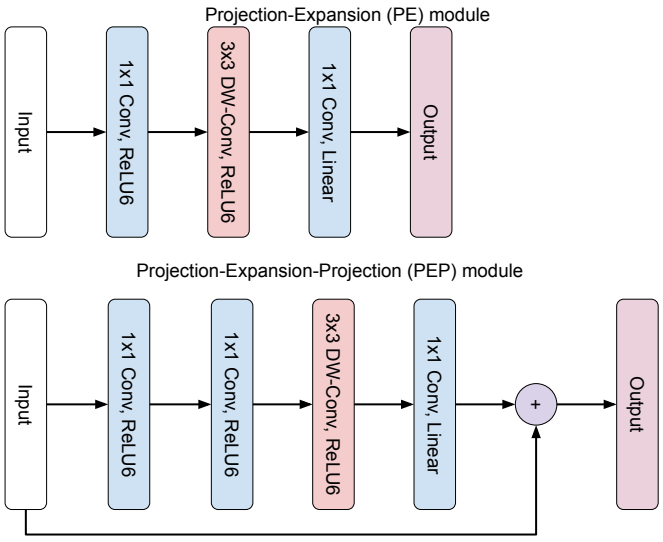


Figure 3.3 – Projection-Expansion (PE) and Projection-Expansion-Projection (PEP) modules. For simplification purposes, we omit the batch-normalization layer between each convolution and activation function.

YOLO Nano also introduces the FCA module as the last operation inside its Darknet feature extractor. The FCA module contains two FC layers to learn the inter-dependencies between channels and attend to more important features. FCA is also used to perform an internal input reduction in a ratio of 8-fold [84]. We can see the FCA structure in Figure 3.4, where the yellow circles symbolize the MLP neurons, and the white circle with an "x" inside means a Hadamard product.

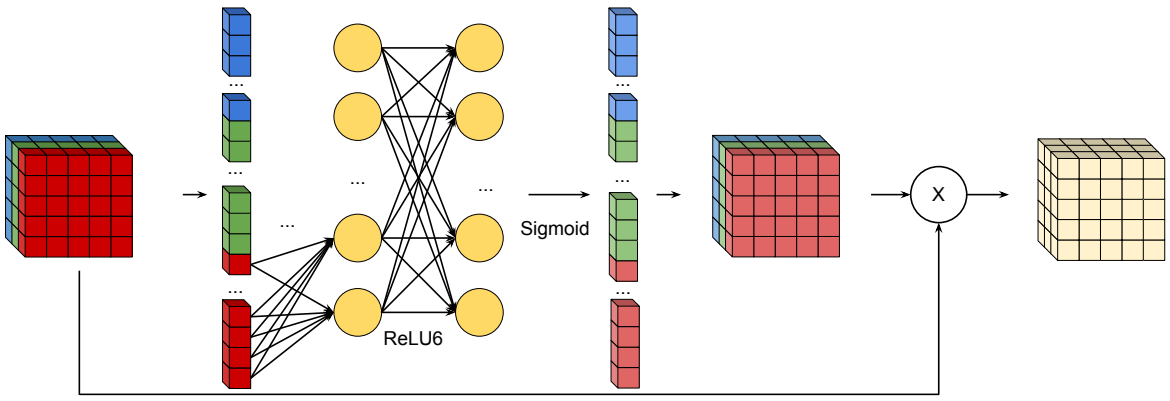


Figure 3.4 – Fully-Connected Attention Module.

3.6 MobileNets and YOLOv3-Mobile

MobileNets are a family of CNN architectures built to focus on parameter and latency reductions and speed up the inference time. In MobileNetV1, Howard *et al.* [38] create the Depth-wise Separable Convolution (DSC) as a basic building block to build an architecture. Traditional convolutional layers both filter and combine inputs into a new set of outputs in only one step. DSC splits the filtering and combining into two layers, a separate layer for filtering and a separate layer for combining. To perform this strategy, DSC provides a depth-wise convolution (as in Figure 3.2, on the right) followed by a 1×1 point-wise convolution. As in PE or PEP modules from YOLO-Nano, each DSC internally increases the number of feature channels while its last layer decreases it.

In MobileNetV2, Sandler *et al.* [68] introduce the linear bottleneck, a 1×1 convolution without activation function to reduce the depth of the features generated by the previous layer. In this manner, the next layer will require fewer convolutional filters, reducing the number of parameters. Another concept introduced is the Inverted Residual Connection (IRC). Traditional CNNs [72, 33, 58, 61] use residual connections between the features with the largest number of channels. IRC uses the residual connection between the features with the least number of channels, always after each bottleneck layer.

In MobileNetV3, Howard *et al.* [37] improve the basic block from MobileNetV2 with the Squeeze and Excite layer, which performs an average pooling, flats the features and forward them to two FC layers. Also, the MobileNetV3 macro-architecture is built with the NetAdapt NAS algorithm, which performs a block-wise search to find the most efficient combination of number of input channels, output channels, and internal expansion ratio of channels. We can see a comparison between the basic blocks of MobileNetV1, MobileNetV2, and MobileNetV3 in Figure 3.5, and the MobileNetV3 architecture in Table 3.1

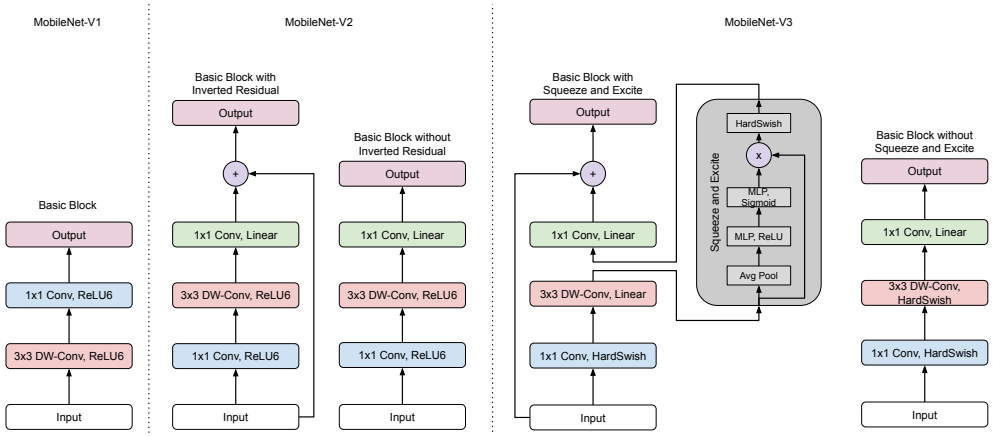


Figure 3.5 – Comparison between the basic blocks from MobileNets V1, V2, and V3. For simplification purposes, we omit the batch-normalization layer between each convolution and activation function.

Table 3.1 – MobileNetV3 architecture. Source: adapted from [37]

Input	Operator	Exp Ratio	Output	SE	NL	Stride
$224^2 \times 3$	conv, 3×3	-	16	-	HS	2
$112^2 \times 16$	BscBlck, 3×3	1	16	-	RE	1
$112^2 \times 16$	BscBlck, 3×3	4	24	-	RE	2
$56^2 \times 24$	BscBlck, 3×3	3	24	-	RE	1
$56^2 \times 24$	BscBlck, 5×5	3	40	✓	RE	2
$28^2 \times 40$	BscBlck, 5×5	3	40	✓	RE	1
$28^2 \times 40$	BscBlck, 5×5	3	40	✓	RE	1
$28^2 \times 40$	BscBlck, 3×3	6	80	-	HS	2
$14^2 \times 80$	BscBlck, 3×3	2.5	80	-	HS	1
$14^2 \times 80$	BscBlck, 3×3	2.3	80	-	HS	1
$14^2 \times 80$	BscBlck, 3×3	2.3	80	-	HS	1
$14^2 \times 80$	BscBlck, 3×3	6	112	✓	HS	1
$14^2 \times 112$	BscBlck, 3×3	6	112	✓	HS	1
$14^2 \times 112$	BscBlck, 5×5	6	160	✓	HS	2
$7^2 \times 160$	BscBlck, 5×5	6	160	✓	HS	1
$7^2 \times 160$	BscBlck, 5×5	6	160	✓	HS	1
$7^2 \times 160$	conv, 1×1	-	960	-	HS	1
$7^2 \times 960$	pool, 7×7	-	-	-	-	1
$1^2 \times 960$	conv 1×1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv 1×1, NBN	-	k	-	-	1

In Table 3.1, **Input**, **Output**, and **Stride** denotes the input, output, and stride sizes, respectively. The squared values in **Input** denote the width/height, while the second number denotes the number of channels. **Operator** is the kind of operation, where *conv* means a default convolution, *BscBlck* means the basic block from MobileNetV3, and *pool* means an Average Pooling. The values after the comma are the kernel size, and *NBN* denotes No Batch-Normalization. **Exp Ratio** is how many times the operator internally increases the number of feature channels. For instance, in the last Basic Block, the input channels is 160, while **Exp Ratio** is 6. Thus, after the depth-wise convolution, there are 160×6 channels in the features. The last convolution from the Basic Block receives a feature map with 960 channels and outputs 160. The checkmarks in **SE** indicates the use or not of the Squeeze and Excite layer. Finally, **NL** is the kind of normalization, where RE indicates the use of ReLU activation function and HS indicates the Hard-Swish activation function.

The Swish activation function (Equation 3.5), in some layers, improves the accuracy of MobileNetV3. To reduce the number of math operations, the Swish function is replaced by the Hard-Swish (Equation 3.6), which approximates the Sigmoidal function with ReLU6.

$$\varphi(x) = x * \sigma(x) \tag{3.5}$$

$$\varphi(x) = x * \frac{ReLU6(x+3)}{6} \tag{3.6}$$

MobileNet architectures are used in image classification, object detection, and image segmentation [37], and its basic blocks are used also in other architectures, such as MnasNet [74] and EfficientNet [75]. Given its relevance, we are also interested in exploring these basic blocks' efficiency at YOLOv3. Thus, based on Table 3.1, we reconstruct YOLOv3 maintaining its macro-architecture and changing the basic blocks from YOLOv3 by those of MobileNetV3.

Looking at Table 3.1, we perceive some patterns, which we maintain to create our YOLOv3-Mobile:

- The first and last layers are default convolutional layers.
- Each Basic Block with stride = 2 has the Squeeze and Excite layer.
- Initial layers have the ReLU activation function. We use the ReLU in all layers from Darknet-53.
- Final layers have the Hard-Swish activation function. We use the Hard-Swish in all layers after Darknet-53.
- Last six Basic Blocks have the Squeeze and Excite layer. We use the Squeeze and Excite layer in the last six layers of each of the three YOLO branches.

3.7 Classical Knowledge Distillation for Object Detection

In a classical KD setting, the Guobin *et al.* [6] approach trains a student model to learn how to mimic the teacher outputs and intermediate features. In their work, both teacher and student are a Faster R-CNN [61], but with different models as a feature extractor. Thus, there are two kinds of loss functions for bounding box regression and object classification: the hard-loss using the ground-truth and the soft-loss using the teacher outputs.

Equation 3.7 computes the classification loss in the KD approach from Guobin *et al.* :

$$\mathcal{L}_{cls} = \mu\mathcal{L}_{hard}(\mathbb{Y}, \hat{\mathbb{Y}}_s) + (1-\mu)\mathcal{L}_{soft}(\hat{\mathbb{Y}}_t, \hat{\mathbb{Y}}_s), \tag{3.7}$$

where $\hat{\mathbb{Y}}_s$ is the student classification set, $\hat{\mathbb{Y}}_t$ is the teacher classification set, $\mathbb{Y}$ is the ground-truth set, and μ is a weight to balance the hard and soft losses. $\mathcal{L}_{hard}$ is the traditional Faster R-CNN classification loss, the Cross Entropy, and $\mathcal{L}_{soft}$ is a class-weighted cross entropy, as in Equation 3.8:

$$\mathcal{L}_{soft}(\hat{\mathbb{Y}}_t, \hat{\mathbb{Y}}_s) = -\sum w_c \hat{\mathbb{Y}}_t \log(\hat{\mathbb{Y}}_s), \tag{3.8}$$

where w_c weights to 1.5 the background classification and 1 the foreground. Since the YOLO architecture families do not treat the background as a class and this background classification does not exist, then we just ignore Equation 3.8 and use a simple Cross Entropy as $\mathcal{L}_{soft}$.

Equation 3.9 computes the bounding box regression loss:

$$\mathcal{L}_{reg} = \ell_1(\mathbb{Y}, \hat{\mathbb{Y}}_s) + \nu\mathcal{L}_{tb}(\mathbb{Y}, \hat{\mathbb{Y}}_t, \hat{\mathbb{Y}}_s), \tag{3.9}$$

where $\mathbb{Y}$ here denotes the bounding box ground-truth set and $\hat{\mathbb{Y}}$ the inference set. The first term of the loss is an ℓ_1 distance between the student prediction and the ground-truth, while the second term ($\mathcal{L}_{ub}$), balanced by the weight ν (default value is 0.5) is the teacher-bounded regression loss, defined in Equation 3.10

$$\mathcal{L}_{tb}(\mathbb{Y}, \hat{\mathbb{Y}}_t, \hat{\mathbb{Y}}_s) = \begin{cases} \left\|\mathbb{Y} - \hat{\mathbb{Y}}_s\right\|_2^2, & \text{if } \left\|\mathbb{Y} - \hat{\mathbb{Y}}_s\right\|_2^2 + m > \left\|\mathbb{Y} - \hat{\mathbb{Y}}_t\right\|_2^2 \\ 0, & \text{otherwise} \end{cases}, \tag{3.10}$$

where the upper bound loss will be the ℓ_2 distance between the student prediction and the ground-truth if this difference plus a margin m is greater than the ℓ_2 distance between the teacher prediction and the ground-truth, otherwise it is zero. The authors create this teacher-bounded loss to mitigate the cases where the teacher prediction return the wrong results. Equation 3.10 makes use of the ℓ_2 distance between the student prediction and the ground-truth, which is the default bounding box regression loss from Faster, but not from our models. For this work, we adapt Equation 3.10 to make use of the GIoU regression loss:

$$\mathcal{L}_{tb}\left(\mathbb{Y}, \hat{\mathbb{Y}}_t, \hat{\mathbb{Y}}_s\right) = \begin{cases} \mathcal{L}_{GIoU}(\mathbb{Y}, \hat{\mathbb{Y}}_s), & \text{if } \mathcal{L}_{GIoU}(\mathbb{Y}, \hat{\mathbb{Y}}_s) + m > \mathcal{L}_{GIoU}(\mathbb{Y}, \hat{\mathbb{Y}}_t) \\ 0, & \text{otherwise} \end{cases}, \tag{3.11}$$

where $\mathcal{L}_{GIoU}$ is the GIoU regression loss described in Section 2.5 with Equation 2.17.

Finally, Equation 3.12 describes the hint loss:

$$\mathcal{L}_{hint} = \ell_1 \left|\mathcal{V}, \mathcal{Z}\right|, \tag{3.12}$$

where $\mathcal{V}$ are features from the teacher's hidden layer and $\mathcal{Z}$ the features from the student's hidden layer. This loss works as if the teacher gives hints for the student to reach the final answer. These features have the same resolution but, in most cases, a different number of channels. Thus, the Hint Layer, a 1×1 convolutional layer, is created to adapt the number of channels of $\mathcal{Z}$ to be equal to $\mathcal{V}$. The authors reported that even in cases where $\mathcal{V}$ and $\mathcal{Z}$ have the same number of channels, the Hint Layer helps the student achieving better results.

Figure 3.6 shows the authors approach adapted to YOLOv3. The black lines indicate the passage of features through the model, while the dashed red lines show the gradient flow. Note that the losses from $\mathcal{L}_{cls}$ and $\mathcal{L}_{reg}$ pass through the entire model, while the loss from $\mathcal{L}_{hint}$ passes only through the Hint Layer and the hidden layers before it.

3.8 KD based on Generative Adversarial Network

Generative Adversarial Networks (GANs) is a framework proposed by Goodfellow *et al.* [27] to train generative models using ANNs. This framework consists of simultaneously training two models in an adversarial manner. While the generator model $\mathcal{G}$ tries to generate

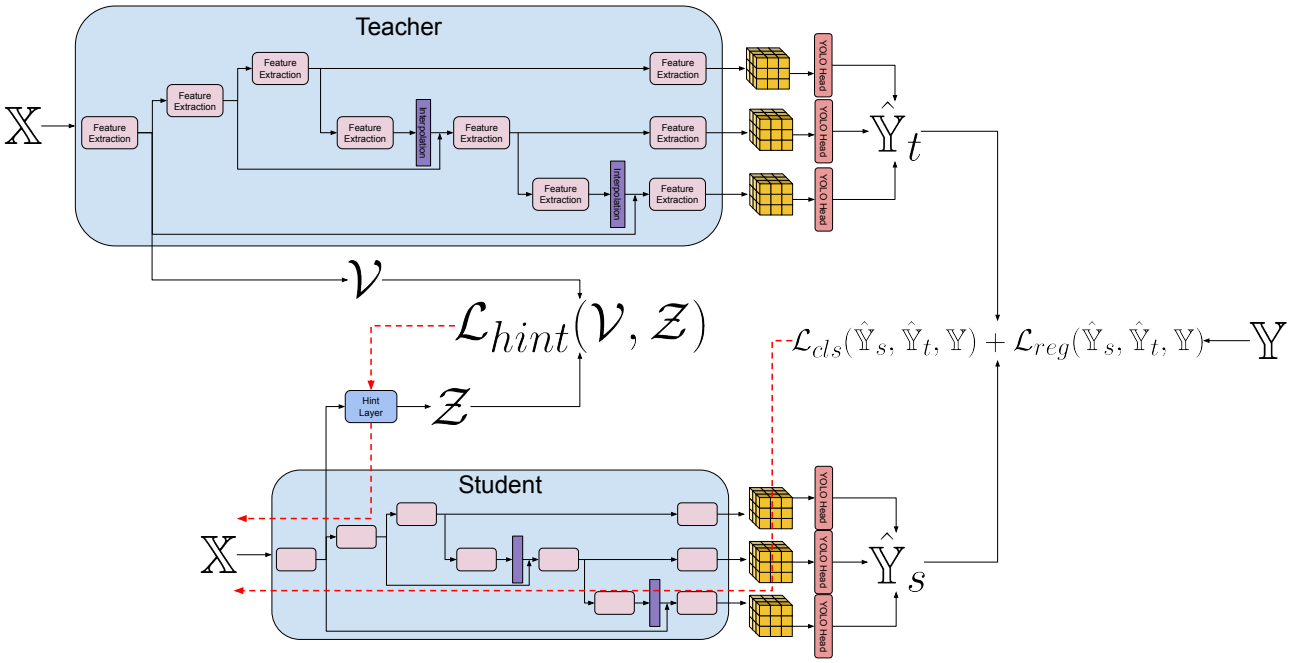


Figure 3.6 – Classical Knowledge Distillation approach.

synthetic images similar to real images, in such a way that they are indistinguishable to the discriminator, the discriminator model $\mathcal{D}$ tries to distinguish which image is real or fake. We can see this framework in Figure 3.7, where z is a prior input noise distribution. While $\mathcal{G}$ tries to learn a function $\mathcal{G} : z \rightarrow \mathbb{X}$, $\mathcal{D}$ tries to learn a function which returns a single scalar $\mathcal{D} : \mathbb{X} \rightarrow 1$ and $\mathcal{D} : \hat{\mathbb{X}} \rightarrow 0$. While the framework trains $\mathcal{G}$ to minimize $\log(1 - \mathcal{D}(\mathcal{G}(z))$, it simultaneously trains $\mathcal{D}$ to maximize $\log(\mathcal{D}(\mathbb{X}))$. At the end of the training, if the procedure has converged, $\mathcal{D}$ returns an output that tends to 1/2 for real and fake data. In other words, $\mathcal{G}$ learns to create realistic images and $\mathcal{D}$ can distinguish them half the time.

Wang *et al.* [81] adopt the GAN framework to perform KD for one-stage object detectors. The approach divides the training into two stages: in the first 2/3 of the epochs, the student trains as a $\mathcal{G}$ from a GAN, and in the last 1/3 of epochs as an object detector. The first stage trains each pair of teacher/student layers as a GAN, where both models generate features with equal shapes. The features generated by the teacher layer is the real input set $\mathbb{X}$, and the features generated by the student layer is the fake input set $\hat{\mathbb{X}}$. To each pair of layers, there is a $\mathcal{D}$ model trying to distinguish which are $\mathbb{X}$ and $\hat{\mathbb{X}}$. Thus, via adversarial training, the student learns to generate features by imitating the teacher features.

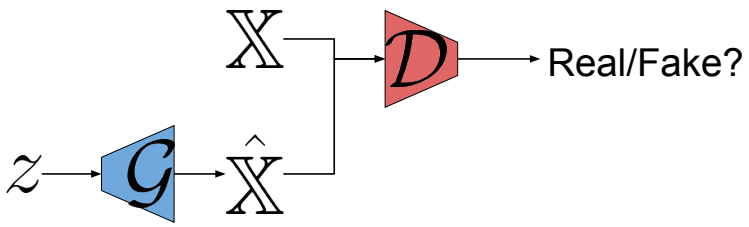


Figure 3.7 – Generative adversarial networks.

With this approach, the authors improve the mAP performance of an SSD [52] by 0.068 mAP points. Figure 3.8 shows the approach adapted to our case. Since our teacher is the YOLOv3 model and our student is the YOLO-Nano model, the KD GAN framework trains three different $\mathcal{D}$ models, using the last feature of each of the three branches, one per $\mathcal{D}$ model. There are six $\mathcal{D}$ models distributed over the teacher/student architectures in the original method. This is not possible in our case because the generated features do not have the same shape, either in resolution or in the number of channels. Comparing Figures 2.9 and 3.8, in Figure 3.8 it is missing the YOLO-Layer, because the goal in the first stage is not the student to imitate the teacher bounding boxes and predictions, but to imitate the teacher features. It is important to remember that YOLO-Layer does not have parameters.

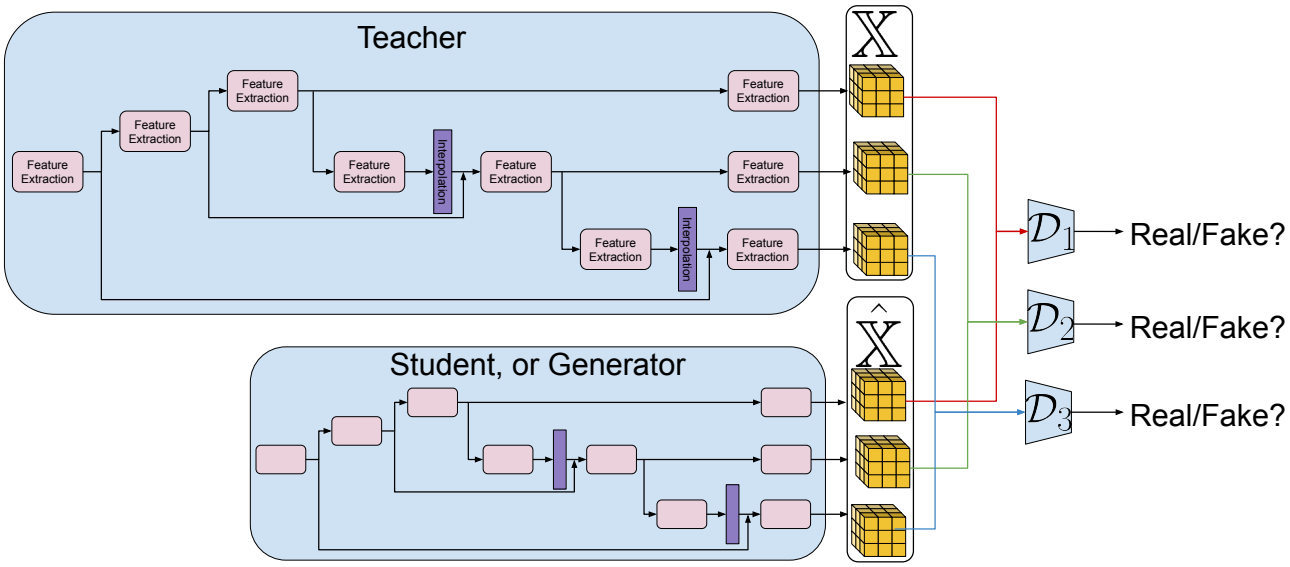


Figure 3.8 – Knowledge Distillation framework using Generative Adversarial Networks.

4. METHODOLOGY

This chapter reports all material necessary to reproduce this work: the datasets that are used in the experiments, the evaluation metrics that are used to evaluate the models, the approach to qualitatively evaluate the distillation, our approach of pruned convolution reconstruction, and the details needed to training each model.

4.1 Datasets

We evaluate the models in two datasets: PASCAL VOC and ExDark, described below:

- **Pattern-Analysis, Statistical Modelling, and Computational Learning Visual Object Classes (PASCAL VOC)** [18]. It is a benchmark for computer vision tasks such as object detection, and promoted challenges from 2005 to 2012. From 2007, there are 20 classes of objects that vary in both size and scenery found, namely: airplane, bicycle, bird, boat, bottle, bus, car, cat, chair, cow, dining table, dog, horse, motorbike, person, potted plant, sheep, sofa, train, and tv monitor. We use the *train/val* splits from PASCAL VOC 2012 [17], containing $5,717$ and $5,823$ images, respectively. Since the labels of the PASCAL VOC 2012 test set is not available, and a server exists for performing the final evaluation, we made the final evaluation on the PASCAL VOC 2007 *test* set, as recommended by the PASCAL VOC 2012 best practices guideline for users who perform multiple experiments with the same algorithm [17]. The split of the PASCAL VOC 2007 *test* set contains $4,952$ images.

- **Exclusively Dark Dataset (ExDark)** [54]. Despite the fact that machine learning models can help us in tasks with low light, such as image enhancement or autonomous cars driving at night, the main computer vision datasets have neglected images of this kind, since the main datasets have at most 2% of low-light images. Thus, Loh and Chan [54] created ExDark, a dataset with exclusively low-light images captured in visible light only. This dataset contains labels similar to PASCAL VOC over 12 classes: bicycle, boat, bottle, bus, car, cat, chair, cup, dog, motorbike, people, and table. There is a total of $3,000$ training set images, $1,800$ for validation, and $2,563$ for testing.

4.2 Evaluation Metrics

Below, we describe the metrics used in this work, the Mean Average Precision, number of parameters, Multiply-Accumulate operations, and storage size.

4.2.1 Mean Average Precision

In this work, we use the Mean Average Precision with 0.5 of threshold (mAP0.5 or mAP@.5) to evaluate the model performance on object detection. This is the standard metric in the literature, based on precision and recall. To compute the mAP@.5, we first need to compute the True Positives (TP), False Positives (FP), and False Negatives (FN):

- TP: measures how many objects the model finds, predicting the correct class and with an IoU between prediction and ground-truth of at least 0.5.

- FP: measures how often the IoU is smaller than 0.5 or when the model finds an object that does not exist in the ground-truth.

- FN: measures how many times the object is in the ground-truth but the model does not predict it.

Computing these measures, then we can compute precision and recall with Equations 4.1 and 4.2:

$$Precision = \frac{TP}{TP + FP} \tag{4.1}$$

$$Recall = \frac{TP}{TP + FN}. \tag{4.2}$$

Precision is the rate between the correctly detected objects among all the detected objects, and recall is the rate between the correctly detected objects among all the objects existing in the ground-truth.

To compute mAP, first we need to compute the average precision. For instance, in a problem with only a single class, we perform the inference over the samples and create a rank sorting the predictions by their confidence. Then, we compute precision and recall at each line of the ranking, from the best to the worst. The definition of average precision is the area under the precision-recall curve. We can see an example in Figure 4.1. In this example, there are five objects in the ground-truth, and the model identifies them all but outputs six objects. Since the precision denominator is the number of predictions, its value at each line equals the Rank value. Since the recall denominator is the number of existing objects, its value is always five. For instance, in the third line, precision = 2/3 and recall = 2/5.

The average precision measures the performance of a single class. Thus, in a dataset with n classes, there are n average precisions, one for each class. The mean average precision, finally, is the mean of all average precisions.

Rank	Correct	Precision	Recall
1	True	1	0.2
2	True	1	0.4
3	False	0.67	0.4
4	True	0.75	0.6
5	True	0.80	0.8
6	True	0.83	1

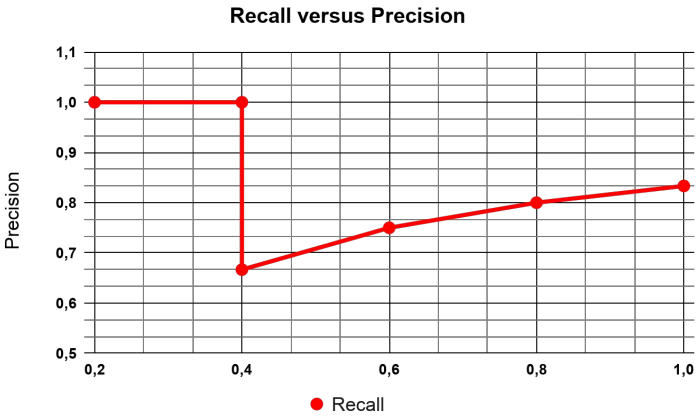


Figure 4.1 – Example of Average Precision.

4.2.2 Number of Parameters

Counting the number of parameters is a way to measure how big a model is, and consequently its storage size, providing a means to understand how costly its inference is. While in parameter pruning the model has an initial number of parameters and a different number after the procedure, in KD it is expected that the student model already has fewer parameters. Counting the number of parameters is relatively simple: *e.g.,* , if a convolutional layer receives a feature map with 32 channels and outputs a feature map with 64 channels, using a kernel with size 3×3, thus there are $32 \times 64 \times 3 \times 3 = 18,432$ parameters if the convolutional layer has no bias, and $32 \times 64 \times 3 \times 3 + 64 = 18,496$ parameters otherwise. If this layer has batch-normalization, we need to sum 64 parameters, related to per-channel learnable means, and more 64 related to the per-channel learnable standard deviations.

4.2.3 Multiply–Accumulate Operation

Roughly speaking, Multiply–Accumulate (MAC) is an operation when the machine computes the product of two numbers and adds that product to an accumulator, like what is done in a linear transformation, in only one step. To count the number of MACs of a convolutional forward pass, let us exemplify with a convolutional layer with a 3×3 kernel ($K = 3^2$), inputting 3 channels ($C_{in} = 3$) and outputting 1 channel ($C_{out} = 1$), with stride 1 and no padding. If the input is a $5 \times 5 \times 3$ volume, then the convolutional kernel convolves the input 9 times ($steps = 9$), since the output will have a 3×3 resolution. Thus, there are 9 steps multiplying the 3×3 kernel at the 3 channels, resulting on $steps \times K \times C_{in} \times C_{out} = 243$ MACs. If the convolutional layer has biases, thus we have the 243 operations plus the bias at each step, resulting in $243 + 1 \times steps = 252$ MACS.

Counting the MAC operations is a way of measuring computational effort to make a prediction in a convolutional network. The higher the number of MACs, the higher the power consumption and the required time to perform a forward pass, which is undesirable mainly in devices powered by a battery or in applications that are constrained in frames per second.

4.2.4 Storage Size

To count the storage size, we simply store a state dictionary from the model and check how many megabytes of space it occupies. State dictionaries are dictionaries containing the parameter name as key and the respective parameter as value. We store all the learnable model parameters, such as the convolutional weights and biases, and the batch-normalization learnable parameters. For automation purposes, we get this size using function `os.path.getsize()` from Python3.

4.3 KD Visualization

It is well known that DL models are black boxes — it is difficult to explain what the model learns and why the model has certain behavior given some input. Consequently, some studies appeared in the literature trying to mitigate this problem and make ANNs less a black box and more a gray box. This is important to understand how KD impacts the student feature extraction. For trying to understand what happens with these models, we use the Gradient-weighted Class Activation Mapping (Grad-CAM) approach from Selvaraju *et al.* [70].

Grad-CAM is a class-discriminative localization approach to generate visual explanations from DL models. After a forward pass, the approach computes the gradients and sets them to 1 for the desired class and to 0 for all the other classes, generating what the authors call as guided backpropagation. Applying this guided backpropagation in the input image generates the **Guided Backprop Image**, generally a gray image with few textures showing what elements in the input activate the output neuron responsible for performing the selected classification.

In parallel with the default gradient, the approach selects the output gradient of some target layer. Averaging this gradient depth-wise, it works as a channel weight. Applying this channel weight in the outputs of the target layer, following by a depth-wise average and a resize to match the input image resolution, it generates the **Grad-CAM Image**, a blue heatmap showing which input regions activate the neurons from the target layer with regard to the evaluated class. Finally, applying an element-wise multiplication between Guided Backprop Image and Grad-CAM Image generates the **Guided Grad-CAM Image**, which also helps to visualize what excites the model neurons. Figure 4.2 shows this workflow.

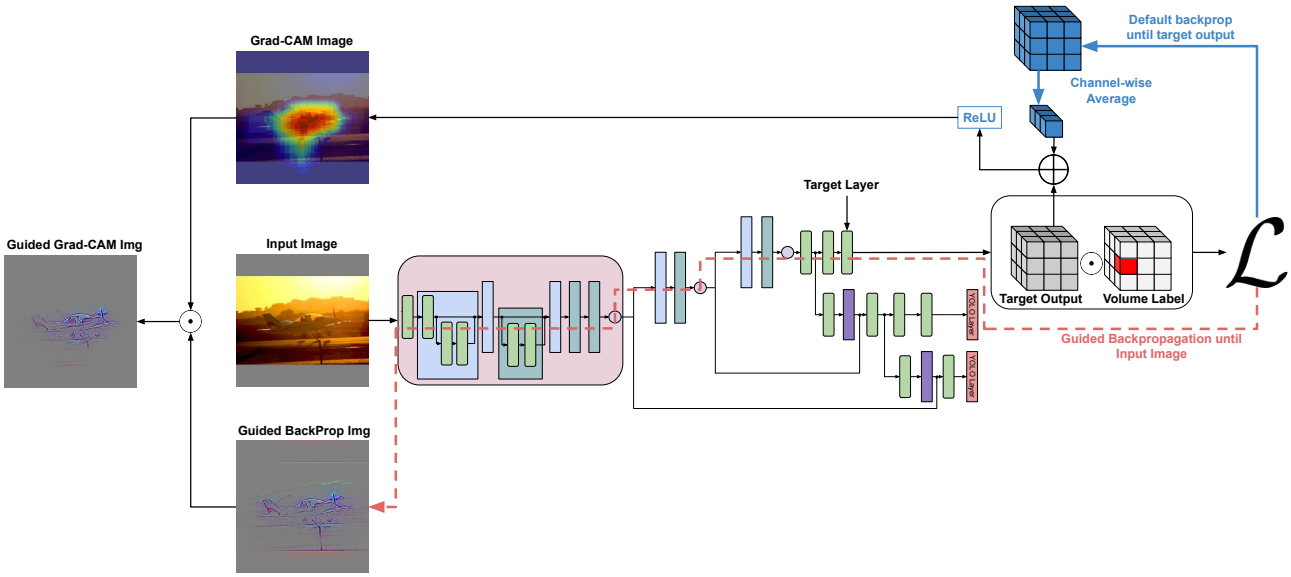


Figure 4.2 – Example of our adapted version of Grad-CAM for YOLOv3.

To apply the Grad-CAM on YOLOv3, we have some issues to address. First, the authors use models that output only one volume, *e.g.,* , a VGG19 outputting the classification of the CIFAR-10 dataset. YOLOv3, otherwise, outputs three volumes, one for each head, as pictured in Figure 2.9. Second, while the model from the previous example outputs only one classification per class, in YOLOv3 there are, for each volume, $S \times S \times 3 \times C$ classifications, that is, C classes times 3 classifications (one per anchor-box) per $S \times S$ grid cell. For instance, in the first head, we have $13 \times 13 \times 3 \times C$ classifications. Moreover, in this volume, there are also the confidence and the bounding box coordinates.

First, we choose only one of the three YOLOv3 volume outputs to deal with these differences. Then, we choose one of the three sub-volumes to evaluate, related to the respective anchor box's classifications. Finally, we select the classification with the highest score of all the $S \times S$ grid cells.

4.4 Sparse Convolution Reconstruction

Despite the fact that non-structured pruning approaches, like LTH and CS, can reduce a large number of parameters in ANNs, there is one more problem to solve: as we can see in Figure 4.3, pruning generates the second structure of the image, which we call a Theoretical Pruned Convolution. However, in practice, what we have is the first structure, the Real Pruned Convolution. What pruning is really doing is only zeroing some weights, still consuming memory storage and processing when performing the multiplications. DL frameworks support storage optimized for sparse matrices, where they store the weights in a floating-point array and the corresponding indexes in unsigned long arrays. This structure can perform matrix multiplication, useful for FC layers. However, for the forward pass in CNNs, it is necessary to

inflate this structure with zeros, reconstructing the Real Pruned Convolution structure and not saving inference MACs.

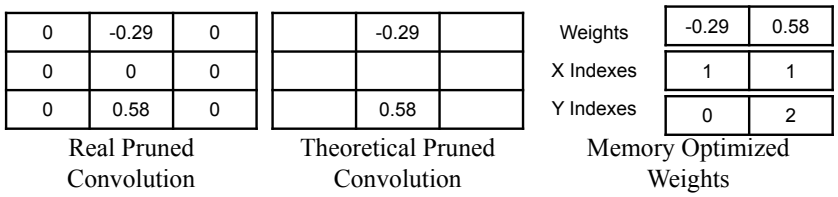


Figure 4.3 – Representations of a pruned convolutional filter.

However, it is possible to perform convolutional inference as matrix multiplication. In this manner, we can use the Memory Optimized Weights structure to save memory consumption and MACs. *E.g.,* , Figure 4.4 shows in the left an input image with 3×3 resolution and two channels, and pruned convolutional layer with a $2 \times 2 \times 2 \times 2$ kernel and only five actual parameters, outputting a $2 \times 2 \times 2$ volume. In the right, there is the same convolution as a matrix multiplication between sparse and dense matrices. The first filter of the convolutional kernel is flattened in the first line of the Flattened Convolution, while the second convolutional kernel is the second line of the matrix. The input image is reshaped in a manner that the number of rows equals the number of columns of the Flattened Kernel. Figure 4.5 givens an example with the same input and convolutional kernel but using one pixel of padding. In both examples, the first line from the flattened output contains the same elements from the first grid channel of the original output. Thus, reshaping the flattened output, we have the same output from the original convolutional inference.

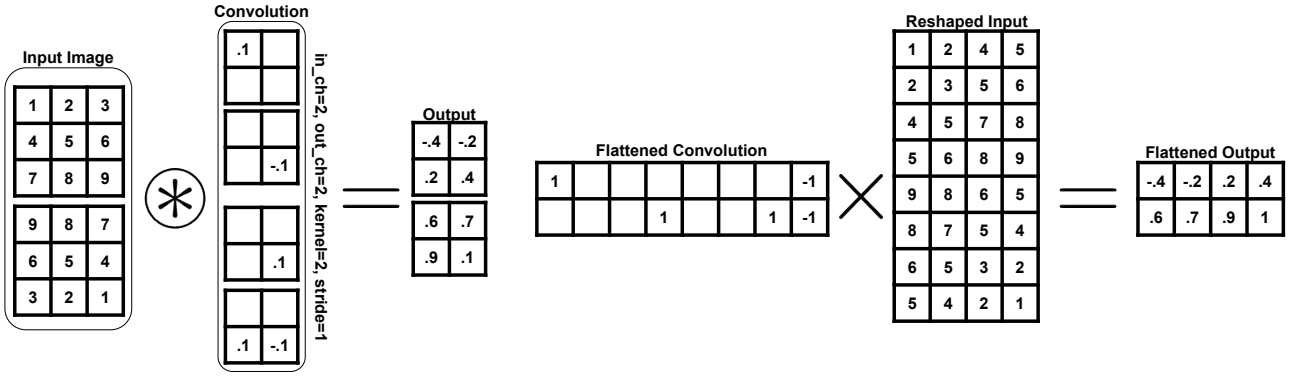


Figure 4.4 – Example of a convolution as matrix multiplication between a sparse and a dense matrix using zero padding.

Reconstructing all the pruned layers with this scheme allows MAC savings, and the MAC computing can be performed by simply multiplying the number of actual parameters by the number of columns on the reshaped input, being $8 \times 4 = 32$ MACs for the matrix multiplication against 64 MACs of the original convolution in the example from Figure 4.4, and $8 \times 16 = 128$ MACs against 256, respectively, in the example from Figure 4.5.

Note that to match the number of rows of the reshaped image with the number of columns of the flattened convolution, many elements of the input are repeated, and depending

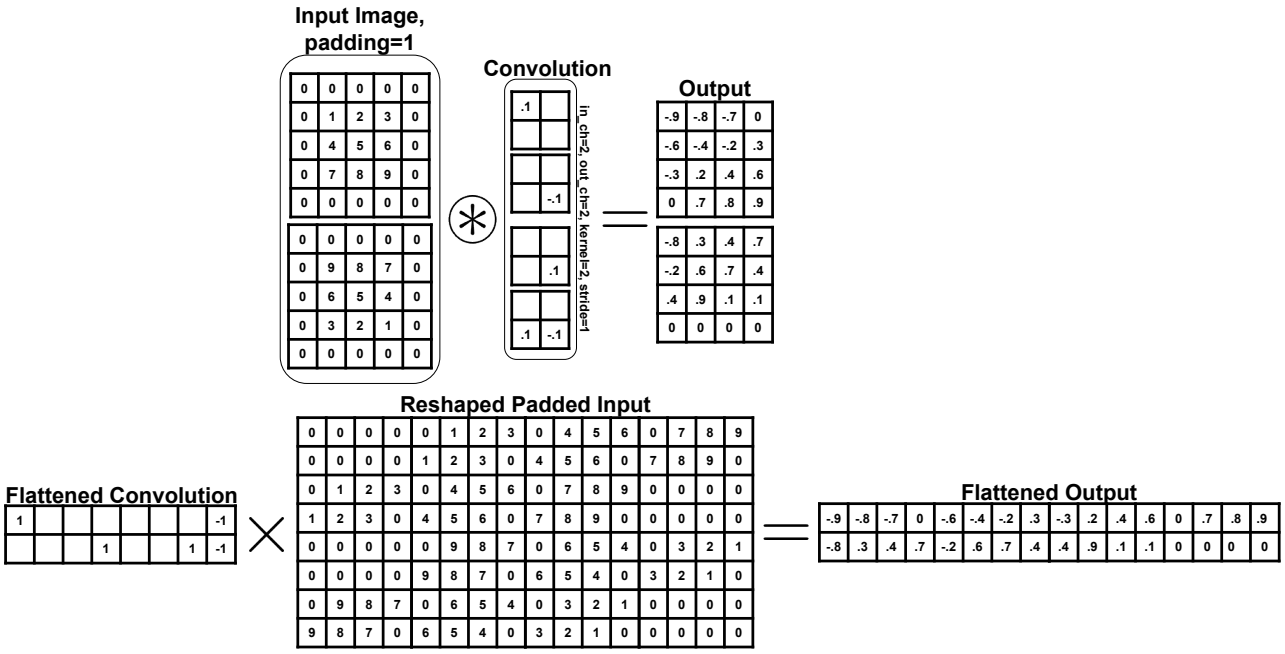


Figure 4.5 – Example of a convolution as matrix multiplication between a sparse and a dense matrix using padding of one.

on the scenario, the MAC reduction will have different proportions. Both examples display a pruned convolution with $100 \times \frac{5}{16} = 31.25\%$ of actual parameters, but containing only $100 \times \frac{4}{8} = 50\%$ of MACs.

4.5 Training Scheme

In this section, we present the training scheme for each evaluated approach. Subsection 4.5.1 shows schemes and hyperparameters from the default training. The next subsections show the specific hyperparameters, mainly the number of epochs and learning rate. The omitted hyperparameters in these subsections are the same as the default training.

4.5.1 Default training

In this work, we use the YOLOv3 implementation from Jocher *et al.* [44]. This implementation manually initializes the bias from the last layer according to Lin *et al.* [51], mitigating the problem of class imbalance, where the majority class, the background, prevails in loss and generates instability in early training. To also mitigate the instability in early training, we use the cosine learning rate decay as in [34]. This approach smoothly reduces the value of the learning rate over time, as in Figure 4.6, which shows a cosine learning rate decay with an initial learning rate of 0.01 in a scheme of 300 epochs.

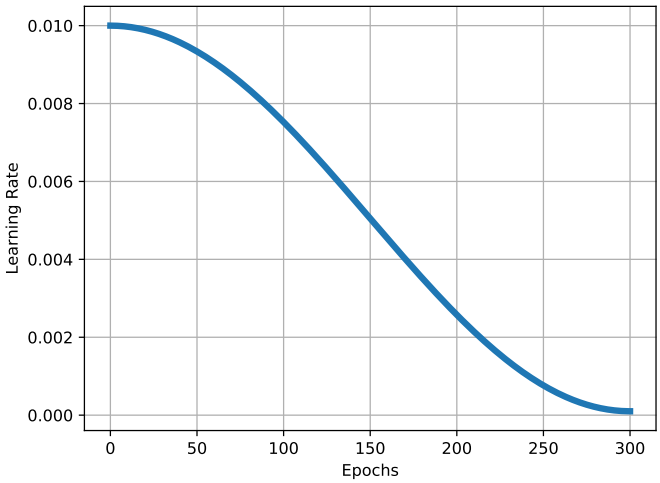


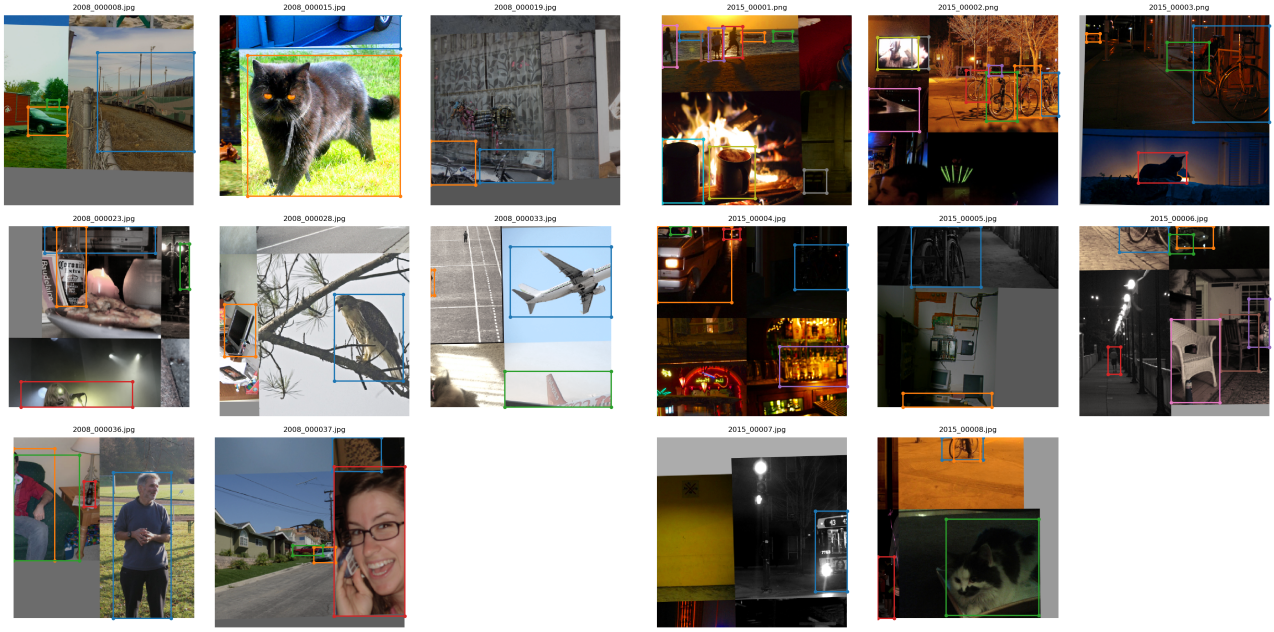
Figure 4.6 – Cosine learning rate decay.

In general, our training protocol makes use of Stochastic Gradient Descendent (SGD) with a learning rate $\eta = 0.01$ for YOLOv3 and YOLOv3-Tiny, and $\eta = 0.001$ for YOLO Nano and YOLOv3-Mobile. To all models, there is a momentum $\alpha = 0.937$, an ℓ_2 weight decay with $\lambda = 4.84e - 4$, and batch size of 64. Regarding data augmentation, we use images with an average size of 416×416, changing the size across multiple scales from 288×288 until 640×640. We build the images over the mosaic idea (building an image with random crops of four random images) with random HSV color jitter and random horizontal flips. We can see examples of images with data augmentation in Figures 4.7(a) and 4.7(b). We train YOLOv3, YOLOv3-Tiny, YOLO Nano, and YOLOv3-Mobile for 300 epochs.

To create our own anchor boxes, we follow Redmon and Farhadi [60]. With the ground-truth bounding boxes of *train/val* splits from PASCAL VOC 2012, we run $k - Means$ with k set to the number of desired anchor boxes. For YOLOv3 and YOLO Nano, which have nine anchor boxes, we have found the following values: $(26; 31)$, $(43; 84)$, $(81; 171)$, $(103; 68)$, $(145; 267)$, $(180; 135)$, $(247; 325)$, $(362; 178)$, $(412; 346)$. For YOLOv3-Tiny with six anchor boxes, we have found the following values: $(26; 31)$, $(43; 84)$, $(81; 171)$, $(103; 68)$, $(145; 267)$, $(180; 135)$, $(247; 325)$, $(362; 178)$, $(412; 346)$. We perform the same idea to ExDark, finding the respective anchor boxes for YOLOv3 and YOLO Nano: $(18; 25)$, $(26; 46)$, $(54; 42)$, $(38; 77)$, $(86; 78)$, $(62; 135)$, $(162; 111)$, $(116; 182)$, $(252; 217)$. For YOLOv3-Tiny, we find the respective anchor boxes: $(18; 28)$, $(35; 49)$, $(82; 66)$, $(53; 108)$, $(116; 141)$, $(232; 196)$.

4.5.2 LTH and CS

For the YOLOv3 models pruned with LTH or CS, we train a total of 300 epochs to learn the mask, and after correctly reinitializing the models, according to the respective algorithm, they train for additional 300 epochs.



(a) Data augmentation example in PASCAL VOC. (b) Data augmentation example in ExDark.

Figure 4.7 – Example of data augmentation, with random HSV color jitter and mosaic over four random images

In LTH with both local and global pruning, we use Algorithm 3.2 with one iteration and a pruning rate ρ of 90%. After the iteration to find the mask, we reinitialize the weights to epoch 10.

In CS, we perform two approaches, pruning with one or three iterations using Algorithm 3.3. In both approaches, we initialize the soft-mask $\mathcal{S}_0 = -0.1$ (trying to perform the same pruning rate as in LTH), a $\lambda = 1e-5$, and a mask learning rate of 0.1 (as in Savarese *et al.* [69]). In the approach with one iteration, we train for 300 epochs, while in the three iterations approach, each iteration trains for 100 epochs. Thus, there are a total of 300 epochs to build the mask in both cases, as in LTH. After the iterations and the binarization of soft-mask $\mathcal{S}$, there is a model reinitialization to epoch 10, and we train it for additional 300 epochs (like in LTH).

4.5.3 Classical KD

We train two architectures as a student network with the approach Guobin *et al.* [6] approach: YOLO Nano and our YOLOv3-Mobile. For both cases, we need to adapt the models, replacing all ReLU6, ReLU, and Hard-Swish activation functions to LeakyReLU, which is the activation function from the teacher. This is necessary because, as the student task is to imitate the teacher, including in the intermediate features, thus these features need to be in the same domain than the teacher's features. This replacing is necessary since ReLU and ReLU6 output only positive values, while LeakyReLU outputs negative and positive values. Hard-Swish also

outputs negative values, but only while $x \in (-0.3, 0)$. In contrast, LeakyReLU outputs negative values for any negative input value. For the teacher, we use the best YOLOv3 model, based on its validation mAP.

For the classification loss, we use Equations 3.7 and 3.8, as described in Section 3.7. To balance the soft and hard classification losses, after the preliminary experiments, we define $\mu = 0.7$. For the bounding box regression loss, we use Equations 3.9 and 3.11. We use the default $\nu = 0.5$ from Guobin *et al.* [6] to balance the soft and hard loss in Equation 3.9. In our preliminary experiments, our student never performed bounding box regression better than the teacher, so the value of m from Equation 3.9 does not influence the results.

4.5.4 KD GAN

We train two architectures as a student network with the KD GAN approach from Wang *et al.* [81], adapting YOLO Nano and YOLOv3-Mobile with LeakyReLU, as described in the previous subsection. In the first stage, the GAN training happens for 200 epochs, while the object detection training happens in the final 100 epochs. We select some hyperparameters based on preliminary experiments, using a subset from PASCAL VOC 2012. For YOLO Nano, we train $\mathcal{G}$ with $\eta = 0.01$ and $\mathcal{D}$ with $\eta = 0.01$. For our YOLOv3-Mobile, we train $\mathcal{G}$ with $\eta = 0.001$ and $\mathcal{D}$ with $\eta = 0.01$. Both $\mathcal{G}$ and $\mathcal{D}$ have the same learning rate decay function with cosine ramp and the other hyperparameters as in the default training scheme.

Based on the preliminary experiments, we had to improve the GAN framework with a series of tricks for training GANs to achieve stability in training. Otherwise, the training collapsed: the model $\mathcal{D}$ reached a state where it returned 1 for both real and fake data ($\mathbb{X}$ and $\hat{\mathbb{X}}$). Consequently, the model $\mathcal{G}$ did not learn to imitate the distribution of real data. These tricks are listed below:

- Consistent batches: each mini-batch contains only either real or fake data [9].
- Avoiding sparse gradients: activation functions as ReLU generate sparse gradients — if a neuron receptive field v is negative, ReLU outputs 0, and the gradient is zero. Thus, neurons from previous layers whose output is directly or indirectly related to this v receive a zero gradient. Like a snowball, this increases and affects $\mathcal{G}$, where few neurons have a gradient to learn. The same occurs when the $\mathcal{D}$ uses Max-Pooling. To avoid these problems, our $\mathcal{D}$ uses only LeakyReLU and Average-Pooling [9].
- Soft labels: if the current mini-batch has real data, we replace label 1 with a randomly sampled value $\in [0.7, 1.2]$. For fake data, we replace label 0 by a randomly sampled value $\in [0.0, 0.3]$, as proposed by Salimans *et al.* [66].

5. RESULTS

In this chapter, we report the results obtained in each dataset that we experiment on, along with a discussion on the trade-offs of each model compression approach.

5.1 PASCAL VOC

In Table 5.1, we show the results of all evaluation measures for all models in the Pascal VOC 2007 test set.

Table 5.1 – Results for all models evaluated at PASCAL VOC 2007 test set.

Model	Training	mAP	Final Params	MACs	Storage (MB)
YOLOv3-Tiny	Default	0.379 ± 0.003	$8,713,766$	$2,753,665,551$	33.29
YOLOv3	Default	0.547 ± 0.012	$61,626,049$	$32,829,119,167$	235.44
YOLO Nano	Default	0.385 ± 0.007	$2,890,527$	$2,082,423,381$	11.38
YOLOv3-Mobile	Default	0.009 ± 0.008	$4,395,985$	$1,419,864,487$	17.59
YOLOv3	LTH Local	0.549 ± 0.009	$6,331,150 \pm 1$	$3,468,547,347 \pm 278$	118.26
YOLOv3	LTH Global	$\mathbf{0.561 \pm 0.009}$	$6,331,114 \pm 1$	$8,796,051,025 \pm 225,877,824$	118.26
YOLOv3	CS 1 It	0.442 ± 0.010	$740,072 \pm 12,161$	$1,137,839,381 \pm 44,191,983$	11.618 ± 0.23
YOLOv3	CS 3 It	0.316 ± 0.015	$\mathbf{421,721 \pm 3,544}$	$\mathbf{618,724,616 \pm 20,611,379}$	$\mathbf{5.544 \pm 0.07}$
YOLO Nano$_{leaky}$	KD fts 79	0.421 ± 0.007	$2,890,527$	$2,098,305,681$	11.38
YOLO Nano$_{leaky}$	KD fts 36, 61	0.408 ± 0.008	$2,890,527$	$2,098,305,681$	11.38
YOLO Mobile$_{leaky}$	KD fts 91	0.253 ± 0.023	$4,395,985$	$1,458,910,247$	17.59
YOLO Mobile$_{leaky}$	KD fts 36, 91	0.244 ± 0.010	$4,395,985$	$1,458,910,247$	17.59
YOLO Nano$_{leaky}$	KD GAN	0.395 ± 0.012	$2,890,527$	$2,098,305,681$	11.38
YOLO Mobile$_{leaky}$	KD GAN	0.311 ± 0.006	$4,395,985$	$1,458,910,247$	17.59

5.1.1 mAP, Parameters and MACs

Regarding mAP, we can see in Table 5.1 that the best models are the original YOLOv3 and the models pruned by LTH. The best model is YOLOv3 with global pruning from LTH, with 0.561 of mAP, equivalent to 102.45% of the YOLOv3 performance with default training. In the second and third places, there is a virtual tie between YOLOv3 with LTH local and with default training, with mAP of 0.549 (or 100.26%), and 0.547, respectively. These results show the effectiveness of both global and local LTH on removing unnecessary parameters, providing 2.45% and 0.26% of mAP improvement, as shown in Table 5.2.

LTH with global pruning outperforms local pruning because the local pruning forces an equal level of pruning at each layer, while global pruning is more flexible in which locations to remove the parameters. Thus, local pruning has a higher probability of removing more sensitive parameters. On the other hand, this layer-wise pruning generates, in general, more sparse layers than global pruning, which favors an extra MAC reduction. Local pruning has 10.57% of MACs against 26.79% from global pruning, which is more than its double. Since the storage size between them is the same, 128.26 MB or 50.23%, and the difference on mAP is

Table 5.2 – Mean relative results for the PASCAL VOC experiments. The evaluation measures are computed regarding the YOLOv3 default training, considering it as 100% of mAP, parameters, MACs, and size.

Model	Training	mAP	Nº Parameters	MACs	Storage
YOLOv3-Tiny	Default	69.16%	14.14%	8.39%	14.14%
YOLOv3	Default	100%	100%	100%	100.00%
YOLO Nano	Default	70.33%	4.69%	6.34%	4.83%
YOLOv3-Mobile	Default	1.64%	7.13%	4.33%	7.47%
YOLOv3	LTH Local	100.26%	10.27%	10.57%	50.23%
YOLOv3	LTH Global	**102.45**%	10.27%	26.79%	50.23%
YOLOv3	CS 1 It	80.67%	1.20%	3.47%	4.93%
YOLOv3	CS 3 It	57.80%	**0.68**%	**1.88**%	**2.35**%
YOLO Nano$_{leaky}$	KD fts 79	76.87%	4.69%	6.39%	4.83%
YOLO Nano$_{leaky}$	KD fts 36, 61	74.50%	4.69%	6.39%	4.83%
YOLO Mobile$_{leaky}$	KD fts 91	46.29%	7.13%	4.44%	7.47%
YOLO Mobile$_{leaky}$	KD fts 36, 91	44.50%	7.13%	4.44%	7.47%
YOLO Nano$_{leaky}$	KD GAN	72.16%%	4.69%	6.39%	4.83%
YOLO Mobile$_{leaky}$	KD GAN	56.86%	7.13%	4.44%	7.47%

minimal, where global pruning has only 2.19% of mAP more than local pruning, it seems that local pruning provides a more interesting trade-off between predictive performance and model compression.

Moreover, the parameter reduction is almost 90% on LTH, but the storage reduction is only about 50%. This happens because in the Memory Optimized Weights, as described in Section 4.4, for each parameter there is a set of unsigned long arrays saving the indexes of these parameters on the inflated parameter (which we named in Section 4.4 as Theoretical Pruned Convolution). This is why the storage size does not decrease linearly with the number of parameters.

The fourth-best performance is from CS with a single iteration, with 0.442 of mAP, or 80.67% of the original YOLOv3 mAP. There is a bigger difference against CS with three iterations, with 0.316 mAP or 57.80%. Since the three-iterations approach executes for 100 epochs per iteration, β increases quickly from the first to the last step, going from 1 to the ceiling value of 200, as in Savarese *et al.* [69]. Thus, the derivative of the sigmoidal function vanishes fast, and the model learns less. It is important to highlight that in both CS cases, the pruning was more aggressive than in LTH, and consequently, the reduction in MACs and storage size is more intense. Looking at Table 5.2, we can see that the single-iteration approach leaves only 1.20% of the parameters, 3.47% of MACs, and 4.93% of storage size against 0.68% of the parameters, 1.88% of MACs, and 2.35% of storage size in the three-iterations approach.

Among the lightweight models, YOLO Nano, our YOLOv3-Mobile, and YOLOv3-Tiny, YOLO Nano is the best choice for any constrained scenario:

- It slightly outperforms YOLOv3-Tiny, having 0.385 and 0.379 of mAP, respectively, or 70.33% and 69.16% of the YOLOv3 mAP performance.
- It has fewer parameters, with 4.69% of the YOLOv3 parameters, against 14.14% from YOLOv3-Tiny.

- It is the lightest model, requiring 11.38% MB, or 4.83%.
- It is the second model with fewest MACs, with 6.34% of the YOLOv3 MACs against 4.33% from YOLOv3-Mobile, who failed to learn and performs very poorly in terms of mAP.

In Tables 5.1 and 5.2, *KD fts* means the classical KD approach from Guobin *et al.* [6], where the following number indicates the i-th teacher layer used to hint, and KD GAN means the KD GAN-based approach from Wang *et al.* [81]. These tables show that both KD approaches have improved the student's mAP performance. YOLO Nano with default training achieves 0.385 of mAP or 70.33% of YOLOv3 performance. With KD, it improves to 0.395 — with a technical tie against the default training — (or 72.16%), 0.408 (or 74.50%), and 0.421 (or 76.87%) using KD GAN, KD fts $36, 61$, and KD fts 79, respectively.

The KD on YOLOv3-Mobile provides the most impressive results: it increased the performance from default training with 1.64% of the YOLOv3 mAP to 44.50%, 46.29%, and 56.86% with KD fts $36, 91$, and KD GAN, respectively. However, none of the cases was sufficient to outperform YOLO Nano. For YOLO Nano, the classical KD approach performs better, while for YOLOv3-Mobile, KD GAN performs better.

We argue that this difference is due to the domain of the generated features: although the activation functions between student and teacher are the same, there is a big macro-architectural difference between YOLOv3 and YOLO Nano. Although YOLO Nano is a NAS model generated inspired on the YOLO family, detecting at three scales, branching the generated features in three paths, and classifying the objects using YOLO Head, YOLO Nano contains significantly fewer layers than YOLOv3, and it does not contain skip connections. Conversely, YOLOv3-Mobile contains the same macro-architecture. Its difference relies only on the micro-architecture, as the layers are different, but the connections between each other, the number of layers, and their order are the same from YOLOv3.

In Figures 5.1 and 5.12, we can see some qualitative comparisons regarding the KD approach on PASCAL VOC. Figure 5.1 shows a Grad-CAM comparison between YOLOv3, YOLO Nano with default training, and YOLO Nano with KD GAN. It shows us a neuron excitation between the three models when they see the bird. The student model with KD GAN seems to make more mistakes than in default training, having more regions with light excitement, denoted by the colors yellow and ocean green.

However, in the Guided Grad-CAM image from Figure 5.2, the bird textures are more distinguishable using the student with KD GAN than with default training. That shows, for instance, that the final neurons of the student with default training are better suited to identifying the bird than the student with KD GAN. On the other hand, due to the different training and activation functions, the gradient flows better through the student with KD GAN, generating the bird textures and indicating that the initial neurons from the student with KD GAN are best suited to identify these fine details than the student with default training.

Figure 5.1 – Learned features visualization with Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD GAN. Image generated using the second YOLO Head and the first anchor box.

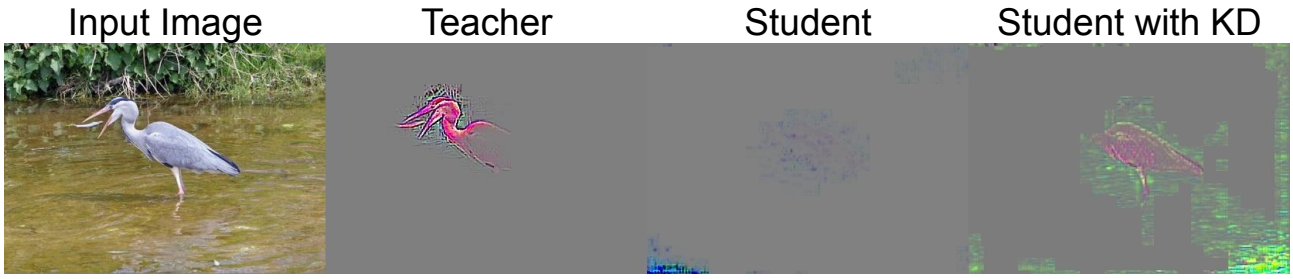


Figure 5.2 – Learned features visualization with Guided Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD GAN. Image generated using the second YOLO Head and the first anchor box.

In Figure 5.3, all models identify the person, but the student with default training identifies with low excitation intensity. The classical KD clearly increased the model's focus on the person. In Figure 5.4, it is possible to see a small part of the person with great details generated by the teacher. On the student with KD, the region increases a bit while the details fade. On the student with default training, there are only contours over the biker.

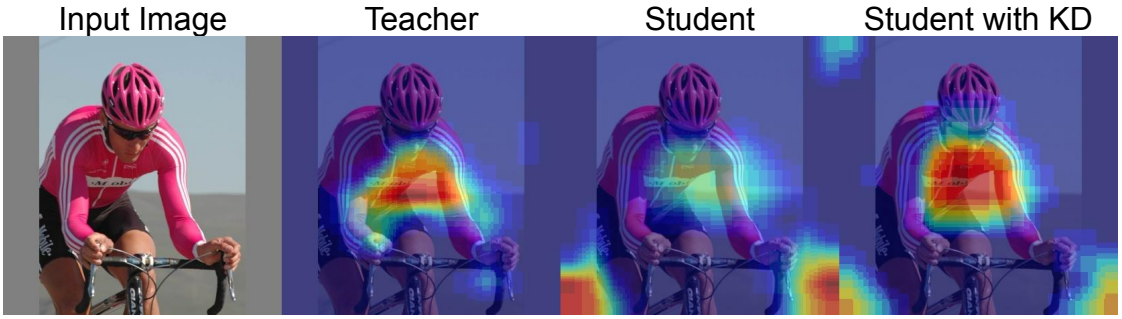


Figure 5.3 – Learned features visualization with Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD fts 36, 61. Image generated using the first YOLO Head and the first anchor box.

Figures 5.5 and 5.6 show a different behaviour. The Grad-CAM image shows that the student with default training and with KD had a greater excitation region than the teacher, and the Guided Grad-CAM image shows a better object reconstruction in those models.

Now, focusing on YOLOv3-Mobile, Figure 5.7 shows a comparison between YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD fts 36, 91. The image

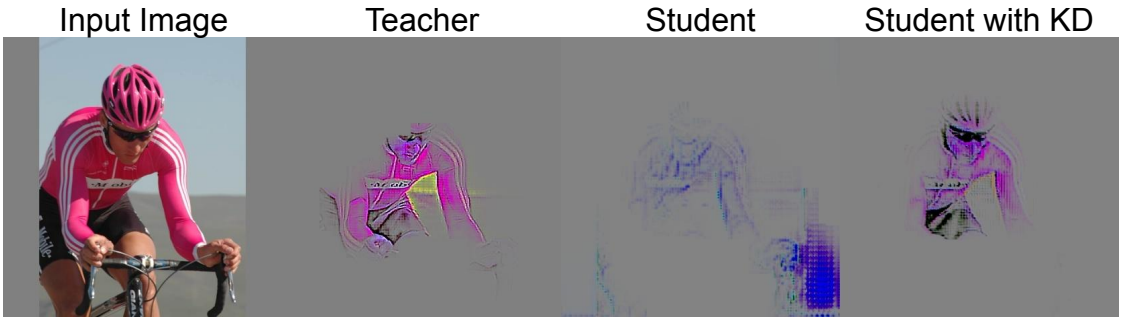


Figure 5.4 – Learned features visualization with Guided Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD fts 36, 61. Image generated using the first YOLO Head and the first anchor box.

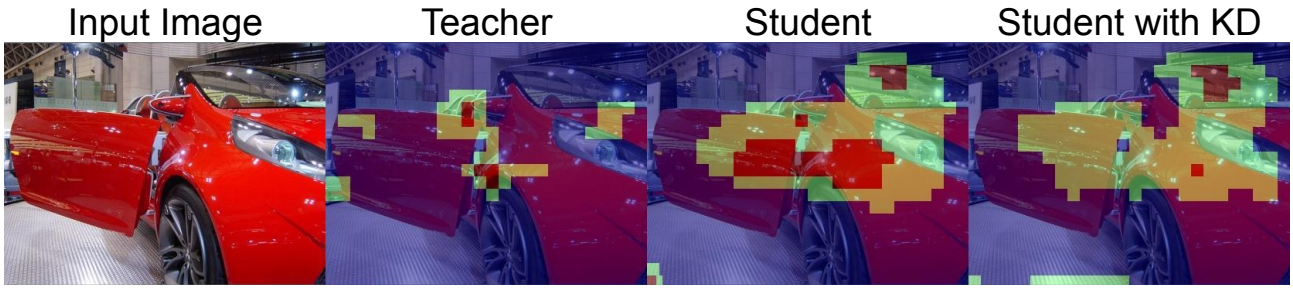


Figure 5.5 – Learned features visualization with Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD fts 79. Image generated using the second YOLO Head and the first anchor box.

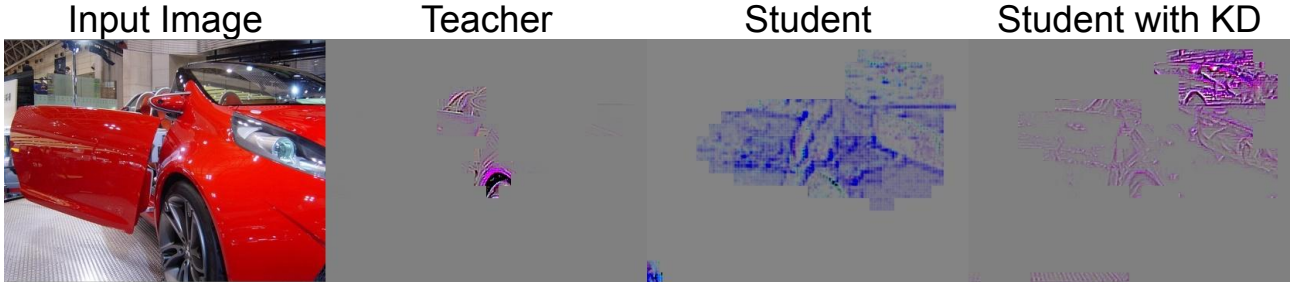


Figure 5.6 – Learned features visualization with Guided Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD fts 79. Image generated using the second YOLO Head and the first anchor box.

shows that the first YOLO Head from YOLOv3-Mobile with default training did not learn to identify objects, displaying random excitation for the person class along the image, while KD sets the focus on the kid and the man standing to the left of the image.

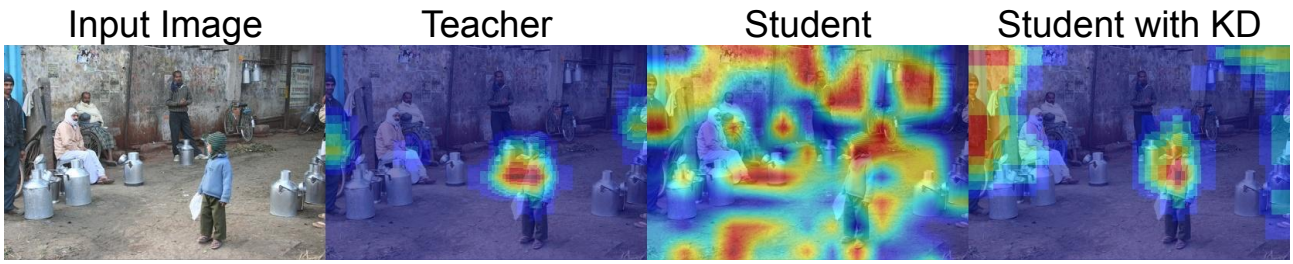


Figure 5.7 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD fts 36, 91. Image generated using the first YOLO Head and the first anchor box.

In Figure 5.8, based on the second YOLO Head, the student with default training did not identify anything, while KD made the model focus on the kid again. Similarly, looking at Figures 5.9 and 5.10, which evaluate KD fts 91, the student with default training on the first YOLO Head displays random excitation for the chair class, while KD make it focus on the left-most chairs, similarly to the teacher. Similar behavior happens with the same input image on the second YOLO Head.

Figure 5.8 – Learned features visualization with Grad-CAM, from YOLOv3, YOLOv3-Mobile in default training, and YOLOv3-Mobile with KD fts 36, 91. Image generated using the second YOLO Head and the first anchor box.

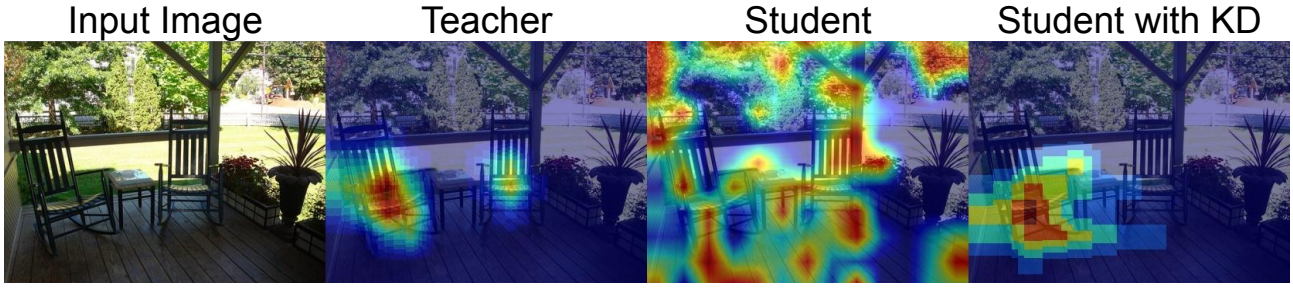


Figure 5.9 – Learned features visualization with Grad-CAM, from YOLOv3, YOLOv3-Mobile in default training, and YOLOv3-Mobile with KD fts 91. Image generated using the first YOLO Head and the second anchor box.

Finally, Figures 5.11 and 5.12 shows that the student with default training presents random excitation in the final neurons from the first YOLO Head, while in the second YOLO

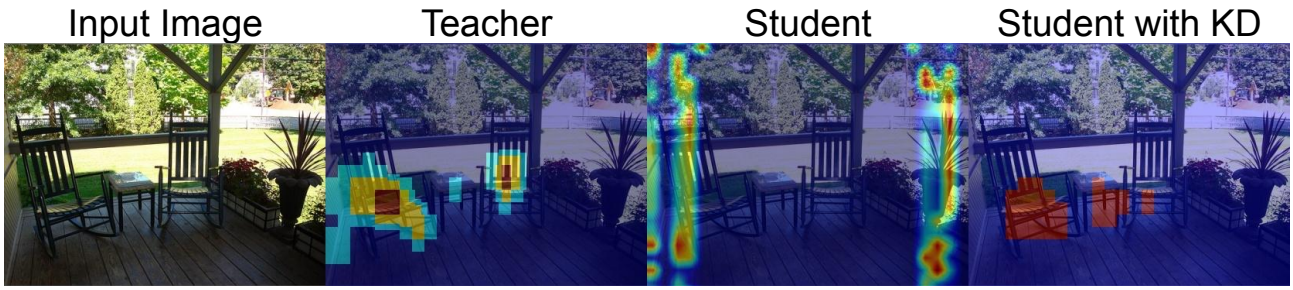


Figure 5.10 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD fts 91. Image generated using the second YOLO Head and the third anchor box.

Head there is no excitation whatsoever. KD GAN make the model focus on the dog, with a similar region to the teacher on the first YOLO Head. On the second YOLO head, there is a larger and stronger excitation on the student than on the teacher.

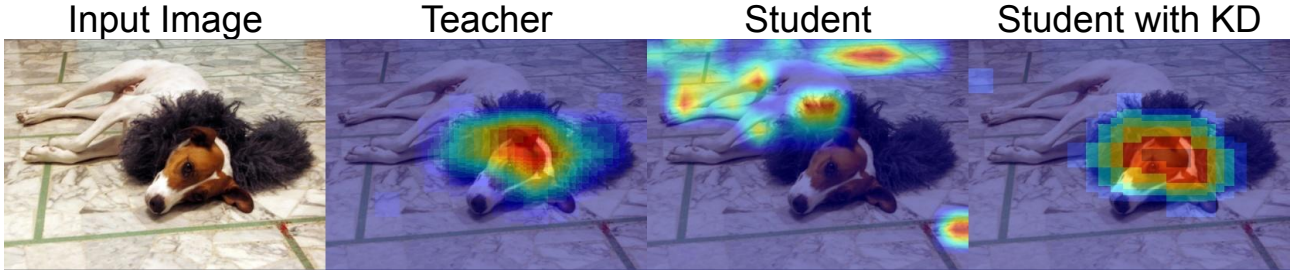


Figure 5.11 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD GAN. Image generated using the first YOLO Head and the first anchor box.

Figure 5.12 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD GAN. Image generated using the second YOLO Head and the third anchor box.

5.1.2 Trade-off

Figure 5.13 shows a heat-map to evaluate the trade-offs on PASCAL VOC. The first bar is the parameter-reduction percentage, which is equal to the complement of the percentages from Table 5.2. Then, it shows the storage-reduction percentage, also computed as the

complement of the values from Table 5.2. Next, there is the mAP percentage with values from Table 5.2, and in the bottom of the figure, there is the MACs-reduction percentage, computed as the complement of the values from Table 5.2. Thus, all the values are on the same scale, where the desired values are the largest, colored in dark blue.

The top-five from the mAP percentage bar has a high difference in color, against all other bars with practically no difference. In this bar, LTH dominates the top-two, where $M6$ is the global pruning and $M5$ the local pruning. However, these models are in the worst-four from the parameter reduction criterion (though still in dark blue), in the worst-three on storage reduction, and in the worst-three on MACs reduction (with dark blue only for LTH local).

Following the top-five on mAP, the fourth and fifth best models are $M7$ (CS with one iteration) and $M9$ (YOLO Nano with KD using the features from the 79-th teacher layer as a hint). These models are also the top-three on parameter reduction and storage reduction, and $M8$ is also the best model on MACs reduction.

It is hard to estimate which is the best model, since there are many possible distinct hardware configurations that can be applied to many different scenarios with different environmental constraints. However, after analyzing the bars from Figure 5.13, we can notice the the following trends:

- Considering a target device with large storage size and computing power, there is a preference for $M5$ (model pruned by LTH local) due to its higher mAP, which is minimally smaller than $M6$ (model pruned by LTH global) and presents more than double reduction on MACs. The MAC reduction for $M5$ is relatively good, with 89.43%.
- In devices where the storage size and computing power is small, there is a preference for $M7$ (model pruned by CS with one iteration) since it is the second-best model on parameter reduction and MACs Reduction, both cases losing to $M8$ (model pruned by CS in three iterations) by an insignificant difference, while being the sixth-best model on storage reduction, losing only to $M8$ and the YOLO Nano versions, with a difference of only 0.1 MB against YOLO Nano.

5.2 ExDark

In Table 5.3, we show the results of all evaluation measures for all models in the ExDark test set.

5.2.1 mAP, Parameters and MACs

The ExDark dataset is more challenging than PASCAL due to the low brightness and visibility images. As we can see in Tables 5.3 and 5.4, all models had a decrease on mAP

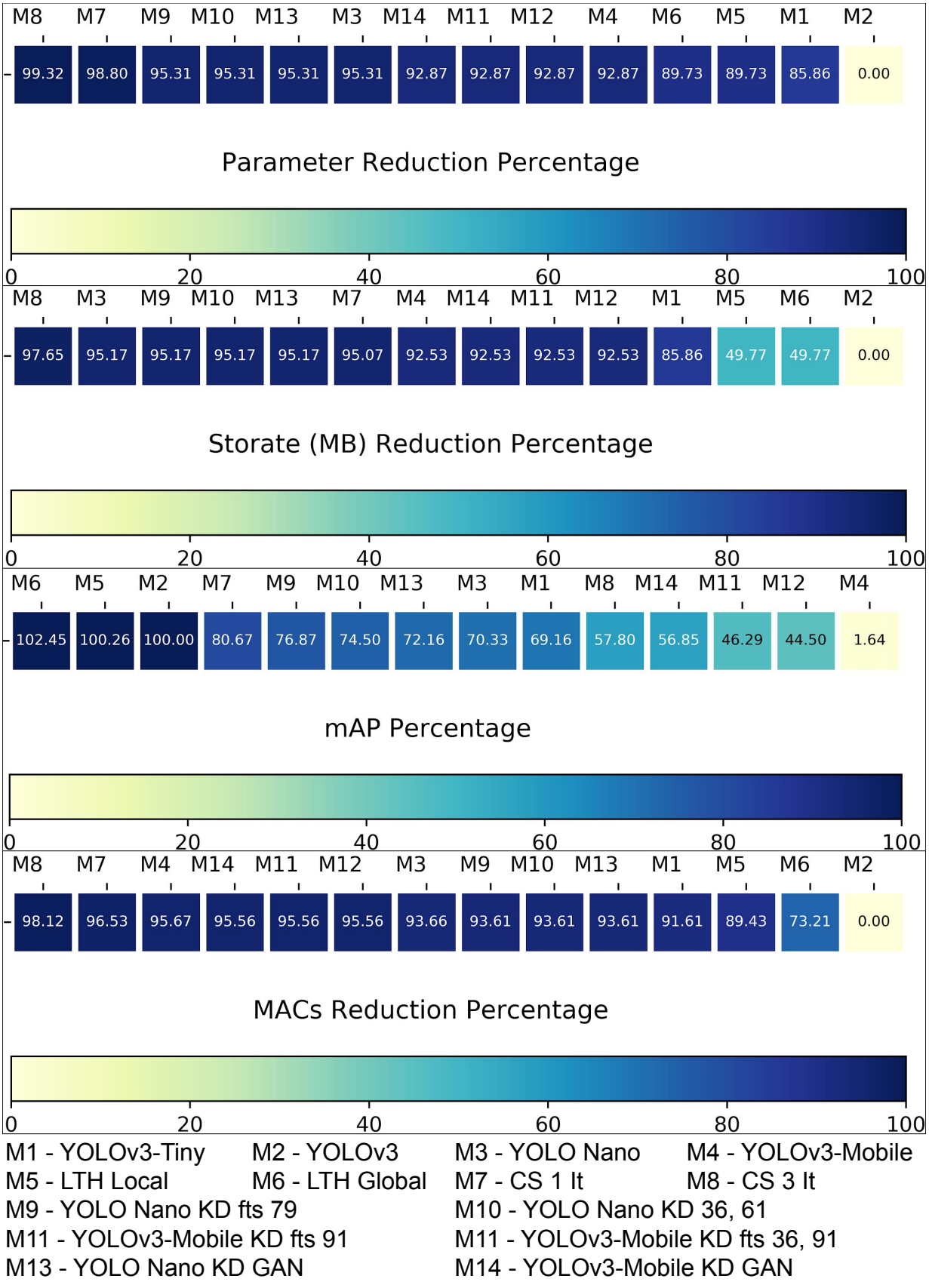


Figure 5.13 – Comparison over Parameters and Storage Reduction, mAP, and MACs Reduction on PASCAL experiments.

Table 5.3 – Results for all models evaluated at the ExDark test set.

Model	Training	mAP	Final Params	MAC	Storage (MB)
YOLOv3-Tiny	Default	0.287 ± 0.020	$8,695,286$	$2,747,415,255$	33.22
YOLOv3	Default	0.453 ± 0.017	$61,582,969$	$32,799,960,583$	235.27
YOLO Nano	Default	0.242 ± 0.013	$2,872,743$	$2,071,460,013$	11.31
YOLOv3-Mobile	Default	0.000 ± 0.000	$4,390,537$	$1,416,145,135$	17.57
YOLOv3	LTH Local	0.461 ± 0.012	$6,288,070 \pm 1$	$3,439,388,763 \pm 278$	118.1
YOLOv3	LTH Global	$\mathbf{0.471 \pm 0.018}$	$6,288,035 \pm 1$	$9,665,082,014 \pm 288,425,550$	118.09
YOLOv3	CS 1 It	0.294 ± 0.012	$525,823 \pm 7,684$	$941,520,024 \pm 58,158,009$	8.188 ± 0.149
YOLOv3	CS 3 It	0.139 ± 0.004	$\mathbf{290,746 \pm 1,638}$	$\mathbf{505,248,788 \pm 15,650,702}$	$\mathbf{3.702 \pm 0.032}$
YOLO Nano$_{leaky}$	KD fts 79	0.303 ± 0.008	$2,872,743$	$2,087,342,313$	11.31
YOLO Nano$_{leaky}$	KD fts 61, 91	0.295 ± 0.010	$2,872,743$	$2,087,342,313$	11.31
YOLO Mobile$_{leaky}$	KD fts 91	0.113 ± 0.021	$4,390,537$	$1,455,190,895$	17.57
YOLO Mobile$_{leaky}$	KD fts 36, 91	0.107 ± 0.005	$4,390,537$	$1,455,190,895$	17.57
YOLO Nano$_{leaky}$	KD GAN	0.254 ± 0.007	$2,872,743$	$2,087,342,313$	11.31
YOLO Mobile$_{leaky}$	KD GAN	0.157 ± 0.005	$4,390,537$	$1,455,190,895$	17.57

performance. Some patterns keep the same while others change. For instance, one pattern that stays the same from PASCAL VOC in ExDark is the best mAP models: the best model is YOLOv3 with global pruning from LTH with 0.471 of mAP, or 104.18%, LTH with local pruning in second place with 0.461 mAP, or 101.79%, and YOLOv3 with default training with 0.453. Global and local LTH increase the mAP with more 4.18% and 1.79%, respectively. Again, both LTH approaches remove around 90% of the parameters, and the local pruning allows a smaller MAC of 10.49% against 29.47% from global pruning, due to the fact that the sparsity is more equally distributed layer-wise. Both models have virtually the same storage size of 50.20%, or 118.1 MB.

Table 5.4 – Mean relative results for the ExDark experiments. The evaluation measures are computed regarding the YOLOv3 default training, considering it as 100% of mAP, parameters, MACs, and size.

Model	Training	mAP	Nº Parameters	MACs	Storage
YOLOv3-Tiny	Default	63.34%	14.12%	8.38%	14.12%
YOLOv3	Default	100%	100%	100%	100.00%
YOLO Nano	Default	53.57%	4.66%	6.32%	4.81%
YOLOv3-Mobile	Default	0.07%	7.13%	4.32%	7.47%
YOLOv3	LTH Local	101.79%	10.21%	10.49%	50.20%
YOLOv3	LTH Global	**104.18%**	10.21%	29.47%	50.19%
YOLOv3	CS 1 It	64.93%	0.85%	2.87%	3.48%
YOLOv3	CS 3 It	30.76%	**0.47%**	**1.54%**	**1.57%**
YOLO Nano$_{leaky}$	KD fts 79	67.01%	4.66%	6.36%	4.81%
YOLO Nano$_{leaky}$	KD fts 61, 91	65.10%	4.66%	6.36%	4.81%
YOLO Mobile$_{leaky}$	KD fts 91	24.88%	7.13%	4.44%	7.47%
YOLO Mobile$_{leaky}$	KD fts 36, 91	23.65%	7.13%	4.44%	7.47%
YOLO Nano$_{leaky}$	KD GAN	56.13%	4.66%	6.36%	4.81%
YOLO Mobile$_{leaky}$	KD GAN	34.65%	7.13%	4.44%	7.47%

Another pattern that also holds is the YOLOv3 pruned by CS with one iteration, with 0.294 mAP or 64.93% of the default YOLOv3 performance, outperforming CS with three iterations, with 0.139 mAP or 30.76%. Both models also lead on the number of parameters, with 0.47% for the three-iterations approach and 0.85% for the single-iteration approach; on

the number of MACs, with 1.54% and 2.87%; and on storage size, with 1.57% and 3.48%, respectively.

Among the lightweight models, YOLO Nano, our YOLOv3-Mobile, and YOLOv3-Tiny, there is a little difference comparing with the PASCAL results: YOLO Nano keeps its place as providing the fewer amount of parameters, MACs, and storage size when compared with YOLOv3-Tiny, with 4.66% versus 14.12% of the parameters, 6.32% versus 8.38% of MACs, and 4.81% versus 14.12% of size, respectively. However, in ExDark, YOLOv3-Tiny achieves a higher mAP than YOLO Nano: 0.287 of mAP, or 63.36%, against 0.242 mAP or 53.57%. YOLO Nano has almost 33.04% of the YOLOv3-Tiny parameters and 84.32% of YOLOv3-Tiny mAP. Despite having a close mAP and a considerable advantage in the number of parameters, the trade-off preference here is harder to identify, unlike in the PASCAL experiments.

In ExDark, KD also improves all the student's performance, as in PASCAL VOC. KD on YOLO Nano improves its mAP performance in such a manner that the difficult trade-off decision between YOLO Nano and YOLOv3-Tiny described previously disappears — KD brings the YOLO Nano mAP from 0.242, or 53.57% of YOLOv3 mAP performance, to 0.254 (or 56.13%), 0.295 (or 65.10%), and 0.303 (or 67.01%), with KD GAN, KD fts $61, 91$, and KD fts 79, respectively. Although KD GAN improves the performance (with a virtual tie), it was not enough to beat YOLOv3-Tiny, unlike KD fts $61, 91$, and KD fts 79. The classical KD makes YOLO Nano better than YOLOv3-Tiny in all metrics, even with the increase in MACs due to the LeakyReLU activation function replacing ReLU6.

KD on YOLOv3-Mobile also provides impressive results — it changed the performance from default training with 0.07% of the YOLOv3 mAP to 23.65%, 24.88%, and 34.66%s with KD fts $36, 91$, and KD GAN, respectively. Comparing the mAP between YOLOv3-Mobile with default training and KD GAN, there is a mean mAP improvement of 522.33 times. In YOLO Nano, KD GAN brings an improvement of only 0.05 times. However, none of the cases was enough to outperform any YOLO Nano version, and as in PASCAL, the classical KD brings the best improvements for YOLO Nano, while KD GAN brings for YOLOv3-Mobile. Since the YOLOv3-Mobile originally has poor performance, it is more difficult for the model to imitate the teacher outputs. On the other hand, since its macro-architecture is the same as the teacher and uses the same activation functions, it is easier to distill the final teacher features. We can see some qualitative comparisons in Figures 5.14 to 5.26.

In the specific example of Figures 5.14 and 5.15, the KD GAN on YOLO Nano decreased the model's performance. The Grad-CAM image shows that the neurons from the second YOLO Head and first anchor box got excited seen the bicycle in the teacher and student with default training. The teacher model is more accurate than the student: while there is a big bicycle piece from the teacher in the Guided Grad-CAM image at Figure 5.15, the student with default training has a tiny blur. With KD, the blur disappears. Figure 5.16 depicts an example where KD GAN improves the student. We can see most contours of the bicycle in the

teacher. In the student model with default training, most of the contours are there, but in very smooth lines. With KD GAN, the line intensity increases, and the bicycle is more visible.

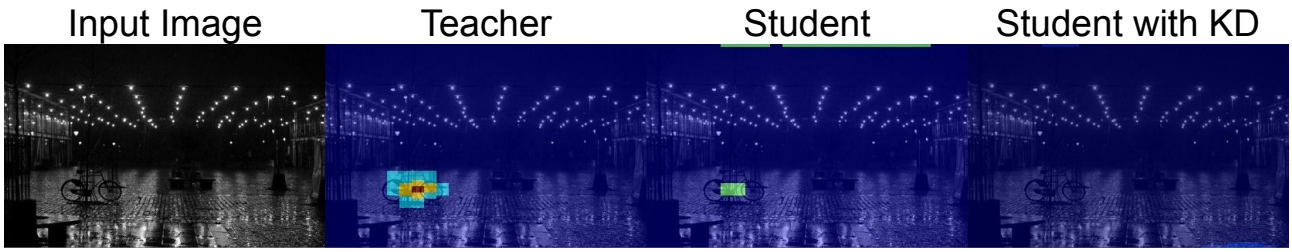


Figure 5.14 – Learned features visualization with Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD GAN. Image generated using the second YOLO Head and the first anchor box.

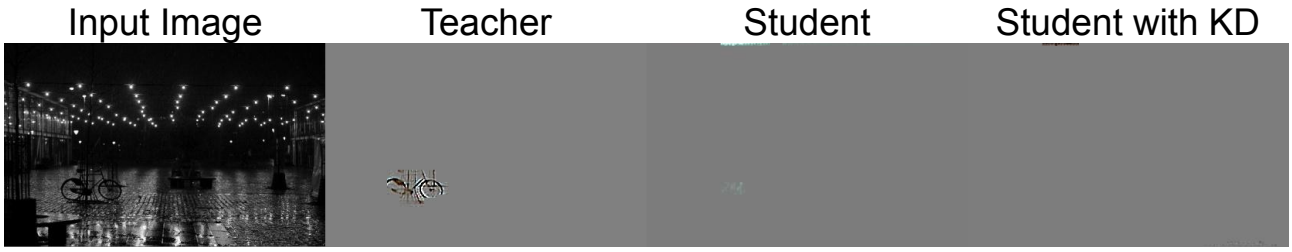


Figure 5.15 – Learned features visualization with Guided Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD GAN. Image generated using the second YOLO Head and the first anchor box.

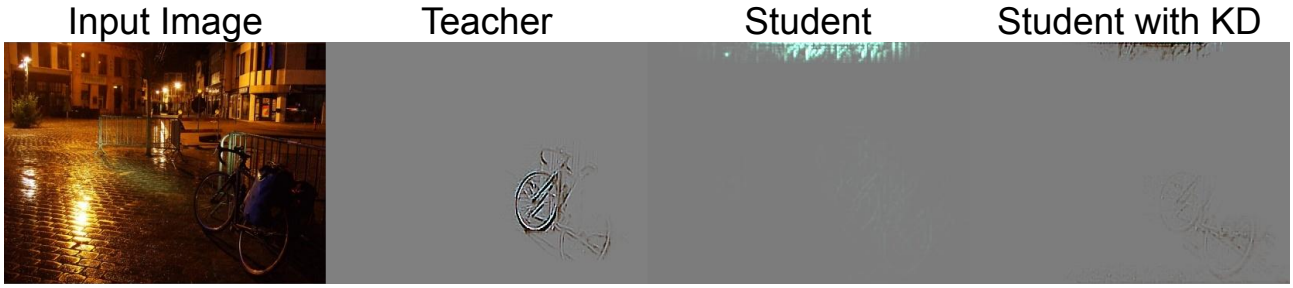


Figure 5.16 – Learned features visualization with Guided Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD GAN. Image generated using the first YOLO Head and the first anchor box.

In Figures 5.17 and 5.18, we can see an example with YOLO Nano trained with KD fts 61, 91. Here, the KD student had its neurons (second YOLO Head and the first anchor box) excited when seeing the bicycle, in a larger and more intense region. In the Guided Grad-CAM image from Figure 5.18, we can see a smaller bike region that does not exist on YOLO Nano with default training. Note that YOLO Nano achieves a higher mAP with KD fts 79 than KD fts 61, 91. Therefore, better visual improvement is most expected in the next example than in the current.

Figure 5.19 shows that the teacher neurons from the second YOLO head and the first anchor box excite when they see the bicycle. In YOLO Nano, this also happens but in a smaller region of the bicycle. KD enlarges this excitation region. Combining these excitation regions with the guided backpropagation, we have the Guided Grad-CAM image, pictured in

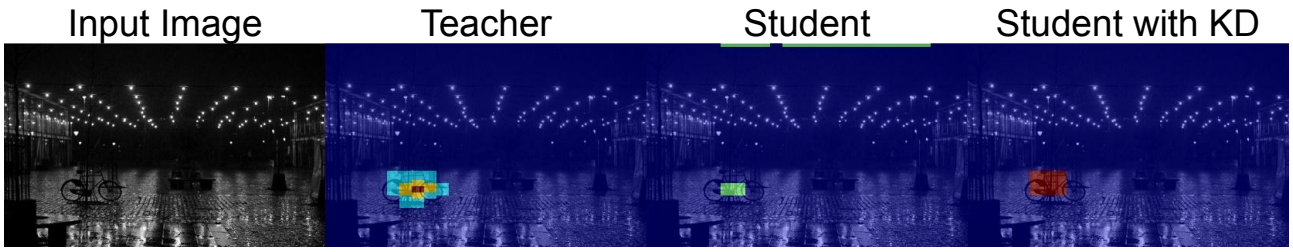


Figure 5.17 – Learned features visualization with Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD fts 61, 91. Image generated using the second YOLO Head and the first anchor box.

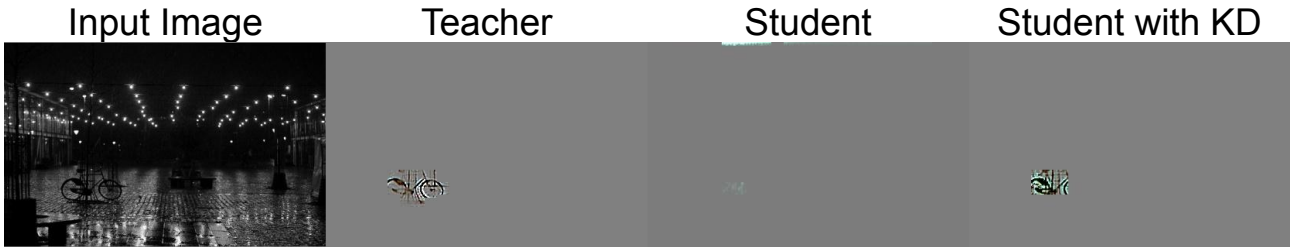


Figure 5.18 – Learned features visualization with Guided Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD fts 61, 91. Image generated using the second YOLO Head and the first anchor box.

Figure 5.20. We can easily see the bicycle in both teacher and student with KD, but this does not happen with the student with default training.

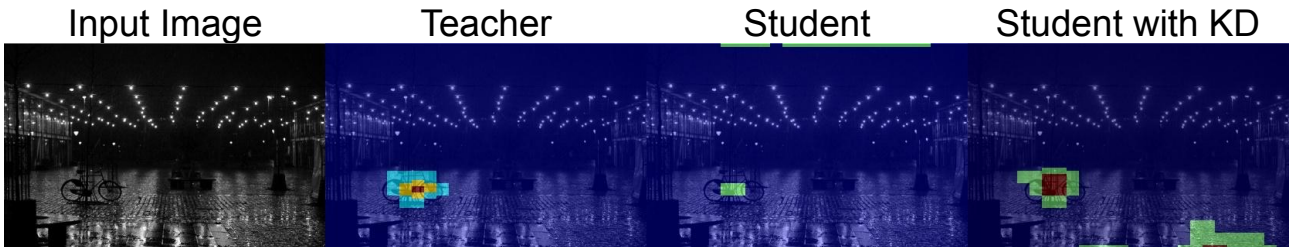


Figure 5.19 – Learned features visualization with Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD fts 79. Image generated using the second YOLO Head and the first anchor box.

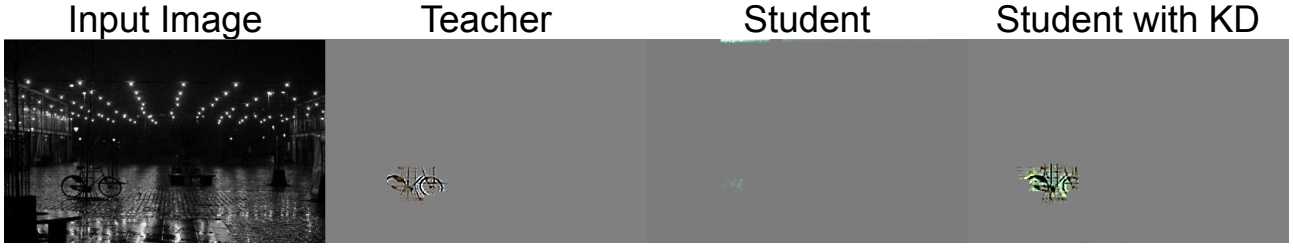


Figure 5.20 – Learned features visualization with Guided Grad-CAM for YOLOv3, YOLO Nano with default training, and YOLO Nano with KD fts 79. Image generated using the second YOLO Head and the first anchor box.

Figure 5.21 depicts the feature visualization using the Grad-CAM image, comparing YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with the KD approach using the teacher features from layers 36 and 91 on Hint Loss. We can see that the teacher

sees a bicycle that the student with no distillation cannot see. With KD, the student neurons excite with the bicycle and some texture on the floor. In Figure 5.22, the first head from the student with KD slightly learns to imitate the teacher, identifying the school desks, while the student with default training does not detect them.

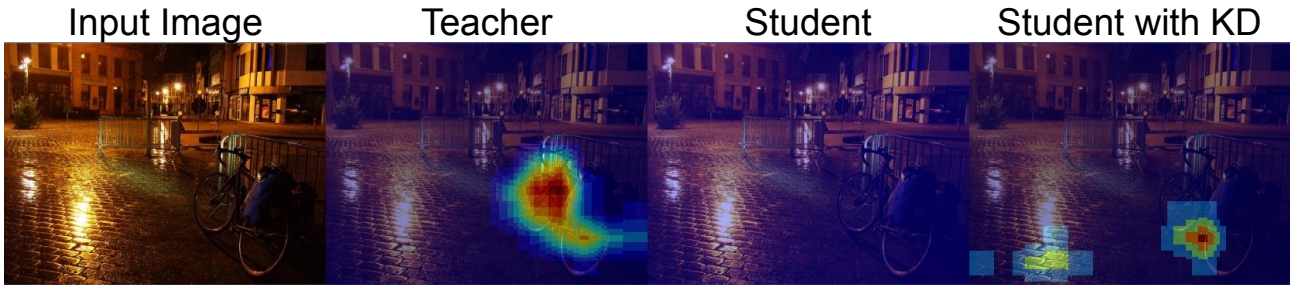


Figure 5.21 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD fts 36, 91. Image generated using the first YOLO Head and the third anchor box.

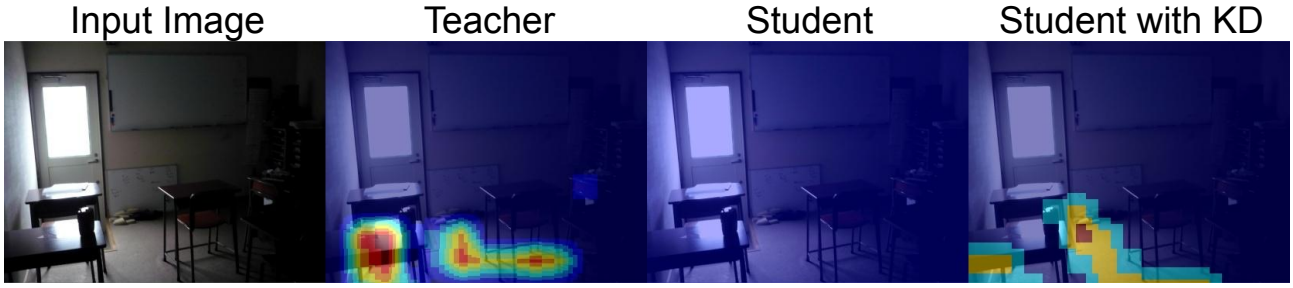


Figure 5.22 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD fts 36, 91. Image generated using the first YOLO Head and the first anchor box.

Figure 5.23 provides the Grad-CAM Image comparing YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile using KD with teacher features from layer 91. The teacher sees a bicycle and a group of chairs and tables in the first head, while YOLOv3-Mobile cannot see anything. After KD, the student model perceives the chairs and tables.

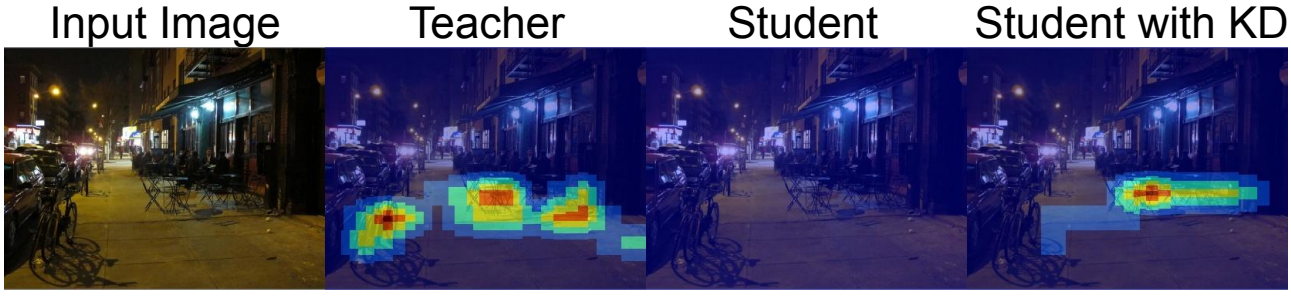


Figure 5.23 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD fts 91. Image generated using the first YOLO Head and the first anchor box.

In Figure 5.24 there is an interesting behaviour. This image was generated using the motorcycle class. Both models identify the motorcycle. However, the teacher focused on the

motorcycle but had a small region with low excitation intensity in the background. On the other hand, the KD student had a large region with high excitation intensity on the motorcycle but covering part of the human's legs.

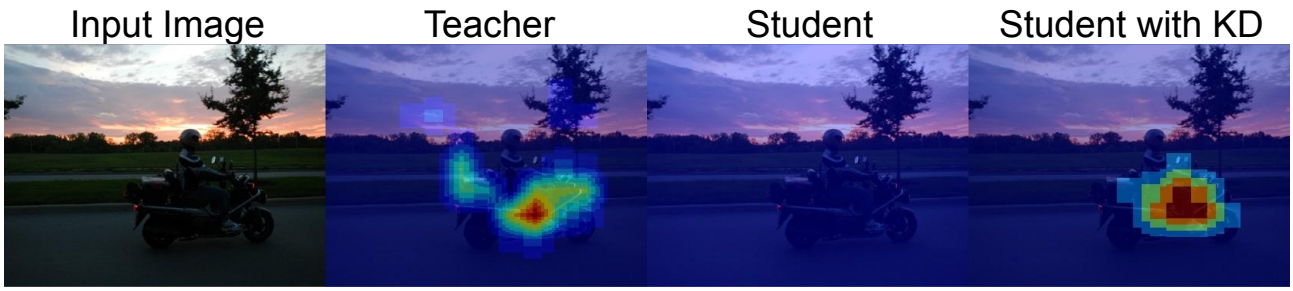


Figure 5.24 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD fts 91. Image generated using the first YOLO Head and the first anchor box.

In the final examples, Figure 5.25 shows the Grad-CAM Image comparing YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile using KD GAN. In the previous example, the student with KD learns to detect the chairs and tables but misses the bicycle. Here, there is smaller excitation in the chairs/tables region, but now the student sees the missed bicycle. In Figure 5.26, all models correctly identify the bus. However, the KD GAN decreased the region of excitation of the student's neurons, centering it more on the object, and increased its intensity, being more reddish and less yellowish, as in the teacher.

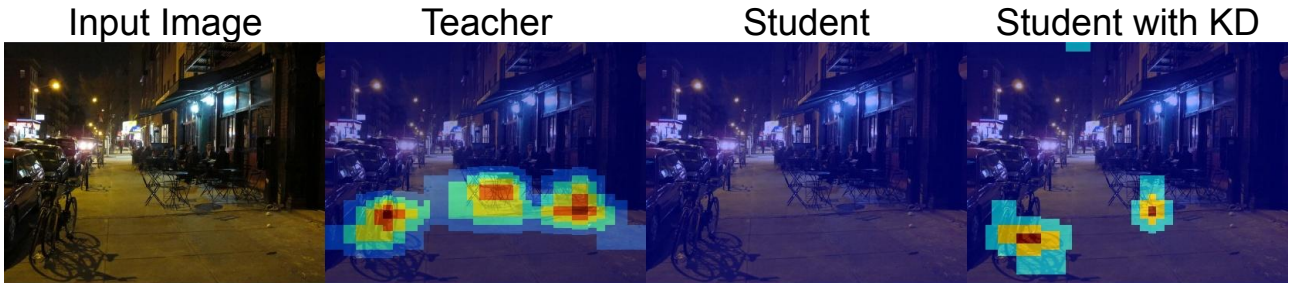


Figure 5.25 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD GAN. Image generated using the first YOLO Head and the third anchor box.

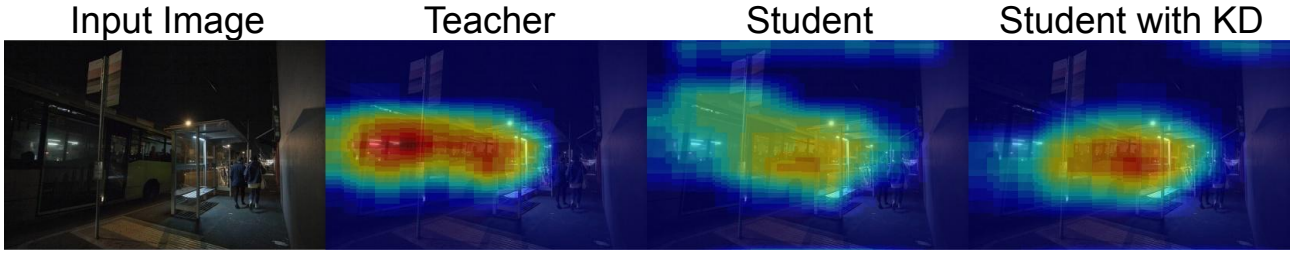


Figure 5.26 – Learned features visualization with Grad-CAM for YOLOv3, YOLOv3-Mobile with default training, and YOLOv3-Mobile with KD GAN. Image generated using the first YOLO Head and the third anchor box.

5.2.2 Trade-off

Figure 5.27 shows a heat-map to evaluate the trade-offs on ExDark Dataset. As in the trade-off evaluation on PASCAL VOC, Figure 5.27 is based on Table 5.4 and all the values are on the same scale, where the desired values are the largest, colored in dark blue.

As in PASCAL VOC, with exception of mAP, all the bars have dark blue in at least the ten first places, showing that the difference between them is not so large. The LTH models $M5$ and $M6$ also lead the mAP bar as was the case in PASCAL VOC, but the CS with one iteration ($M7$) falls from the fourth position to the sixth position, leaving its space for the climb of YOLO Nano with KD on the classical approach, with both schemes of hint layers (models $M9$ and $M10$). This happens because the pruning on ExDark was more aggressive than on PASCAL VOC, which increased the difference between CS and YOLO Nano on all other metrics, where in PASCAL VOC, models $M8$ and $M7$ are the top-two on MACs reduction and parameter reduction, while in ExDark they are also on storage reduction.

Notwithstanding, models $M7$ and $M8$ paid the cost of losing some mAP, with $M7$ on the sixth mAP position, as described before, and $M8$ is four positions below (with less than half of the mAP percentage).

The remaining patterns are the same as on PASCAL VOC. Finally, we describe below some considerations about the trade-off on ExDark:

- As in PASCAL VOC, in devices with larger storage size and computing power, model $M5$ (pruned by LTH local) is more desirable due to its higher mAP, minimally smaller than $M6$ (model pruned by LTH global), with only 2.39% less mAP, and with more than double of MACs reduction than $M6$. The MAC Reduction for $M5$ is relatively good, with 89.51%.

- In devices with smaller storage size and computing power, the best approach is not so clear, with a decision to be made between $M7$ and $M9$ (models pruned by CS with one iteration and YOLO Nano with KD fts 91, respectively). $M7$ is the second best model on parameter reduction, storage reduction, and MACs reduction, with 0.85%, 3.48%, and 2.87%, respectively, against $M9$ with 4.66%, 4.81%, and 6.36%. The difference is relatively larger on parameter and MACs reduction. For instance, there is $941,520,024$ of MACs for $M7$ against $2,087,342,313$ of MACs for $M9$, and a small difference on storage reduction, with 3.48 MB for $M7$ against 4.81 MB for $M9$. On the other hand, there is not much of a difference on mAP, with 0.303 mAP or 67.01% for $M9$ against 0.294 mAP or 64.93% for $M7$. Thus, there are pros and cons for each approach, and so one should look closely to available hardware to make a decision. In general, it seems that $M7$ is a better option, since the large increase on MACs from $M9$ does not seem to compensate for the small increase in mAP.

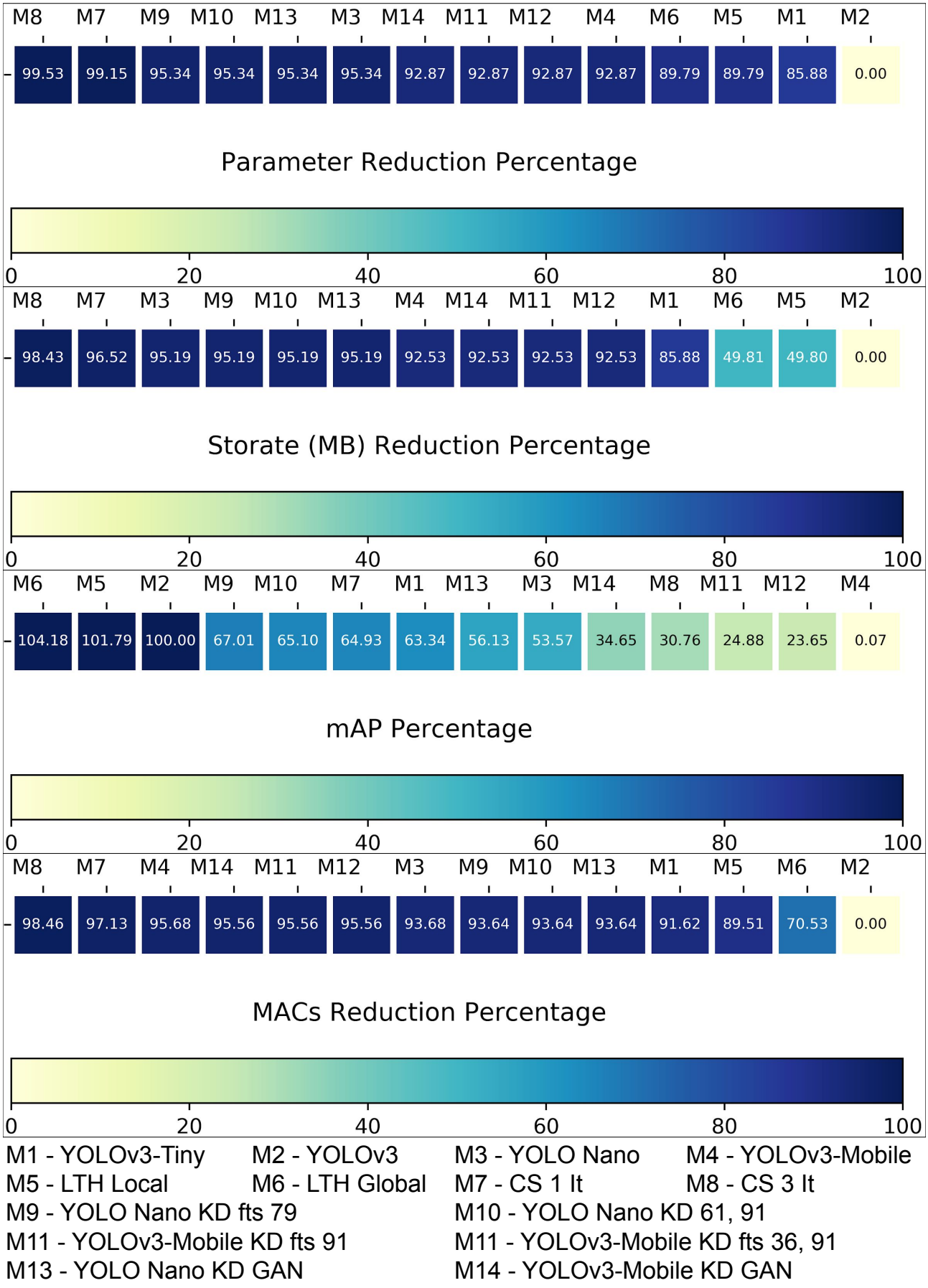


Figure 5.27 – Comparison over mAP, mAP Reduction, and Parameters Reduction on ExDark experiments.

6. CONCLUSIONS

In this dissertation, we detail the problem of Model Compression and its importance for saving computing resources, allowing the use of Deep Learning models to perform tasks such as object detection on limited devices. By exploiting model compression, one can focus on the automation of the decision-making based on visual contexts, *e.g.,* , in surveillance, for object counting, tracking, and control of restricted areas.

We analyze the trade-offs in Model Compression, carefully contrasting model performance, size, and computational effort. We also explain different literature paradigms, as Parameter Pruning, Parameter Sharing, Quantization, Low-Rank Factorization, Efficient Convolutions, and Knowledge Distillation.

Our main contribution in this dissertation was in evaluating techniques from the three most used approaches for model compression to date, namely *Parameter Pruning*, *Neural Architecture Search*, and *Knowledge Distillation*, fixing the evaluated model, dataset, and training scheme, and allowing a fairer assessment of the performance obtained in the object detection task. To the best of our knowledge, this is the first work that compares these three approaches, evaluating all of them in a common experimental setting regarding the main evaluation metrics in the literature. We also are the first to demonstrate the pruning performance from Lottery Tickets Hypothesis and Continuous Sparsification on the object detection task, and also proposing a reconstruction approach that is easy to implement, independent of the Deep Learning framework or hardware resource, and that allows real computing saving for pruning approaches.

Our results show that pruning approaches allow the larger mAP among the compared compression approaches: on PASCAL VOC, LTH can generate models with almost 90% fewer parameters and also outperforming the mAP of the original models in 2.45%, while NAS generates a model with 70.33% of the original model performance. On ExDark, LTH outperforms the original model in 4.18%, while YOLO Nano (from NAS) presents only 53.57% of performance. Looking for the best trade-off between performance and effective computing saving, pruning approaches also provide the best results. CS with an aggressive pruning generates a model on PASCAL VOC with 80.67% of the original model performance but with 98.8% fewer parameters, 96.53% fewer MACs, and 95.07% fewer megabytes to store, while its best competitor, YOLO Nano with KD, has 76.87% of the original model performance, with 95.31% fewer parameters, 93.61% fewer MACs, and 95.17% fewer megabytes. On ExDark, YOLO Nano slightly outperforms CS on predictive performance, with 67.01% and 64.93% respectively, but increasing the CS advantage on parameters and MACs reduction, and also outperforming regarding storage size.

Grad-CAM allows us to perceive that some models either had their output layer neurons getting excited by random regions, or they never got excited, and the use of KD helped models focus on the correct objects. KD succeeded in improving performance in all the proposed scenarios. For instance, the original YOLO Nano performance in PASCAL VOC

increases from 70.33% to 76.87% in the best KD case, and in ExDark it increases from 53.57% to 67.01%. However, it is important to highlight that in our scenario, KD was combined with NAS, which improves its competitiveness in the trade-off. YOLOv3-Tiny has only two YOLO Heads and uses six anchor boxes, while YOLOv3 has three YOLO Heads and nine anchor boxes, thus YOLOv3-Tiny is not useful for any KD technique. For instance, the studies in the literature that propose to compress Faster R-CNN using KD generally use a teacher Faster R-CNN based on a VGG-19 as feature extractor, and a student Faster R-CNN based on a VGG-16, ResNet-101 or ResNet-50 for feature extractor. These examples have an amount of parameters, relative to the teacher, of 96.30%, 31%, and 17.79%, which are students even bigger when compared with our scenario using YOLO Nano.

One final question not addressed in this work is the time to generate those trained models. As described in Chapter 3, NAS is very expensive, generally requiring weeks and using from tens to hundreds of GPUs. We use YOLO Nano, a model created by Wong *et al.* [84] and reported in a short paper that unfortunately lacks this kind of details. KD provides a non-significant increase in the model's training time, and pruning doubles the time in our scenario with one-shot pruning. Both KD and pruning increase the model generation time in a way that can be considered insignificant when compared to NAS.

Evaluating all these questions, our experiments lead us to conclude that, for model compression using YOLOv3, and probably all the YOLO architectures, pruning is the best choice: it provides the larger compression rate and the best mAP, outperforming KD combined with NAS with a consistent and significant gap in terms of parameters and MACs, and a variable gap in terms of mAP and storage size, depending on the approach.

6.1 Limitations

Due to time and hardware constraints, this work has the following limitations:

- LTH is an iterative approach for parameters reduction that can be used; i) in a single-shot, removing all the parameters at one time, as we perform in this work, removing 90% of parameters at the end of the first iteration; or ii) in multi-shots, for example, removing 20% of remaining parameters at the final of each iteration, running many iterations until the total pruning rate approximates 90%. It is well known that the multi-shot approach allows a more aggressive pruning with the best performance. However, to prune 90% of parameters in a multi-shot approach using a pruning rate $\rho = 0.2$, it would be necessary a total of 11 iterations, thus, 11 times slower than normal training. For comparison purposes, a single default training execution with YOLOv3 on PASCAL VOC, using our hardware preset, takes around three days. Thus, five sequential executions with multi-shot takes around 165 days, which is not feasible given the objectives of this research. Thus, we chose the pruning rate $\rho = 90\%$ because it is the highest possible pruning value

that is still safe to maintain the performance of the model, according to the author's experiments on the image classification task.

- The same for LTH can be said to CS. As described in the author's results, CS does not need as many iterations as LTH to perform pruning, since the real pruning is performed only at the end of the iteration loop when the soft-mask is binary. For a more ideal scenario, the three-iteration version of CS should have 300 epochs per iteration, as opposed to 100 iterations.

- Regarding YOLOv3-Mobile, it is our model reconstructed with efficient convolutions, which we manually build using the basic blocks from MobileNetV3, maintaining the macro-architecture from YOLOv3 and trying to use the Squeeze and Excite module in the same logic as on MobileNetV3 macro-architecture. The gap between its performance with default training and KD is the greatest one obtained in this work. However, even with KD, its performance is still poor, being always inferior to YOLOv3-Tiny, which is a manually reduced architecture, with no heuristics or search algorithm. The ideal solution here would be to build a new NAS approach to outperform YOLO Nano. However, as described in Chapter 3, NAS requires weeks with tens or hundreds of GPUs.

- Regarding quantization, we did not evaluate quantization approaches, since it is correlated with hardware optimizations, and we only explore software optimizations due to our restricted environment. As reported by Jeon *et al.* [43], general CPUs and GPUs do not fully support quantization, because those types of hardware support fixed data transfers of, at least, 32 bits. Thus, for the quantized weights, CPUs and GPUs may not access multiple quantized weights without memory bandwidth waste. Consequently, quantization works are mainly performed on FPGA's or other specific hardware that allow a real inference gain with quantization [43, 4, 19, 85, 79]. Moreover, as reported by Ge [22], without a hybrid compression scheme taking into account the software and hardware optimization, quantization and approximation approaches leads to insufficient compression and large accuracy loss.

- Regarding Low-Rank Factorization, we ended up evaluating LRF indirectly, via NAS. For example, the depth-wise 3×3 convolution after or before a 1×1 convolution, as in the YOLO Nano and MobileNets basic blocks, are a kind of LRF from a conventional 3×3 convolution. However, we do not explore a full LRF approach due to time constraints. Furthermore, recent achievements in the literature demotivated us to explore it, due to the smaller compression rates and dropping of performance. For example, Yu *et al.* [88] reduced a GoogLeNet only to 21.43% of its size, but also reducing the ImageNet classification accuracy in 2.4%. Chen *et al.* [7] reduced a MobileNet to 66.67% of parameters and to 36.35% of MACs. Idelbayev and Carreira-Perpiñán [42] propose an approach which reduces the AlexNet parameters to 19.90%. Note that the described studies achieve smaller compression compared to our results, both using pruning or NAS.

Even more, LRF is not related only to model compression, but also to other issues as input compression or anomaly detection [53, 87, 89]. Thus, it is required a more deeper analysis to identify whether model compression alone with Low-Rank Factorization is enough, aiming to achieve comparative results with the other approaches, or if it is necessary to combine it with input factorization to achieve the desired results.

6.2 Future Work

As future work, we believe it would be enriching to use these model compression techniques on other object detection models, like two-step object detectors. Furthermore, we would like to experiment with new pruning and KD approaches, *e.g.,* the work of Tanaka *et al.* [76], that performs pruning that does not require prior training to find out which parameters to remove. Another example is the work of Wu and Gong [86], which performs collaborative KD, that is, both teacher and student are trained from scratch together. We also have some ideas to improve the evaluated approaches. For example, the KD GAN approach can be improved with the Kullback-Leibler Divergence, a metric that, receiving two sets of points, evaluates how the distribution function of each set are similar. This can improve the student feature generation to be more similar to the teacher, as the Kullback-Leibler is already used in GAN studies for image generation. For YOLOv3-Mobile, maybe a restricted NAS can improve its performance, freezing the macro-architecture and using the NAS only to decide the number of channels of each block and the use or not of the Squeeze and Excite modules.

REFERENCES

[1] Agarwal, S.; Terrail, J. O. D.; Jurie, F. "Recent advances in object detection in the age of deep convolutional neural networks". Source: https://arxiv.org/pdf/1809.03193.pdf, 12/24/2020.

[2] A'râbi, M.-A.; Schwarz, V. "General constraints in embedded machine learning and how to overcome them—a survey paper". Source: https://bit.ly/36moJHL, 12/26/2020.

[3] Ba, J.; Caruana, R. "Do deep nets really need to be deep?" In: *27th International Conference on Neural Information Processing Systems*, 2014, pp. 2654–2662.

[4] Bacchus, P.; Stewart, R.; Komendantskaya, E. "Accuracy, training time and hardware efficiency trade-offs for quantized neural networks on fpgas". In: Applied Reconfigurable Computing. Architectures, Tools, and Applications, 2020, pp. 121–135.

[5] Canziani, A.; Paszke, A.; Culurciello, E. "An analysis of deep neural network models for practical applications". Source: https://arxiv.org/abs/1605.07678, 06/18/2020.

[6] Chen, G.; Choi, W.; Yu, X.; Han, T.; Chandraker, M. "Learning efficient object detection models with knowledge distillation". In: 30th International Conference on Neural Information Processing Systems, 2017, pp. 742–751.

[7] Chen, T.; Lin, J.; Lin, T.; Han, S.; Wang, C.; Zhou, D. "Adaptive mixture of low-rank factorizations for compact neural modeling". Source: https://openreview.net/forum?id=r1xFE3Rqt7, 03/01/2021.

[8] Cheng, Y.; Wang, D.; Zhou, P.; Zhang, T. "A survey of model compression and acceleration for deep neural networks". Source: https://arxiv.org/abs/1710.09282, 03/20/2020.

[9] Chintala, S.; ELM. "How to train a gan? tips and tricks to make gans work". Source: https://github.com/soumith/ganhacks, 11/20/2020.

[10] Choudhary, T.; Mishra, V.; Goswami, A.; Sarangapani, J. "A comprehensive survey on model compression and acceleration", *Artificial Intelligence Review*, vol. 53–7, Oct 2020, pp. 5113–5155.

[11] cocodataset.org. "2. metrics". Source: http://cocodataset.org/#detection-eval, 08/04/2020.

[12] Courbariaux, M.; Bengio, Y.; David, J.-P. "Binaryconnect: Training deep neural networks with binary weights during propagations". In: *28th International Conference on Neural Information Processing Systems*, 2015, pp. 3123–3131.

[13] Courbariaux, M.; Hubara, I.; Soudry, D.; El-Yaniv, R.; Bengio, Y. “Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1”. Source: https://arxiv.org/abs/1602.02830, 06/16/2020.

[14] Dave, S.; Baghdadi, R.; Nowatzki, T.; Avancha, S.; Shrivastava, A.; Li, B. “Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights”. Source: https://arxiv.org/pdf/2007.00864.pdf, 02/28/2021.

[15] Denton, E. L.; Zaremba, W.; Bruna, J.; LeCun, Y.; Fergus, R. “Exploiting linear structure within convolutional networks for efficient evaluation”. In: *27th International Conference on Neural Information Processing Systems*, 2014, pp. 1269–1277.

[16] Elsken, T.; Metzen, J. H.; Hutter, F.; et al.. “Neural architecture search: A survey.”, *Journal of Machine Learning Research*, vol. 20–55, Mar 2019, pp. 1–21.

[17] Everingham, M.; Van Gool, L.; Williams, C. K. I.; Winn, J.; Zisserman, A. “The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results”. Source: http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html, 02/03/2020.

[18] Everingham, M.; Van Gool, L.; Williams, C. K. I.; Winn, J.; Zisserman, A. “The pascal visual object classes (voc) challenge”, *International Journal of Computer Vision*, vol. 88–2, Jun 2010, pp. 303–338.

[19] Fan, H.; Wang, G.; Ferianc, M.; Niu, X.; Luk, W. “Static block floating-point quantization for convolutional neural networks on FPGA”. In: International Conference on Field-Programmable Technology, 2019, pp. 28–35.

[20] Frankle, J.; Carbin, M. “The lottery ticket hypothesis: Finding sparse, trainable neural networks”. Source: https://arxiv.org/abs/1803.03635, 07/24/2020.

[21] Frankle, J.; Dziugaite, G. K.; Roy, D. M.; Carbin, M. “Stabilizing the lottery ticket hypothesis”. Source: https://arxiv.org/abs/1903.01611, 07/24/2020.

[22] Ge, S. “Efficient Deep Learning in Network Compression and Acceleration”. 2018, chap. 6.

[23] Girshick, R. “Fast r-cnn”. In: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1440–1448.

[24] Girshick, R.; Donahue, J.; Darrell, T.; Malik, J. “Rich feature hierarchies for accurate object detection and semantic segmentation”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580–587.

[25] Gong, Y.; Liu, L.; Yang, M.; Bourdev, L. “Compressing deep convolutional networks using vector quantization”. Source: https://arxiv.org/abs/1412.6115, 07/10/2020.

[26] Goodfellow, I.; Bengio, Y.; Courville, A. “Deep Learning”. 2016, 800p, http://www.deeplearningbook.org.

[27] Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; Bengio, Y. “Generative adversarial nets”. In: *27th International Conference on Neural Information Processing Systems*, 2014, pp. 2672–2680.

[28] Gou, J.; Yu, B.; Maybank, S. J.; Tao, D. “Knowledge distillation: A survey”. Source: https://arxiv.org/pdf/2006.05525.pdf, 02/28/2021.

[29] Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; Narayanan, P. “Deep learning with limited numerical precision”. In: Proceedings of the 32th International Conference on Machine Learning, 2015, pp. 1737–1746.

[30] Han, S.; Mao, H.; Dally, W. J. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. Source: https://arxiv.org/abs/1510.00149, 04/05/2020.

[31] Hassibi, B.; Stork, D. G.; Wolff, G. J. “Optimal brain surgeon and general network pruning”. In: IEEE International Conference on Neural Networks, 1993, pp. 293–299 vol.1.

[32] Haykin, S. “Neural Networks: A Comprehensive Foundation”. Upper Saddle River, NJ, USA, 1998, 2nd ed., 842p.

[33] He, K.; Zhang, X.; Ren, S.; Sun, J. “Deep residual learning for image recognition”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[34] He, T.; Zhang, Z.; Zhang, H.; Zhang, Z.; Xie, J.; Li, M. “Bag of tricks for image classification with convolutional neural networks”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 558–567.

[35] Hinton, G.; Vinyals, O.; Dean, J. “Distilling the knowledge in a neural network”. Source: https://arxiv.org/abs/1503.02531, 07/14/2020.

[36] Hoefler, T.; Alistarh, D.; Ben-Nun, T.; Dryden, N.; Peste, A. “Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks”. Source: https://arxiv.org/pdf/2102.00554.pdf, 02/28/2021.

[37] Howard, A.; Sandler, M.; Chu, G.; Chen, L.-C.; Chen, B.; Tan, M.; Wang, W.; Zhu, Y.; Pang, R.; Vasudevan, V.; Le, Q. V.; Adam, H. “Searching for mobilenetv3”. In: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 1314–1324.

[38] Howard, A. G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. Source: https://arxiv.org/pdf/1704.04861.pdf, 08/16/2020.

[39] Hui, J. “map (mean average precision) for object detection”. Source: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173, Dez 2019.

[40] Iandola, F.; Moskewicz, M.; Karayev, S.; Girshick, R.; Darrell, T.; Keutzer, K. “Densenet: Implementing efficient convnet descriptor pyramids”. Source: https://arxiv.org/pdf/1404.1869.pdf, 10/16/2020.

[41] Iandola, F. N.; Han, S.; Moskewicz, M. W.; Ashraf, K.; Dally, W. J.; Keutzer, K. “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size”. Source: https://arxiv.org/abs/1602.07360, 06/14/2020.

[42] Idelbayev, Y.; Carreira-Perpinan, M. A. “Low-rank compression of neural nets: Learning the rank of each layer”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2020, pp. 8049–8059.

[43] Jeon, Y.; Park, B.; Kwon, S. J.; Kim, B.; Yun, J.; Lee, D. “Biqgemm: Matrix multiplication with lookup table for binary-coding-based quantized dnns”. Source: https://arxiv.org/pdf/2005.09904.pdf, 17/01/2021.

[44] Jocher, G.; guigarfr; perry0418; Ttayu; Veitch-Michaelis, J.; Bianconi, G.; Baltacı, F.; Suess, D.; WannaSeaU. “ultralytics/yolov3: Video Inference, Transfer Learning Improvements”. Source: https://doi.org/10.5281/zenodo.2624708, Apr. 2019.

[45] Kriesel, D. “A Brief Introduction to Neural Networks”. 2007, 244p.

[46] Krizhevsky, A. “Learning multiple layers of features from tiny images”. Source: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.222.9220&rep=rep1&type=pdf, 08/04/2020.

[47] Krizhevsky, A.; Sutskever, I.; Hinton, G. E. “Imagenet classification with deep convolutional neural networks”. In: *25th International Conference on Neural Information Processing Systems*, 2012, pp. 1097–1105.

[48] LeCun, Y.; Denker, J. S.; Solla, S. A. “Optimal brain damage”. In: 3th International Conference on Neural Information Processing Systems, 1990, pp. 598–605.

[49] Li, H.; Ouyang, W.; Wang, X. “Multi-bias non-linear activation in deep neural networks”. In: Proceedings of The 33rd International Conference on Machine Learning, 2016, pp. 221–229.

[50] Liang, T.; Glossner, J.; Wang, L.; Shi, S. “Pruning and quantization for deep neural network acceleration: A survey”. Source: https://arxiv.org/pdf/2101.09671.pdf, 02/28/2021.

[51] Lin, T.-Y.; Goyal, P.; Girshick, R.; He, K.; Dollar, P. “Focal loss for dense object detection”. In: Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 2980–2988.

[52] Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.-Y.; Berg, A. C. “Ssd: Single shot multibox detector”. In: The European Conference on Computer Vision, 2016, pp. 21–37.

[53] Liu, X.; Li, C.; Dai, C.; Lai, J.; Chao, H.-C. “Nonnegative tensor factorization based on low-rank subspace for facial expression recognition”, *Mobile Networks and Applications*, vol. 26, Jan 2021, pp. 1–12.

[54] Loh, Y. P.; Chan, C. S. “Getting to know low-light images with the exclusively dark dataset”, *Computer Vision and Image Understanding*, vol. 178, Jan 2019, pp. 30–42.

[55] Mitchell, T. M. “Machine Learning”. New York, NY, USA, 1997, 1 ed., 432p.

[56] Moody, J. E.; Rögnvaldsson, T. S. “Smoothing regularizers for projective basis function networks”. In: 10th International Conference on Neural Information Processing Systems, 1997, pp. 585–591.

[57] Rastegari, M.; Ordonez, V.; Redmon, J.; Farhadi, A. “Xnor-net: Imagenet classification using binary convolutional neural networks”. In: The European Conference on Computer Vision, 2016, pp. 525–542.

[58] Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. “You only look once: Unified, real-time object detection”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 779–788.

[59] Redmon, J.; Farhadi, A. “Yolo9000: Better, faster, stronger”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 779–788.

[60] Redmon, J.; Farhadi, A. “Yolov3: An incremental improvement”. Source: https://arxiv.org/abs/1804.02767, 05/11/2020.

[61] Ren, S.; He, K.; Girshick, R.; Sun, J. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *28th International Conference on Neural Information Processing Systems*, 2015, pp. 91–99.

[62] Rezatofighi, H.; Tsoi, N.; Gwak, J.; Sadeghian, A.; Reid, I.; Savarese, S. “Generalized intersection over union: A metric and a loss for bounding box regression”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 658–666.

[63] Rigamonti, R.; Sironi, A.; Lepetit, V.; Fua, P. “Learning separable filters”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2013, pp. 2754–2761.

[64] Romero, A.; Ballas, N.; Kahou, S. E.; Chassang, A.; Gatta, C.; Bengio, Y. “Fitnets: Hints for thin deep nets”. Source: https://arxiv.org/abs/1412.6550, 07/14/2020.

[65] Russel, S.; Norvig, P. “Artificial Intelligence. A modern approach”. 2002, 1152p.

[66] Salimans, T.; Goodfellow, I.; Zaremba, W.; Cheung, V.; Radford, A.; Chen, X.; Chen, X. “Improved techniques for training gans”. In: 29th International Conference on Neural Information Processing Systems, 2016, pp. 2234–2242.

[67] Salvi, A. A.; Barros, R. C. “An experimental analysis of model compression techniques for object detection”. In: Symposium on Knowledge Discovery, Mining and Learning, 2020, pp. 49–56.

[68] Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.-C. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 4510–4520.

[69] Savarese, P.; Silva, H.; Maire, M. “Winning the lottery with continuous sparsification”. Source: https://arxiv.org/abs/1912.04427, 06/14/2020.

[70] Selvaraju, R. R.; Cogswell, M.; Das, A.; Vedantam, R.; Parikh, D.; Batra, D. “Grad-cam: Visual explanations from deep networks via gradient-based localization”. In: Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 618–626.

[71] Shang, W.; Sohn, K.; Almeida, D.; Lee, H. “Understanding and improving convolutional neural networks via concatenated rectified linear units”. In: Proceedings of the 33th International Conference on Machine Learning, 2016, pp. 2217–2225.

[72] Simonyan, K.; Zisserman, A. “Very deep convolutional networks for large-scale image recognition”. Source: https://arxiv.org/pdf/1409.1556.pdf%20http://arxiv.org/abs/1409.1556.pdf, 08/04/2020.

[73] Szegedy, C.; Ioffe, S.; Vanhoucke, V.; Alemi, A. A. “Inception-v4, inception-resnet and the impact of residual connections on learning”. In: AAAI Conference On Artificial Intelligence, 2017, pp. 4278–4284.

[74] Tan, M.; Chen, B.; Pang, R.; Vasudevan, V.; Sandler, M.; Howard, A.; Le, Q. V. “Mnasnet: Platform-aware neural architecture search for mobile”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 2820–2828.

[75] Tan, M.; Le, Q. V. “Efficientnet: Rethinking model scaling for convolutional neural networks”. Source: https://arxiv.org/pdf/1905.11946.pdf, 10/16/2020.

[76] Tanaka, H.; Kunin, D.; Yamins, D. L. K.; Ganguli, S. “Pruning neural networks without any data by iteratively conserving synaptic flow”. Source: https://arxiv.org/pdf/2006.05467.pdf, 01/14/2021.

[77] Tung, F.; Mori, G. “Clip-q: Deep network compression learning by in-parallel pruning-quantization”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 7873–7882.

[78] Vanhoucke, V.; Senior, A.; Mao, M. Z. “Improving the speed of neural networks on cpus”. Source: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/37631.pdf, 10/08/2020.

[79] Wang, E.; Davis, J. J.; Cheung, P. Y. K.; Constantinides, G. A. “Lutnet: Rethinking inference in fpga soft logic”. In: International Symposium on Field-Programmable Custom Computing Machines, 2019, pp. 26–34.

[80] Wang, E.; Davis, J. J.; Zhao, R.; Ng, H.-C.; Niu, X.; Luk, W.; Cheung, P. Y. K.; Constantinides, G. A. “Deep neural network approximation for custom hardware: Where we’ve been, where we’re going”, *ACM Comput. Surv.*, vol. 52–2, May 2019, pp. 1–39.

[81] Wang, W.; Hong, W.; Wang, F.; Yu, J. “Gan-knowledge distillation for one-stage object detection”, *IEEE Access*, vol. 8, Mar 2020, pp. 60719–60727.

[82] Wang, X.; Zhang, R.; Sun, Y.; Qi, J. “Kdgan: Knowledge distillation with generative adversarial networks”. In: *31th International Conference on Neural Information Processing Systems*, 2018, pp. 775–786.

[83] Weigend, A. S.; Rumelhart, D. E.; Huberman, B. A. “Generalization by weight-elimination with application to forecasting”. In: 3th International Conference on Neural Information Processing Systems, 1990, pp. 875–882.

[84] Wong, A.; Famuori, M.; Shafiee, M. J.; Li, F.; Chwyl, B.; Chung, J. “Yolo nano: a highly compact you only look once convolutional neural network for object detection”. Source: https://arxiv.org/abs/1910.01271, 07/20/2020.

[85] Wu, C.; Wang, M.; Li, X.; Lu, J.; Wang, K.; He, L. “Phoenix: A low-precision floating-point quantization oriented architecture for convolutional neural networks”. https://arxivorg/pdf/200302628pdf.

[86] Wu, G.; Gong, S. “Peer collaborative learning for online knowledge distillation”. Source: https://arxiv.org/pdf/2006.04147.pdf, 01/14/2021.

[87] Xue, J.; Zhao, Y. Q.; Bu, Y.; Liao, W.; Chan, J. C. W.; Philips, W. “Spatial-spectral structured sparse low-rank representation for hyperspectral image super-resolution”, *IEEE Transactions on Image Processing*, vol. 30, Feb 2021, pp. 3084–3097.

[88] Yu, X.; Liu, T.; Wang, X.; Tao, D. “On compressing deep models by low rank and sparse decomposition”. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 7370–7379.

[89] Zhang, X.; Ma, X.; Huyan, N.; Gu, J.; Tang, X.; Jiao, L. "Spectral-difference low-rank representation learning for hyperspectral anomaly detection", *IEEE Transactions on Geoscience and Remote Sensing*, vol. 1, Jan 2021, pp. 1–14.

[90] Zoph, B.; Le, Q. V. "Neural architecture search with reinforcement learning". Source: https://arxiv.org/pdf/1611.01578.pdf, 12/23/2020.

[91] Zoph, B.; Vasudevan, V.; Shlens, J.; Le, Q. V. "Learning transferable architectures for scalable image recognition". In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 8697–8710.

[92] Zou, Z.; Shi, Z.; Guo, Y.; Ye, J. "Object detection in 20 years: A survey". Source: https://arxiv.org/pdf/1905.05055.pdf, 12/24/2020.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br