António Filipe Correia Semitela

COMPUTER VISION ON THE EDGE

UNIVERSIDADE D COIMBRA
1 2 9 0

António Filipe Correia Semitela

COMPUTER VISION ON THE EDGE

Internship Report in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Professor Joel Arrais and Eng. João Garcia (Ubiwhere) and presented to
Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

Faculty of Science and Technology

Department of Informatics Engineering

Computer Vision on the Edge

António Filipe Correia Semitela

Internship report in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems advised by Prof. Joel Arrais and Eng. João Garcia and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

1 2 9 0

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Abstract

Given the growth over the years in CPU and GPU processing capabilities and the advancements made in Deep Learning, it is now possible to develop more complex Computer Vision models and architectures. As a result, Computer Vision algorithms are present and running in multiple devices and objects of our daily lives, from self-driving cars to smartphone facial recognition.

The goal of the present work is to implement an efficient model that can identify and classify cars with good performance through the use of Convolutional Neural Networks and Deep Learning techniques. The solution should run on an Edge Hardware Device to be placed in Ubiwhere's SmartLamppost. This lamppost can contain different modules, such as cameras and edge nodes capable of processing data. It will be useful in counting and classifying vehicles passing through a certain area, using the video feed from the cameras.

To achieve this goal, different approaches were followed to develop the final model, from creating and optimizing a model to using pre-trained models. The final choice was a One Stage Detectors structure, optimized for speed on edge devices. As a result, the final model obtained good results that reached 39.36 % of mAP in the COCO dataset and good inference values on the edge device.

This document also presents a study of state of the art in detecting and classifying objects in general, and specifically about the different object detection models more suited for real-time detection and the concepts that were essential for the realization of this work.

Keywords

Computer Vision, Deep Neural Networks, Convolution Neural Networks, Object Detection

This page is intentionally left blank.

Resumo

Dado o crescimento ao longo dos anos nas capacidades de processamento de CPU e GPU e os avanços feitos na área de Deep Learning, é possível desenvolver modelos e arquiteturas de visão computacional mais complexos. Algoritmos de visão computacional estão presentes e em execução em vários dispositivos e dispositivos do nosso dia-à-dia, desde carros autónomos até o reconhecimento facial em smartphones.

O objetivo deste trabalho passa por implementar uma solução eficiente que, através do uso de Redes Neurais Convolucionais e técnicas de Deep Learning, seja capaz de identificar e classificar veículos com bom desempenho. A solução deve ser capaz de correr num Edge Hardware Device a ser colocado no SmartLamppost da Ubiwhere, um poste de luz que pode conter diferentes módulos, como câmeras e edge nodes capazes de processar dados. Será útil em tarefas como contagem e classificação de veículos que passam por uma determinada área, usando o feed de vídeo das câmeras.

Para atingir este objetivo, diferentes abordagens foram seguidas para desenvolver o modelo final. Desde a criação e optimização de um modelo, para a utilização de modelos pré treinados. A escolha final foi de uma estrutura de One Stage Detectors, otimizados para velocidade em dispositivos móveis. Como resultado, o modelo final obteve bons resultados que alcançaram 39.36 % de mAP no dataset COCO e bons valores de inferência no dispositivo de edge.

Este documento também apresenta um estudo do estado da arte na detecção e classificação de objetos em geral e, especificamente, sobre os diferentes modelos de detecção de objetos mais adequados para a detecção em tempo real.

Palavras-Chave

Visão Computacional, Redes Neuronais Profundas, Redes Neuronais de Convolução, Deteção de Objectos

This page is intentionally left blank.

Acknowledgements

I would like to thank my advisors João Garcia and professor Joel Arrais for their time and valuable contributions through out the year, and for speaking up to me when needed. Thank you.

A big special thank you to my parents for their unconditional support and for making this possible, for always being there in the good and bad moments.

And to my friends, for always keeping my spirit up and brightening up the days. Without you, it would not be the same.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

AP Average Precision. xvii, 34, 48

CNN Convolutional Neural Network. 3, 8, 12, 14, 15, 17–23, 25, 27–29, 40, 50, 53

COCO Common Objects in Context. 27, 32, 33, 37, 39, 40, 45, 47, 53

ConvLayer Convolutional Layer. 12–14, 18, 21

CPU Central Processing Unit. 35, 36

CV Computer Vision. 2, 5, 6, 14, 15, 19, 20, 25

DL Deep Learning. 1, 19, 20, 23–25, 27, 53

DNN Deep Neural Network. 1, 2, 5, 8, 11, 12, 16, 19, 20, 24, 25, 53

Faster R-CNN Faster Region-based Convolutional Network. 21, 22

FC Fully Connected. 7, 14, 18, 21, 40, 41

FFNN Feed-Forward Neural Network. 7, 12

FPS Frames per Second. 1, 3, 22–24, 34–36, 51

GPU Graphics Processing Unit. 1, 22, 23, 35, 36

HOG Histogram of Oriented Gradients. 20

ILSVRC ImageNet Large Scale Visual Recognition Competition. xv, 20

IOU Intersection Over Union. xvii, 31, 33, 34, 47, 48, 51

IT Inference Time. 1, 3, 22, 24, 34, 35, 50

mAP Mean Average Precision. xv, xvii, 34, 46–48, 51

ML Machine Learning. 5, 19

MSE Mean Squared Error. 9, 41

R-CNN Regions with CNN features. 21, 22

ReLU Rectified Linear Unit. 8, 9, 12, 40, 41

ResNet Residual Neural Network. 20, 21, 25

RGB Red, Green and Blue. 13, 18, 40

RPN Region Proposal Network. 21

SDK Software Development Kit. 43

SGD Stochastic Gradient Descent. 11, 12

SIFT Scale-Invariant Feature Transform. 19, 20

SSD Single Shot MultiBox Detector. 22, 28, 39, 42, 51

SVM Support Vector Machine. 21

YOLO You Only Look Once. 21–23, 28, 30

YOLOV4 You Only Look Once V4. x, xv, 29, 30, 37, 39, 41–43, 45–51, 53

ZCA Zero Component Analysis. 15

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

We are witnessing a constant technological evolution where Artificial Intelligence and Machine Learning have assumed a leading role. As a result, more and more intelligent systems are developed to perform tasks associated with human behavior. One of those is Computer Vision, a computer science field that focuses on enabling computers to see, identify, and process images, producing the desired output.

There are many applications of Computer Vision nowadays: in our roads, cameras detecting and identifying cars, in the field of Medical Informatics, where image classification algorithms are used to analyze medical images to detect diseases, in our smartphones with facial recognition as a safety measure, for pose estimation that models the human body, in self-driving cars where multiple object detection is needed, in Optical Character Recognition, to recognize digits and signatures in documents, and many others.

Three major factors have been essential in the big transformation of the paradigm in Computer Vision, the success of Deep Neural Networks (DNNs) in Computer Vision tasks, the creation of larger and better-labeled image datasets for training, and the availability of more powerful Graphics Processing Units (GPUs) on the market. These robust networks can extract features from images, which, combined with large datasets of images, can present better results than previous machine learning-based approaches. The large datasets provide the variability often seen in real life that images come in different shapes and colors, therefore amplifying the network's capabilities. Training the models on such large datasets requires extensive mathematical computations that use high quantities of memory, such as matrix multiplications and convolutions. Therefore the use of GPUs is essential for Deep Learning (DL) since they allow for multiple parallel computations and are bandwidth optimized for large models. This led to increasing investment from companies in the last years to improve the quality of this GPUs while making it affordable for research.

When trying to do Object Detection and Image Classification in real-time, the two significant problems are performance, that is, processing images at a reasonable rate of Frames per Second (FPS) and Inference Time (IT), and how accurate the model is at identifying the different objects. In this work, a solution will be implemented that will use Convolutional Neural Networks and Deep Learning techniques to overcome these challenges efficiently. This solution will be running in an edge hardware device that will be added to one of the edge nodes of the Smart Lamppost, one of the leading products at Ubiwhere, a software company based in Aveiro, Portugal.

1.1 Context and Motivation

Ubiwhere is a company based in Aveiro that focuses on developing intelligent systems for Smart Cities, Telecommunications, and the Internet of the Future. Ubiwhere started designing and developing several solutions for the demanding challenges that smart cities face, such as environmental monitoring, energy efficiency, mobility, sustainability, among others. All this culminated in the Urban Platform[14], created from Ubiwhere's vision of providing cities with a holistic view of their smart urban environment.

The Smart Lamppost is one of Ubiwhere's flagship products, integrated on the Urban Platform[14], which consists of a modular lighting pole, where it is possible to install different modules depending on the needs of the city. Examples of such modules can be 5G cells to support this network, environmental stations to measure air quality, EV charging stations for charging electric vehicles, and Edge nodes that allow information processing at the end of the network.

The Plug-and-play Edge Computing solution off the Smart Lamppost lets different service providers deploy compute, storage, and network resources, in a distributed way, which can be explored for many sorts of applications. One of those applications is the one described in this project, the detection and classification of vehicles that pass through certain zones. The solution developed for this task can serve as a base of many applications that require Computer Vision (CV) in the context of Smart Cities, such as Smart Parking, a system responsible for managing parking lots for vehicles, and Smart Traffic, to manage traffic.

The reason to deploy the detection model on the edge nodes on the Smart Lamppost is that it allows to process and analyze the data collected locally, which will lead to better performance since it will not be needed to transfer the data in real-time to the cloud, which can be a challenge to do cost-effectively. In addition, this improves data processing speed due to the less latency and bandwidth requirements as opposed to processing the large quantities of data all in the cloud.

1.2 Objectives

The main goal of this internship project is to implement an object detection model that will be applying real-time vehicle defections on edge hardware. In other words, the goal is to train a DNN capable of accomplishing the task mentioned before with a high confidence score, low consumption of CPU and GPU resources, and high performance. The hardware will then be placed on edge nodes on the Smart Lamppost, capable of detection and classification of vehicles.

Another objective for this work will be fine-tuning and optimizing the network responsible for the classification and detection, making it as lightweight as possible for deployment. Since the memory requirements and the processing speed on these complex models running on the edge can be a problem due to the edge hardware's relatively small memory capacity compared to a regular server on the cloud.

1.3 Challenges

To develop a detection model that can accomplish the objectives defined in section 1.2, some challenges must be taken into consideration:

- Convolutional Neural Network (CNN) architectures are complex, and their parameters are sometimes harder to fine-tune to get low error rates.
- Preparing, extracting, and selecting information and data from large datasets can be time-consuming.
- The computation of object detection and image classification in real-time can be very resource-heavy, so the solution needs to be fast and simple, so it doesn't waste resources. The efficiency of the final solution is essential, and this means it to be running at a good rate of FPS, low IT and good accuracy at test.
- Given that the computational capabilities in edge hardware are lower than in a regular server, the final model must not bottleneck the device and optimizing it can be difficult.
- The proper hardware configuration can be time-consuming.

1.4 Document Structure

Chapter 2 contains the research made about the state of the art, first explaining the basic concepts and techniques about this work and concluded with a study on related work and some conclusions taken.

Chapter 3 presents the approaches followed to develop the model implemented in this project, as well as the different datasets studied that will be used and the metrics the model's evaluation.

Chapter 4 represents the development made for this work, as well as presents the models used.

Chapter 5 are the results of the development, as well as the results on the Jetson Nano, with an analysis of the obtained result.

Finally, Chapter 6 addresses the overall conclusions and retrospectives about this internship. It also presents final remarks and the future work that can be done.

This page is intentionally left blank.

Chapter 2

State of the Art

This chapter presents the different concepts that serve as the background for this project and the research made about the related work. The study focuses on solutions for vehicle detection, image classification, and object detection. A brief explanation about the frameworks that can be used to build these types of models is presented and the types of hardware that can be used to run these models efficiently.

2.1 Concepts

This section explains all the concepts needed for the understanding of the discussion on Computer Vision (CV) and Deep Neural Networks (DNNs) mentioned throughout the work.

2.1.1 Computer Vision

CV is an interdisciplinary field of study, based on AI and Machine Learning (ML), that can be defined as the task of a machine learning the qualitative representation of visual elements in their raw form to quantify them. CV is a field of computer science that focuses on enabling computers to see, identify and process images, producing the desired output. In other words, the computer sees visual objects. It automatically extracts and processes information of a scene based on an image or sequences of images, performing a specific task [43].

CV can be decomposed into eight different tasks. Depending on the type of visual task we want to perform, DNNs can be trained differently with different layers, and different sets of algorithms [28]. These tasks are image classification, segmentation, and localization, tracking, action recognition, perception, generative models, clustering, and decision-making, as shown in figure 2.1.

Of these eight different tasks, we'll be focusing on image classification and segmentation, and localization, the latter including object detection as a subcategory.

2.1.2 Image Classification

Image Classification is a fundamental task in CV that can be defined as the task of classifying an image to a specific label, given a discrete set of labels. Since the computer sees

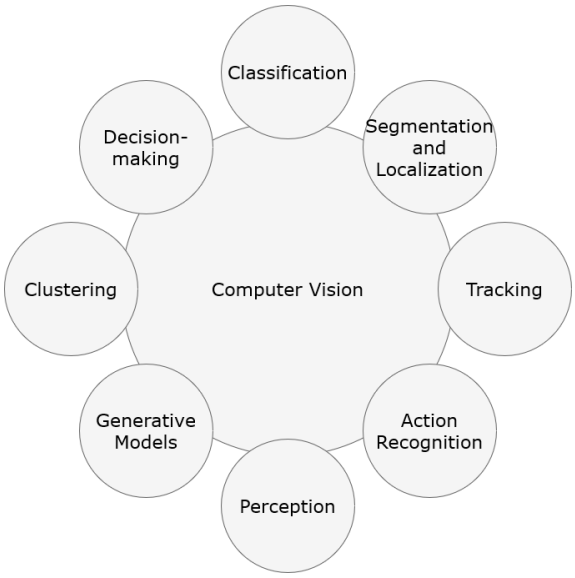

Figure 2.1: Computer Vision Tasks. [35].

images as large grids of pixels and their corresponding values, the task of classification can be challenging. For the same object, different variations can occur in small areas of the grid corresponding to different angles, lighting, different positions, occlusion, and intra-class variation (the same object with different versions) [43]. The models that will classify images must be trained to learn the features of several images to classify unseen visual data. The image classification task can be both supervised if based on a labeled dataset of images or unsupervised if trying to learn new features from unseen data. For this project, we'll focus on a supervised learning classification, where we'll train our models using a large labeled dataset.

A model trained on the Cifar10 Dataset[1] is a good example of supervised Image Classification, where there are ten different discrete classes like dogs, cats, and cars, and we try to classify each image into one of these classes. For an image of a cat, if the trained model outputs a higher probability for the class "cat" than the others, we say the model is able to identify a cat on the image. The precision of models is determined by how accurate the predictions are for all the classes and different images.

2.1.3 Object Detection

Object Detection is an important computer vision task that deals with detecting instances of semantic objects of a particular class (such as humans, cars, or animals) in digital images and videos [72]. It involves both classification and localization tasks and analyzes more realistic cases in which multiple objects may exist in an image.

The core goal of object detection is developing models that can both recognize the objects we want to identify and where they are in an image. In other words, it aims to answer the following question: "What objects are where?". Object detection is the base for many other CV tasks, such as segmentation, image captioning, and object tracking. It has been widely present in many real-world applications, such as autonomous driving, robot vision, video surveillance, vehicle detection, etc.

In an image with multiple objects, Object Detection focuses on identifying where the different objects are in the image; that is, it distinguishes what the background is and

what is an object. It separates the object from the image's background by predicting a bounding box to surround the different objects and identifies the object's class using Image Classification.

2.1.4 Neural Networks

Neural Networks, inspired by the biological brain, is composed of layers, each containing neurons connected to each other. Each connection can transmit a signal, like the synapses in the brain, to other neurons. Each neuron produces a sequence of real-valued activations. Input neurons get activated through the input values, and other neurons get activated through weighted connections from previously active neurons [60].

The perceptron is the simplest form of Feed-Forward Neural Network (FFNN), which can be visualized in figure 2.2. The goal of a FFNN is to approximate some function f. For example, for a classifier, $y = f(x)$ maps an input x to a category y. A FFNN defines a mapping $y = f(x; w)$ and learns the value of the parameters w that result in the best function approximation [28]. In a FFNN, the information moves in only one direction, forward from the input nodes, through the output nodes. There are no cycles or loops in this network, as opposed to other more complex networks that contain feedback connections, such as Recurrent Neural Networks.

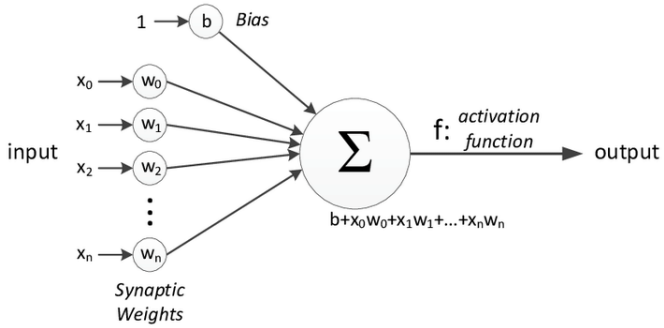

Figure 2.2: Perceptron. From: [49]

The perceptron is composed of several components. The **Inputs**, characterized as a vector x of size n, where each position is a value for the feature n-th. A **Bias**, a special input, in most cases equal to 1, shifts the decision boundary away from the origin. The **Weights**, characterized as vector w, containing the values of the weights of the connections from the inputs to the output neuron (the perceptron). The values are initialized and multiplied by the input values. These values are then updated for each training error. Finally, the output neuron, responsible for calculating the weighted sum for the weights and inputs and producing an output using an **activation function**, for example, for binary classification, that is 0 if the arguments are negative and 1 otherwise.

The output of the perceptron can be mathematically described as follows:

$$\mathrm{f(x)} = \begin{cases} 1, & \text{if } w * x + b > 0 \\ 0, & \text{otherwise} \end{cases}$$

A simple FFNN, also called Multi-Layer Perceptron or Deep Forward Networks, are composed of multiple layers that allow arranging neurons into Fully Connected (FC) layers. There are three main types of layers: the input layer, the hidden layers, and the output layer. The abstraction of the layer to represent a set of neurons has the nice property that

it allows us to use efficient vectorized code, for example, matrix multiplication of grids of pixels. An n-th Layer Neural Network has n-1 hidden layers. Each neuron in the layers is connected to the other neurons, and it functions as the perceptron, where it calculates the weighted sum of its inputs and produces an output according to the activation function in that layer. The figure 2.3 shows a 2-hidden layer Neural Network.

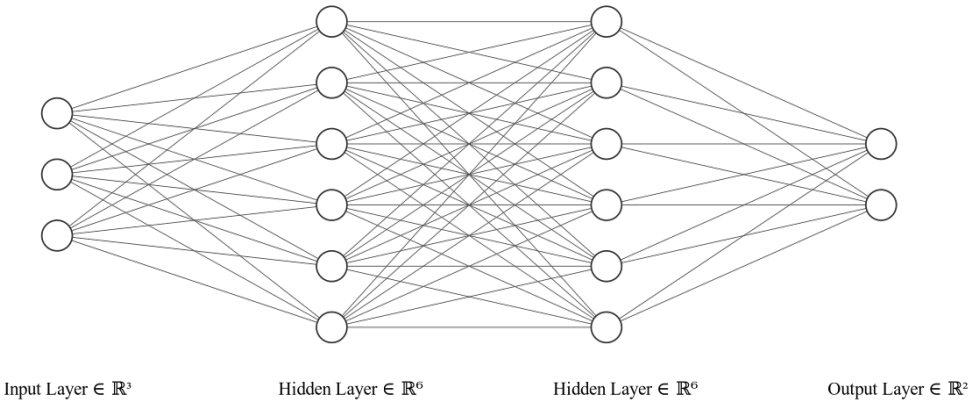

Figure 2.3: 2-Hidden Layer Neural Network

Activation Functions

The activation functions, inspired by biology, decide if the neuron is active or not, taking as parameters the weighted sum and bias, as described in the expression:

$$G(\sum_i W_i * x_i + b)$$

where G can be any activation function, W_i the weights, x_i the inputs and b the *bias*. Activation functions are useful because they add non-linearities into neural networks, allowing the feed-forward neural networks to learn powerful operations.

There is a set of different non-linear activation functions useful for DNNs, such as the Sigmoid, Tahn, Maxout, and others, but in this work, the focus will be on the most used for Convolutional Neural Networks (CNNs), the Rectified Linear Unit (ReLU) [40]. Deep networks with ReLUs are more easily optimized than networks with Sigmoid or Tanh functions because gradients do not saturate when the input to the ReLU function is positive. Thanks to its simplicity and effectiveness, ReLU has become the default activation function used across the deep learning community [55]. The ReLU activation function transforms the negative values to zero and maintains the positive values. The mathematical expression for ReLU is:

$$f(x) = max(0, x)$$

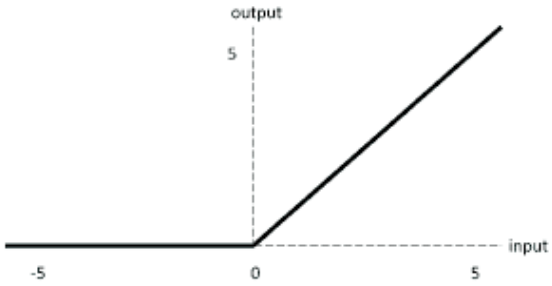

Figure 2.4: ReLU function.

Other variants, like the leaky-ReLU, are similar and try to overcome the problem of a neuron that can become trapped in the zero region, and backpropagation will never change its weights, and the gradient gets stuck.

The activation function in the last output layer is different according to the task at hand. For example, for Multi-Class Classification, the **Softmax** activation function is a good option to normalize the output to a probabilistic distribution for the different class candidates. It can be defined as:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

It converts the vector of numbers z into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each element z_i in the vector. Then, it applies the standard exponential function in the nominator and divides by the normalization term, each j representing a different class from 1 to K total classes.

For Object Detection, when trying to predict the bounding boxes surrounding the objects, there are several options such as linear regression functions such as the Mean Squared Error (MSE), or two-class Softmax function [58], to identify if it is an object or not. MSE can be represented as:

$$L(y, \widehat{y}) = \frac{1}{N} \sum_{i=0}^{N} (y - \widehat{y}_i)^2$$

where y represents the true values and $\widehat{y}$ the predicted values on a vector of n samples, producing the error between both values.

Loss Functions

A loss function tells how good the classifier is after each moment of training. It evaluates how good the algorithm is at modeling the dataset. It helps understand if the predictions made are good or not, i.e., if the error between our network's output and the target value is high or not. This way, it permits our network to improve by minimizing the error, with the help of an optimization function 2.1.4. There are three types of Loss functions:

- **Regression Loss Functions**: for when the predictions are on continuous domains, that is, for when trying to predict a real value.

- **Binary Classification Loss Functions**: To when the output assigns one of two labels, typically represented as 1 and 0, or probabilities of belonging to one label, between 0 and 1, using a threshold. If the value is bigger than the threshold, then it belongs to class 1.
- **Multi-Class Classification Loss Functions**: when there are more than two class targets.

The loss is calculated for each training sample, and the loss over the dataset is the sum of the losses over all the training samples. For example, in image classification problems where there are multiple classes, given a dataset with values $(x_i, y_i), i = 1..N$, where x_i is an image, y_i is an integer label, N the number of samples, the loss for all the dataset, L, can be generalized as follows:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i(s_i, y_i)$$

where L_i is the loss function for each of the training sample x_i and respective target y_i, and $s_i = f(x_i, W)$ is the score for each input x_i and weights W, according to a score function f, for example $f = xW$ (linear).

An example of a loss function for Multi-Class Classification can be the Categorical Cross-Entropy Loss. In the same context as before, assuming that the output layer has a softmax activation, meaning that the output scores s will be distributed probabilities, and we have a fixed number of classes C, we can describe the Cross-Entropy Loss L_i as follows:

$$L_i = -log(\frac{e^{s_{y_i}}}{\sum_j^C e^{s_j}})$$

For a training sample x_i, if the network outputs a high score or probability for the true class of the image, the loss will take a low value for this sample.

For the most common Object Detection models, two essential loss functions are used. First, the Focal Loss, presented in [46], for the classification task, has the goal to penalize an error more strongly if the probability of the class is high, in other words, solve the class imbalance problem, where one class is more represented than others. Second, the IoU loss is used [20] when predicting the bounding boxes, and this loss is calculated using the formula for the Intersection Over Union explained later in section 3.3.

Training the network

In order to train a Deep Neural Network, its common to use the Backpropagation algorithm to compute the gradients of all parameters according to a loss function by recursively applying the chain rule of calculus. The gradients are backpropagated to update the weights. The network learns by updating the weights to minimize the loss (or error). Backpropagation typically consists of two phases:

- **A forward phase**, where the input is passed completely through the network to represent the input image with the current parameters (weights and bias) in each layer, then computing the loss.
- **A backward phase**, that allows the information from the loss to then flow backward through the network to compute the gradient.

Assuming an Loss function L, and z the output of a function f, such as $z = f(x, y)$ for the parameters x, y, the gradient for the Loss in respect to the output z is calculated as:

$$\bigtriangledown L = \frac{\partial L}{\partial z}$$

Applying the chain rule of calculus, the gradient for the parameters x and y can be calculated as follows:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x}$$
$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial y}$$

Back-propagation refers only to the method for computing the gradient, while another algorithm, such as Stochastic Gradient Descent 2.1.4, is used to perform learning using this gradient[27].

One common problem when training a neural network is the **vanishing gradient problem**, where the gradient decreases and becomes too small, effectively preventing the weights from changing its value, and therefore, makes it hard to be back-propagated trough the network, especially for very deep networks.

Optimizers

TThe optimizers build a connection between the loss function and model parameters by updating these parameters concerning the output of the loss function. The goal of the optimizers is to improve the accuracy of the model by regulating the way the weights are updated, when they are updated and how often they are updated. Better optimization algorithms help reduce the training loss.

Optimizers are typically defined by their update rule, which is controlled by the hyperparameters that determine its behavior [21]. Hyperparameters are the global variables of the model that are used to control the learning process. Contrary to the other parameters that are learned via training, these variables have a fixed value at the beginning of training and can be optimized over training through trial and error, as seen in the section. 2.2.2.

Stochastic Gradient Descent (SGD) is an optimizer commonly used in DNNs, since it is a good generalizer for many problems. SGD is an extension of the Gradient Descent algorithm. Both in Gradient Descent and SGD, the parameters are updated according to a certain step towards a local minima, called **Learning Rate**, that is multiplied by the gradient of the parameters. **Learning Rate** is a hyperparameter of the network that controls the rate or speed at which the model learns or, in other words, controls the update of the weights.

The difference between Gradient Descent and SGD, is that SGD uses a sample of the dataset selected randomly to update the parameters instead of the whole dataset. This results in a faster convergence to the local minima of the loss function. Therefore, we want to find the optimal w that minimizes f according to a sample x_i and y_i selected from the data each iteration i, and we compute for each iteration:

$$w = w - \alpha \bigtriangledown f_i(w)$$

where α is the learning rate, a positive scalar determining the size of the step. We can

choose α in several different ways. A popular approach is to set α to a small constant, in the range between $[1e^{-2}, 1e^{-5}]$.

One of the problems with SGD is when the function has local minima or saddle points, and in these points, the gradient is zero or almost zero, and the gradient gets stuck. To overcome this, other optimizers can be used that use techniques such as momentum and learning rate decay. One popular choice is **Adam** [38].

Adam is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks. The algorithms leverage the power of adaptive learning rates methods to find individual learning rates for each parameter. Adam uses estimations of the first and second moments of the gradient to adapt the learning rate for each weight of the neural network [38]. To estimate the first and second moments, Adam utilizes exponentially moving averages, computed on the gradient evaluated on a current mini-batch:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Where m and v are moving averages for the first and second moment respectively, g is gradient on current mini-batch t, and β is a newly introduced hyper-parameter of the algorithm. Now, it calculates the step that is usually referred to as bias correction to correct the estimator's bias. The bias-corrected estimators for the first and second moments are as follow:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally, it uses those moving averages to scale learning rate individually for each parameter, so in order to perform weight update, the update rule for Adam is the following:

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where w are the weights of the model, α is the learning rate (or step size).

2.1.5 Convolutional Neural Networks

CNNs are a type of DNNs that use convolution in place of general matrix multiplication in at least one of their layers [27]. These networks maintain the spatial information of the input and apply the filters on spatially-neighboring pixels across all axis. This is what allows CNNs to identify spatial patterns like edges, shading changes, shapes, objects, and others. In a regular FFNN the image is flattened into a vector, and then its applied an FC linear combination followed by non-linear activation. CNN are composed of a sequence of Convolutional Layers (ConvLayers) interspersed with activation functions and the pooling layer, followed by a fully-connected layer for classification [28] [43]. Each layer has its function in the overall network:

- **Convolutional Layer**: Responsible for learning the features from the images used to train the network coupled with an activation function like ReLU to add non-linearity.

- **Pooling Layer**: Reduces the dimensionality of the activation maps produced by the Convolutional Layers to save memory. Can also be a form of regularization 2.2.3.
- **Fully-Connected Layers**: Are responsible for converting the features extracted by the Convolutional Layers into neurons to be fed forward to the fully-connected layers. The last Fully-Connected layer is to adjust the weights and predict an output.

ConvLayers convolve filters (or kernels) of weights with the image input or previous layers output. Typically slide over the input spatially and apply N different F x F filters. This operation is called Kernel Convolution. The layer stores the weighted sum of dot products values between the filters and the chunk of the image, called activation maps (or feature maps). Some hyperparameters to have in consideration:

- **Number of filters**: $\boldsymbol{N}$, typically powers of two
- **Dimension of the filters**: $\boldsymbol{F}$
- **Stride**: $\boldsymbol{S}$, step at which the filters slide through the image
- **Amount of Zero Padding**: $\boldsymbol{P}$, adds zeros to the border in order to preserve spatial dimensions of the activation maps.

Considering a regular image of dimensions Weight x Height x Depth (for example, 3 Red, Green and Blue (RGB) channels), we assume the image with the values $W_1 * H_1 * D_1$, a ConvLayer produces a volume of size $W_2 * H_2 * D_2$, where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$
- $D_2 = N$

The dimension D_2 of the output volume will always be equal to the number of filters applied and will correspond to the number of different activation maps calculated. With parameter sharing, it introduces $F * F * D_1$ weights per filter, for a total of $(F * F * D_1) * N$ and N biases.

In the output volume, the d-th depth activation map (of size $W_2 * H_2$) is the result of performing a valid convolution of the d-th filter over the input volume of stride S, and then offset by the d-th bias. Figure 2.5 represents an overview of the ConvLayer.

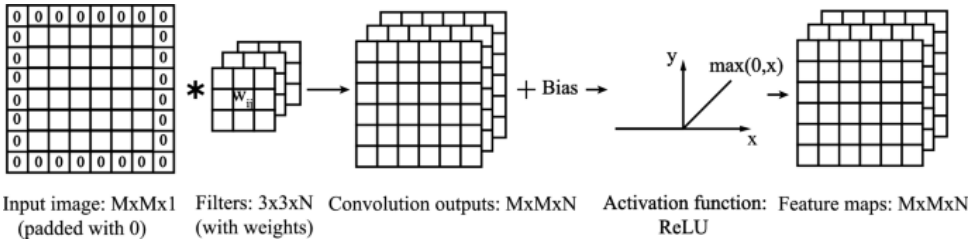

Figure 2.5: Convolutional Layer. From: [33].

Pooling layers generally follow a convolutional layer and can reduce the dimensions of the activation maps and parameters. They are similar to convolution layers, sliding the kernels through the feature maps with a certain stride. For example, for an activation map of N

x N dimensions, a Max Pooling layer with 2 x 2 kernels with stride 2, will reduce the dimensional space in half, to $N/2$ x $N/2$, selecting the maximum value within each kernel instead of calculating the dot products as in the ConvLayers.

Average pooling and Max pooling are the most common strategies. A study comparing these two methods [59] found that max-pooling can lead to faster convergence, select superior invariant features, and improve generalization. Due to these results, Max pooling is now present in the most recent CNN architectures [40] [22].

Fully-connected layers follow the last pooling layer and perform like a regular neural network. It enables us to feed forward the neural network into a vector with a predefined length containing the parameters learned from the convolution layers. Then we feed forward the vector into specific number categories for image classification, in which each neuron on the last layer corresponds to a certain score or probability of a certain label [40]. The last FC layer's structure can later be changed to be adapted to other tasks, preserving the parameters learned by the previous layers [52]. This method is called transfer learning and is explained in the next section 2.1.6.

Two or three stages of convolution, non-linearity, and pooling, are stacked, followed by more convolutional and fully-connected layers. Backpropagating gradients through a ConvNet is as simple as through a regular deep network, allowing all the weights in the activation maps to be trained. An overall example of a CNN hierarchical structure is represented in figure 2.6.

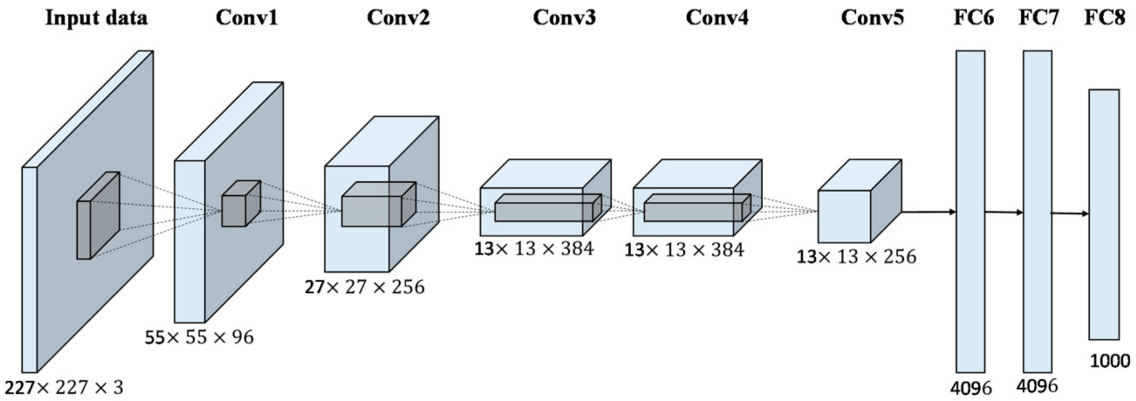

Figure 2.6: AlexNet: Convolutional Neural Network. From: [40]

2.1.6 Transfer Learning

Transfer learning is the process of transferring the knowledge from a related source to a related target [52]. Transfer learning is exploiting what has been learned in one setting to improve generalization in another setting [27]. In the field of CV, transfer learning is used to overcome situations where there are scarce samples of training data for some categories by adapting the already trained classifiers and their learned features to different categories or tasks. Other methods aim to cope with different data distributions in the source and target domains for the same categories, for example, due to lighting, background, and viewpoint variations.

The usual transfer learning approach is to train a base network and then copy its first n layers to the first n layers of a target network. The remaining layers of the target network are then randomly initialized and trained toward the target task. One can choose to backpropagate the errors from the new task into the base (copied) features to fine-tune

them to the new task, or the transferred feature layers can be left frozen, meaning that they do not change during training on the new task [70].

Typically, for the case of training CNNs for the tasks of Image classification and Object Detection, the second one is a better option. Freezing the transferred layers and their high number of parameters learned on large datasets, like the ImageNet Dataset [8], is the best option to avoid overfitting since the features learned are more generic. After that, re-train the network only by updating the weights of the Fully-Connected Layers to perform certain smaller tasks, like recognizing cars or calculating bounding boxes for object detection. In some cases, despite the task being simple, the target dataset may also be large. When this happens, it's preferable to fine-tune all layers, lowering the learning rate of the weights by 1/10 of the original (source) as a good starting point [70].

2.2 Methods

This section presents several methods that are used when developing models for image classification and object detection and that are mentioned throughout the work.

2.2.1 Data Preprocessing

Many application areas require sophisticated preprocessing because the original input comes in a difficult form for many deep learning architectures to represent, which is not the case for images. CV usually requires relatively little of this preprocessing. The images should be standardized so that their pixels all lie in the same, reasonable range, like [0,1] or [-1, 1]. Mixing images that lie in [0,1] with images that lie in [0, 255] will usually fail. Many computer vision architectures require images of a standard size, so images must be cropped or scaled to fit that size. Other architectures may rely on their pooling layers to adjust the sizes and do not need to be cropped [27]. In general, the images on the scientific community's datasets all come in the same size to avoid bad results.

Typically, other machine learning preprocessing methods like PCA or whitening are not used when working with images. For example, the AlexNet [40] has only one preprocessing step, that is, subtracting the mean across the training examples. The formula is $X' = X - u$, where X' is the normalized data, X is the original data, and u is the mean vector across all the features.

Standardization is another common form of data preprocessing that aims to make the data mean and variance normalized. The standardization of an input data X is calculated by $X' = (X - u)/s$, where X' is the normalized data, u is the mean vector across all the features and s is the standard deviation vector across all the features [54].

Another function is Zero Component Analysis (ZCA), this transformation makes the edges of the objects more prominent [54]. First, the data, X, is size normalized by feature scaling using by $X' = X/255$, then is mean normalized, and finally, whitening is performed by the following:

$$X_{ZCA} = U * diag(1/\sqrt{diag(S) + \varepsilon}) * U^T * X'$$

where $diag(M)$ represents the diagonal matrix of given matrix M, U is the Eigen vector matrix and S is the Eigen value matrix of singular value decomposition of covariance matrix, U^T is the transpose of the Eigen vector matrix U, ε is the whitening coefficient [54].

Data Augmentation, a process of transforming existing samples of the dataset or generating more samples for the dataset, is a form of preprocessing and regularization, and it's explained in section 2.2.3.

2.2.2 Hyperparameter Optimization

As mentioned before, hyperparameters take a fixed initial value before training and are not learned by the network, such as the other parameters. One strategy for optimizing the hyperparameters is the **Cross-Validation**, wherein a first stage, we train the model for only a few iterations to get the idea of how these parameters are behaving, and in a second stage, we extend the training time and fine search for the optimal hyperparameters. Other strategies are **Grid Search** and **Random Search** [18]. In Grid Search, for each hyperparameter, the user selects a small finite set of values to explore. The grid search algorithm then trains a model for every combination of hyperparameter values in the Cartesian product of the set of values for each hyperparameter. In contrast, Random Search replaces the exhaustive enumeration of all combinations by selecting them randomly [27]. This allows controlling the number of parameter combinations that are attempted explicitly.

The most common hyperparameters to be optimized are divided in two categories. The first ones are associated with the optimizer, such as the learning rate, the batch size, and the number of training epochs. The others are associated with the network structure, such as the number of layers in a DNN, the choice of activation function, and the weight initialization.

Looking at the hyperparameters in the first category, the number of epochs is the number of times the whole training data is shown to the network while training. Increasing the number of epochs can increase the training accuracy and thus result in overfitting, so this parameter needs to be carefully chosen. Instead of using all the datasets to train the model, we can use batches of samples from the training data, resulting in a more cost-effective training time. The batch size determines the number of subsamples of the dataset given to the network after which parameters are updated. Powers of two are common values to use for this size. All optimizers in the context of DNN have Learning Rate that has a hyperparameter. Learning rate decay overtime is an essential form of hyperparameter optimization while training the model since it increases performance and reduces training time [28]. Three main types of learning rate schedules:

- **Step Decay**, decay the learning rate by half every few epochs of training.
- **Time-Based Decay**, the initial learning rate is divided by the decay, where α and k are hyperparameters, and t the iteration or epoch:

$$\alpha = \alpha_0/(1 + kt)$$

- **Exponential Decay**, the initial learning rate is multiplied by an exponential factor, where α and k are hyperparameters, and t the iteration or epoch:

$$\alpha = \alpha_0 e^{-kt}$$

These hyperparameters cannot be optimized while training the network since they are referred to the network selection task and need to be optimized manually within the network structure category. The number of hidden layers is essential for the accuracy of the model.

Generally, adding more hidden layers improves accuracy but can be very computationally expensive and not needed. The number of layers can be optimized according to the problem at hand. The choice of activation function that should be used to process the inputs flowing into each neuron is a common hyperparameter since the activation function can impact the network's ability to converge and learn for different ranges of input values [27]. The way the weights are initialized for the first forward pass is also an important hyperparameter. A bad initialization of the weights can result in vanishing or exploding gradient, which will make it difficult to train the model. The common choice is using Xavier initialization [26].

2.2.3 Regularization

Regularization is any supplementary technique that aims at making the model generalize better and produce better results on the test set. Regularization helps the model to avoid overfitting, and thus perform better on unseen data. It can be included in various properties of the loss function, in the loss optimization algorithm, in the architecture of the network, in the data and other parameters [41], depending on the type of network, its goal, and the number of parameters it contains. For this work and in the context of CNNs we will be talking about Dropout, L2 Regularization for weight decay, Batch Normalization and Data Augmentation.

L2 Regularization

Considering the formula on section 2.1.4, we can add the term $\lambda R(W)$ to the Loss function, where λ is a hyperparameter that represents the strength of the Regularization and R is the function that regularizes the weights matrix W. This term is independent of the other parameters and is added to the Loss function chosen for the task:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

L2 Regularization, also known as Weight Decay, forces the weights to decay to almost zero. It is independent of the gradient of the loss function, and it makes the weights a little bit smaller for each backward pass. Smaller weights reduce the impact of the hidden neurons since those hidden neurons become negligible, and the overall complexity of the neural network is reduced. The L2 norm, also known as the Euclidian norm, is the sum over all squared weight values of the weight matrix W. Its represented as follows:

$$R(W) = \sum_{i}\sum_{j} W_{ij}^2$$

Dropout

Dropout [64] is a regularization technique that, in each forward pass, zeros out the activation values of randomly chosen neurons during training. The probability of dropping a neuron to zero can be considered a hyperparameter. For example, if we have a 0.5 probability of dropout, half the neurons will be set to zero. This constraint forces the network to learn more robust features rather than relying on the predictive capability of a small

subset of neurons in the network [61]. In the context of CNN, Dropout is applied on the FC layers. Another idea is the concept of Spatial Dropout, which follows the same principle as the regular dropout, but instead drops out entire activation maps rather than individual neurons, this being applied on the ConvLayers.

Batch normalization

Batch normalization is another regularization technique that normalizes the set of activations in a layer. Normalization works by subtracting the batch mean from each activation and dividing it by the batch standard deviation. This normalization technique, along with standardization, is a standard technique in the preprocessing of pixel values [61]. In addition, batch normalization speeds up the training process by improving the gradient flow through the network and allowing for higher learning rates [36]. As said before, the first step is to compute the empirical mean E, and variance Var, independently for each dimension of the activation maps, then normalize according to those values. For a layer with d-dimensional input $x = (x^{(1)}...x^{(d)})$, we will normalize each dimension:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

Then, for each normalized activation $\widehat{x}^{(k)}$, a scale and shift is performed by introducing the learnable parameters $\gamma^{(k)}$ and $\beta^{(k)}$:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

This will allow the network to squash the range of the normalized values if it wants to and recover the identity mapping.

Data Augmentation

Data Augmentation is both a form of data preprocessing and regularization. Augmenting the data means adding transformed images of the already existing ones to the dataset, helping the model to be more generic. Generalizability refers to the performance difference of a model when evaluated on previously seen data (training data) versus data it has never seen before (testing data). Models with poor generalizability have overfitted the training data [61]. These transformations help the model be more generic by presenting different shapes, colors, light saturation, random translations, and rotations. It can be applied both on the training set and the test set, the first to help the classifier recognize different versions of the classes and the second to test its ability to classify and detect identical figures to the ones seen on the training set. This type of data augmentation is called **Distortion**, some common examples for training datasets include:

- **Flipping**: flips the image horizontally or vertically.
- **Color Jitter**: randomize the contrast and brightness, perform augmentations on the RGB channels.
- **Random Crops and Scales**: random cropping different areas of the images and applying different scales.
- **Noise injection**: consists of injecting a matrix of random values usually drawn from a Gaussian distribution.

- **Translation and Rotation**: shifting or rotating the image to the right, up, left, or down.

Another form of data augmentation is **Image Occlusion**, used by most state of the art object detection models. It consists of occluding or erasing parts of the images, which helps the model to prevent memorizing the training data and overfitting. Some examples are:

- **Random Erase** - Replaces regions of the image with random values, or the mean pixel value of training set.
- **Hide and Seek** - Divide the image into a grid of square patches. Hide each patch with some probability of hiding.
- **Cutout** - Square regions are masked during training in the first layes of the CNN.
- **Grid Mask** - Regions of the image are hidden in a grid like fashion.
- **Mosaic data augmentation** combines 4 training images into one with different ratios. Allows to identify objects at a smaller scale than normal and reduce the batch size.

Some studies show other forms of Data Augmentation based on Deep Learning (DL) that include GAN-based data augmentation, Feature Space Augmentation, and Self Adversarial Training [61].

Self Adversarial Training uses the state of the model to inform vulnerabilities by transforming the input image. Unfortunately, it alters the images in a way that would be most detrimental to the model. Later on in training, the model is forced to learn this complex example.

2.3 Related Work

In this section, it will be presented the study was done in the field of object detection in general and vehicle detection, as well as an overview of image classification. It is also explored the state of the art CNN architectures and systems that integrate object detection and classification on edge.

Prior to DL being the state of the art in this field, more specifically, DNNs, conventional machine learning methods were used for the different tasks of visual understanding [53]. The conventional approach is to use well-established CV techniques such as feature descriptors for object detection. An example is Viola [67], the first framework for facial recognition in real-time, an essential task in object detection. It uses Integral Images, which allows the features used by the detector to be computed very quickly. It slides a box over the image to identify the facial features and select the important ones based on the AdaBoost learning algorithm.

Feature descriptors such as Scale-Invariant Feature Transform (SIFT) [50] are generally combined with traditional ML classification algorithms such as Support Vector Machines and K-Nearest Neighbours to solve the different CV tasks. SIFT aims at mapping the features of an object into the same object on a different scale, angle, or shape.

Histogram of Oriented Gradients (HOG) [23] can be considered as an important improvement of the SIFT and was introduced for pedestrian detection. To balance the feature invariance (including translation, scale, illumination, etc) and the non-linearity (on discriminating different objects categories), the HOG descriptor is designed to be computed on a dense grid of uniformly spaced cells and use overlapping local contrast normalization on blocks for improving accuracy [72].

Since DNNs used in DL are trained rather than programmed, applications using this approach often require less expert analysis and fine-tuning and exploit the tremendous amount of video data available in today's systems. DL also provides superior flexibility because CNN models and frameworks can be re-trained using a custom dataset for any use case, contrary to the traditional methods, which tend to be more domain-specific [53]. While CNNs where not new, used first for digit recognition in 1998 [44], since being reignited by Krizhevsky in 2012 [40] on the ImageNet Large Scale Visual Recognition Competition (ILSVRC), covered bellow, CNNs have dominated the domain of CV ever since due to a substantially better performance compared to traditional methods.

The ILSVRC [7], organized since 2010, evaluates algorithms for object detection and image classification at a large scale. One high-level motivation is to allow researchers to compare progress in detection across a wider variety of objects, taking advantage of the quite expensive labeling effort. Another goal of this challenge is to measure the progress of computer vision for large-scale image indexing for retrieval and annotation. This challenge has been a benchmark in object category classification and detection on hundreds of object categories and millions of images. The ImageNet dataset [8] contains twenty-two thousand categories and fourteen million images. The algorithms trained on the dataset intend to perform three different tasks: Image classification, Single-object Localization, and Object Detection.

There have been many breakthroughs in image classification and object detection tasks that originated from this challenge. The figure 2.7 shows the evolution of the winning entries on the ILSVRC from 2010 to 2015. Since 2012, CNNs have outperformed hand-crafted descriptors and shallow networks by a large margin, dropping the error rate percentage (y-axis) from 25.8% to 16.4%, peeking in 2015 with Residual Neural Network (ResNet) with an error of 3.57%.

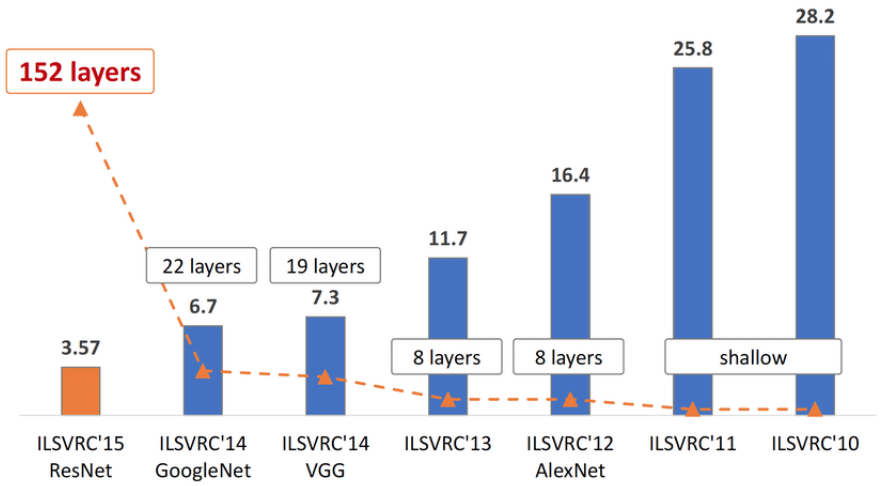

Figure 2.7: ILSVRC error rates over the years (from right to left).[51]

AlexNet follows the architecture explained in section 2.1.5. It's composed of eight layers,

including the FC layers. VGG [62] followed the same principle as AlexNet, using smaller filters and deeper networks. VGG increased the model's depth to 19 layers and used tiny 3x3 convolution kernels instead of 5x5 and 7x7 previously used in AlexNet.

GoogleNet [65] increased both of a CNN's width and depth up to 22 layers by creating inception modules that design a good local network topology (network within a network) and then stack these modules on top of each other. The inception modules use parallel convolution and pooling layers whose outputs are then concatenated together depth-wise.

ResNet [30], is a new type of CNN architecture that is substantially deeper than those mentioned previously, with up to 152 layers. ResNet aims to ease the training of networks by reformulating its layers as learning residual functions about the layer inputs. Stack residual blocks in which each residual block has two 3x3 ConvLayers. ResNet has only one FC layer, the output layer.

On the topic of deeper networks, DenseNet [32] serves as a base for different detection models. DenseNet connects each layer to every other layer in a feed-forward fashion. Whereas traditional CNNs with L layers have L connections, one between each layer and its next layer, DenseNet has $L(L+1)/2$ direct connections. The feature maps of all preceding layers are used as inputs for each layer, and their own feature maps are used as inputs into all subsequent layers. DenseNet has several compelling advantages, such as alleviating the vanishing-gradient problem, strengthening feature propagation, encouraging feature reuse, and substantially reducing the number of parameters.

By looking at the networks proposed in the ImageNet challenge, we can have an idea of different models for the project to be developed. These models can serve as engines for the object detection models, like Faster Region-based Convolutional Network (Faster R-CNN) [58] and You Only Look Once (YOLO) [56], since they are capable of learning high-level features trained on these large datasets and the accuracy of a detector depends heavily on its feature extraction networks. Since this can be very time expensive, applying the concept of Transfer Learning, explained in section 2.1.6, is sometimes the approach for certain tasks.

CNN-based Object Detection models can be divided into two categories, Two-Stage Detectors, and One Stage Detectors. Two-Stage Detectors are based on Region Proposals for the first stage, and in the second stage, the region proposals are sent down the pipeline for object classification and bounding-box regression. One Stage Detectors treat object detection as a simple regression problem by taking an input image, learning the class probabilities, and bounding box coordinates, having no proposal regions. Two-Stage Detectors reach the highest accuracy rates but are typically slower than One Stage Detectors [63].

The first example of CNN-Based Two-Stage Detector is Regions with CNN features (R-CNN) [25], which was the first to bring CNNs to object detection. It starts with the extraction of a set of object proposals, called Regions of Interests, by Selective Search. Then each region is re-scaled to a fixed size image and fed a CNN model trained on ImageNet to extract features. Finally, linear Support Vector Machine (SVM) classifiers are used to predict the presence of an object within each region and to recognize object categories. Although R-CNN has made great progress, the redundant feature computations on a large number of overlapped proposals, with over 2000 Regions per image, leads to an extremely slow detection speed.

To overcome this, later in 2015, Faster R-CNN[58] was proposed. The main contribution of Faster-RCNN is the introduction of a Region Proposal Network (RPN) that enables nearly cost-free region proposals. In contrast with R-CNN, where first the region proposals were

generated on the image and then fed to the CNN for feature extraction, RPN predicts proposals from features extracted by the CNN applied to the image. This led to a great increase in detection speed. From R-CNN to Faster R-CNN, most individual blocks of an object detection system such as proposal detection, feature extraction, and bounding box regression have been gradually integrated into a unified, end-to-end learning framework.

One-stage detectors have been the topic of discussion in Object Detection due to their increase in detection speed. YOLO [56] was the first of this class of detectors, and it follows a different approach: apply a single CNN to the full image. This network divides the image into *nxn* grid cells and predicts bounding boxes and class probabilities for each grid cell simultaneously. Due to this pipeline, it only predicts less than 100 bounding boxes per image while R-CNN using selective search predicts 2000 region proposals per image [37]. YOLO frames detection as a regression the problem, so a unified architecture can extract features from input images straightly to predict bounding boxes and class probabilities.

The experiments showed that YOLO was not good at accurate localization, and localization error was the main component of prediction error, struggling to localize small objects. The most recent versions of YOLO, the most recent being Yolov3[57] and Yolov4 [20], address this issue by using multi-label classification to adapt to more complex datasets containing many overlapping labels and by utilizing three different scale feature maps to predict the bounding boxes.

Another example of one stage detectors is Single Shot MultiBox Detector (SSD) [48]. Given a specific activation map, instead of fixed grids adopted in YOLO, the SSD takes advantage of a set of default anchor boxes with different aspect ratios and scales to discretize the output space of bounding boxes. In order to handle objects of various sizes, the network fuses predictions from multiple feature maps with different resolutions.

The state of the art models for object detection presented before, while excelling in accuracy, are running on multiple Graphics Processing Units (GPUs) with high memory capacity to support the millions of parameters of their CNN engines. Therefore, when deploying the models on edge hardware, sometimes it is necessary to compress or re-design the model to maintain a stable rate of Frames per Second (FPS), low Inference Time (IT) and low memory requirements.

Both SSD and YOLO present specific versions to be running on embedded edge devices, they are Tiny-SSD [69], and Yolov3-Tiny [57], that aim to minimize the model's size while maintaining object detection performance.

In [45], the authors use network pruning to reduce unnecessary computation and increase the execution speed, dropping some of the unnecessary weights. In other cases, some models are designed specifically for mobile, and embedded vision applications, such as MobileNets [31]. MobileNets introduce two simple global hyper-parameters that efficiently trade-off between latency and accuracy. These hyper-parameters allow the model builder to choose the right sized model for their application based on the problem's constraints.

Focusing in vehicle detection using CNNs, in [24], the authors train a semi-supervised CNN to classify different types of vehicles. For a given vehicle image captured from video traffic cameras, the network can provide the probability of each type to which the vehicle belongs. The model learns features from unlabeled data as well as labeled data.

The work explored by the authors in [15], studies several one-stage detectors for pedestrian and vehicle detection in embedded devices, more specifically the Jetson Nano. The articles review the performance of the different networks and show that SSD with MobileNetV1

feature extractor, from the MobileNet family of CNN, presented the highest accuracy for detecting vehicles and had lower processing time. The authors also made some remarks for the Jetson Nano hardware [10], explained in section 2.4.2, stating that it has a good performance but not the best compared to the most expensive models with better features, but its development kit allows the implementation of complex models to create a variety of applications at a low budget.

In [42] the authors present a solution based on YOLO to recognize license plates in traffic. The model was trained from a dataset containing videos from the traffic where both cameras and vehicles are moving, with different types of vehicles.

Sometimes the quality of the cameras for object detection have low quality. In [16], the authors evaluate the performance of CNN in the detection and classification of vehicles using low-resolution traffic images. The developed network is applicable for real-time traffic monitoring implementations using 4-input low-resolution traffic video at 6 FPS.

2.4 Technologies

2.4.1 Deep Learning Frameworks

A brief study was made in order to choose which framework to use to develop the network. These frameworks allow to easily build big computational graphs, easily compute gradients in computational graphs and run it all efficiently on GPUs. The two most popular are Tensorflow [13] and PyTorch [12].

Both are excellent frameworks for DL. According to the article, [4], both have since adopted good features from each other and become better in the process. There are still things that are slightly easier in one compared to another, but it's now also easier than ever to switch back and forth between the two due to increased similarity. But there are still some differences.

The major difference comes from the purpose of the project to be developed. PyTorch offers better development and debugging experience and is more suitable for research, creating highly-customized architectures. If focusing on research, the production of non-functional requirements is not very demanding [5]. On the other hand, TensorFlow is a very powerful and mature deep learning library with strong visualization capabilities and several options to use for high-level model development. In addition, it has production-ready deployment options, and support for mobile platforms [6].

Since we want to deploy our solution to the Smart Lamppost for this project, the flexibility when it comes to deploying models and the support for mobile platforms from Tensorflow is a major deciding factor. The integration of a high-level wrapper like Keras [9] with Tensorflow is another advantage. Keras allows for easy production of models and easy and fast prototyping. It is scalable, and its possible advantage is the processing power of distributed environments or machines with multiple GPUs. The combination of the Tensorflow/KerasAPI also offers support for CNNs, making it easy to build and analyze these networks. For these reasons, the framework chosen to elaborate this project is Tensorflow integrated with Keras. Table 2.1 show a qualitative comparison between the frameworks.

	Tensorflow/KerasAPI	Pytorch
Level of API	High Level API	Lower-level API
Speed	High	High
Architecture	Easily interpretable	Complex
Debugging	Difficult to debugging	Good debugging capabilities
Uniqueness	Object Detection Functionalities	Flexibility
Graphs	Static Graphs	Dynamic Graphs
Focus	Production/Deployment	Research

Table 2.1: Qualitative Comparison between Frameworks

2.4.2 Edge Hardware

The choice about the hardware we want running the deployed DNNs on the edge nodes at the Smart Lamppost is important in many aspects, such as versatility to the different Deep Learning Frameworks, the CPU usage, price, and performance on test. We studied the options available at the market and made a comparison between the most popular ones.

The debate is between Nvidia Jetson Nano, Edge Coral GPU, and Intel NCS. Comparing these three [11], the Nvidia Jetson Nano presents better FPS and IT than the other two. However, it uses more memory. It also uses fewer CPU resources, which is important when deploying complex and multi-layer DNNs on edge. Prices are almost identical for these options, but there are other high-end expensive options that may be on the market.

Another comparison was the Nvidia Jetson Nano Board and the Edge Coral TPU Board. It argued that the Coral Edge TPU Board is better as it comes with Wifi and BT support and other components such as sensors for temperature and ambient light. However, we opted for the latter since these features are not in our interest, and the Jetson Nano has better software support.

For our project, the Nvidia Jetson Nano's [10] versatility in DL frameworks, its focus on Deep Learning on the latest Jetson versions, the good performance, and better confidence scores, make this a better option at this early stage on the overall project. Another factor is that Nvidia Jetson Nano does not need a host for data processing, which is an advantage for this project since we want to process the data locally. Table 2.2 shows the comparison between edge hardware made by done in [11], a simple simulation was made using the same model.

Parameters	Nvidia Jetson Nano	Google Coral	Intel NCS
Inference Time	38ms	70 to 92.32ms	225 to 227 ms
FPS	25	9 to 7	4.43 to 4.39
CPU usage	47 to 50 %	135 %	87 to 90 %
Memory usage	32%	8.7 %	7 %

Table 2.2: Comparison between edge harware devices

2.5 Review

This chapter presents research about tools and a review of the concepts and literature related to the proposed objectives. Important conclusions for the correct development of the model can be taken from studying the state of the art. Following next is the discussion

of the several key points taken from the state of the art, as well as the advantages and disadvantages of deep learning, feature extraction methods, using pre-trained models, the trade-off between speed and accuracy, and the optimization of DNNs.

- **DL vs Feature Extraction Methods**: Deep Learning has pushed the limits of what was possible in the domain of CV. However, that is not to say that the feature extraction methods which had been undergoing progressive development in years before the rise of DL have become obsolete. Since neural networks used inDL are trained rather than programmed, applications using this approach often require less expert analysis and fine-tuning and exploit the tremendous amount of video data available in today's systems. One advantage of using DL is that it is an end-to-end learning process, where given an input dataset, the machine learns the underlying features and produces an output. While in feature extraction methods, the features must be manually extracted and selected and then passed through a shallow network for classification. For the project, it was decided to use DNNs due to the better accuracy results they present, since nowadays it is possible to deploy these networks to mobile/edge devices without many computational costs, where in the past, it was a difficult task due to the lack of powerful edge devices and unoptimized DNNs for production purposes.

- **Optimizing the DNN**: When developing a network, it's essential to take into consideration the different techniques to optimize its learning process, its capacity for generalization (avoid overfitting), and increasing accuracy. The choice of the optimizers, the number of layers, learning rate schedules, and data preprocessing play an important role in the learning process. Techniques described in regularization, such as Data Augmentation and Batch Normalization, are present in every state of the art process of training networks in CV and are essential for the model's ability to generalize. Tweaking these parameters is necessary for good results.

- **Using Pre-Trained Models**: Using pre-trained models through transfer learning, with optimized parameters and architectures, and containing features learned from large benchmark datasets like ImageNet and COCO, can be beneficial in reducing training time and overfitting since the features learned are more generic. However, these models may have millions of unnecessary parameters for vehicle detection, resulting in large memory and computational costs when deploying to the edge, becoming unviable for object detection in real-time. One approach is to use the pre-trained networks on ImageNet like ResNet as feature extractors and fine-tune the rest of the network for the task of bounding box regression and classification. Another option is to follow the pre-trained model's architecture and hyperparameters values, developing and training a CNN on the desired data.

- **Trade-off between speed and accuracy**: When considering solutions on edge devices, it is important to balance speed, memory, and accuracy. When comparing the two-stage and one-stage detectors, we observe that the first ones have more accuracy and memory requirements but less speed. This is due to many different parameters, such as the model's size and the number of region proposals per image. When developing, it's essential to know these trade-offs for better performance on the mobile device.

This page is intentionally left blank.

Chapter 3

Approach

After analyzing the current state of the art in object detection and image classification, following the study of several technologies and the defined objectives and challenges in mind, a plan for the development of the work is presented, as well as an overview of the data that will be used and the metrics for evaluating the solution.

3.1 Approaches to Experiment

For this project, in order to find the best solution, different models will be trained and compared. One way of doing this is training and optimizing the model from scratch, the other is to use pre-trained models trained on large datasets, using transfer learning 2.1.6. The models created for the different approaches will then be compared against each other to find the one with better performance. To compare the models, we'll use the metrics in section 3.3.

3.1.1 Creating the Model

It is possible to create the model from scratch based on a state of the art Deep Learning (DL) architecture. Using Tensorflow/KerasAPI it is possible to create these models in a structured way. The next steps represent a general approach for creating the model:

- Create the Convolutional Neural Network (CNN) for feature extraction, defining the architecture following the model structures of ImageNet classifiers, making it simpler and adapted to the problem. Start by creating the layers of the network and set hyperparameters, such as initial learning rate and batch size.
- Choose the optimizers and the loss functions that are most fit for the learning process.
- Train the network on the Common Objects in Context (COCO) dataset to learn the features.
- Optimize the model through different techniques.
- Use an object detection framework on this network.

There are some advantages and disadvantages to this approach. The advantages are the complete control over the model's design so that we can define all the parameters needed

for the learning process. This means the model can be more fit to be running on the edge. The main disadvantages are that this model may not meet the same standards of accuracy and precision as other already existing models for the same task and training will be much harder, since training CNN requires large amounts of data, resulting in is very time-consuming approach.

3.1.2 Using Pre-Trained Models

The following approach uses pre-trained models through transfer learning, optimized parameters and architectures, and features learned from large benchmark datasets like ImageNet and COCO. The approach consists of using the pre-trained networks on ImageNet as feature extractors and fine-tuning the rest of the network for the task of bounding box regression and classification for detection of vehicles specifically.

They are some advantages of using Pre-Trained models. One advantage is it reduces training time and overfitting since the features learned are more generic, retraining our model only to learn the features from the vehicles. Another advantage is the deployment simplicity of the models presented for edge devices since they are already optimized for real-time detection. The disadvantages are that these models may have millions of parameters that may be unnecessary for vehicle detection, which will result in large memory and computational costs when deploying to the edge, so focusing on optimizing the model for vehicle detection is important. Each model will be taken into consideration, as well as their lightweight versions that might be explored in this approach.

One-Stage Detectors

For this approach, the focus is on One-Stage detectors since the most recent models give a higher speed than Two-Stage detectors and can maintain a relatively good level of accuracy [63]. More specifically, we'll be focusing on You Only Look Once (YOLO) and Single Shot MultiBox Detector (SSD) models and their variants. Both these models follow the same meta-architecture, presented in figure 3.1 below.

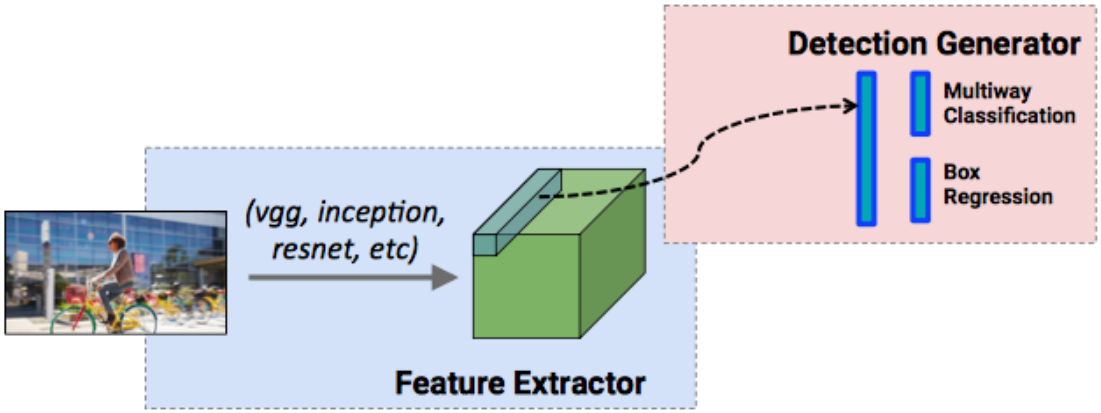

Figure 3.1: Meta-architecture for One Stage Detectors. From: [34].

Through the use of transfer learning, we have two options. First, transfer learning via feature extraction, where use the backbone feature extractors from YOLO and SSD, respectively Darknet and MobileNet, and only adapt the last fully connected layers for the

different vehicle classes. The other approach is via fine-tuning, which requires updating the feature extractor's weights themselves and then re-training the networks on the vehicle datasets.

You Only Look Once V4 (YOLOV4)

YOLOV4 is the most prominent model of the One-Stage Detectors. The architecture of the network follows the same meta-architecture presented in figure 3.1, and will be explained in more detail.

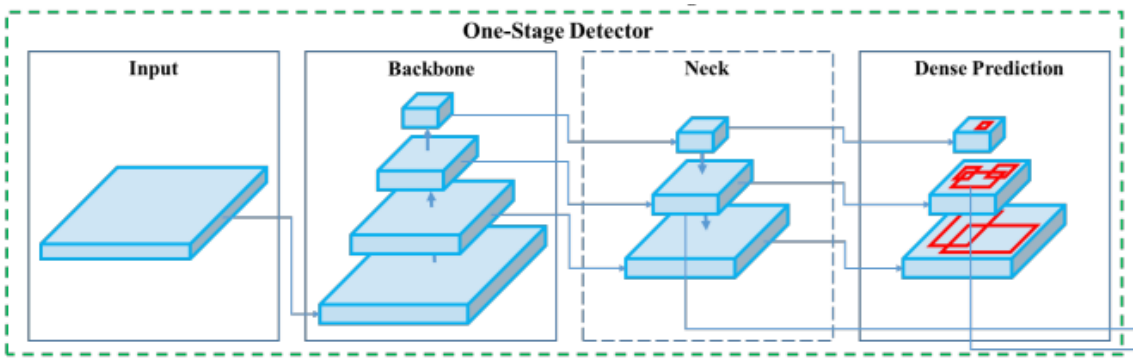

Figure 3.2: YoloV4 Architecture. From: [20]

The architecture of YOLOV4 is represented in figure 3.2 and is divided into four essential components: Input, Backbone, Neck, and Dense Predictor.

Starting with the **Input** phase, the images go through a process of data preprocessing and data augmentation in the training pipeline. The images need to be in the correct resolution size (multiples of 32x32), and it has standardized aspects such as grayscale (shades of black and white) and contrast. The data augmentation techniques applied by YOLOV4 are Distortion, Self Adversarial Training, and Image Occlusion, described in section 2.2.3, with the latter having a special focus on Mosaic Data Augmentation and Cutout. These techniques are also applied by the backbone, explained next.

After the input layer, the images go through the CNN feature extractor, also called **Backbone** in Object Detection models. The backbone is composed of three components in YOLOV4, the network called **CSPDarknet53**, the **Bag of Freebies** and the **Bag of Specials**.

CSPDarknet53 is pre-trained on ImagenNet dataset and is based on DenseNet, explained in section 2.3, but edited in a way that it becomes a CSPNet [68]. This is done by separating the feature map of the base layer, copying it, and sending one copy through the dense block, and sending another copy onto the next stage of the network. The idea with CSPDarknet53 is to remove computational bottlenecks in the DenseNet and improve learning by passing on an unedited version of the feature map. This mechanism can visualized in figure 3.3. Two copies are made from the output x_0 from the previous dense block. One copy, $x_0{}'$ is passed through is passed to the Partial Transition Layer, while $x_0{}''$ is passed through the Dense Block.

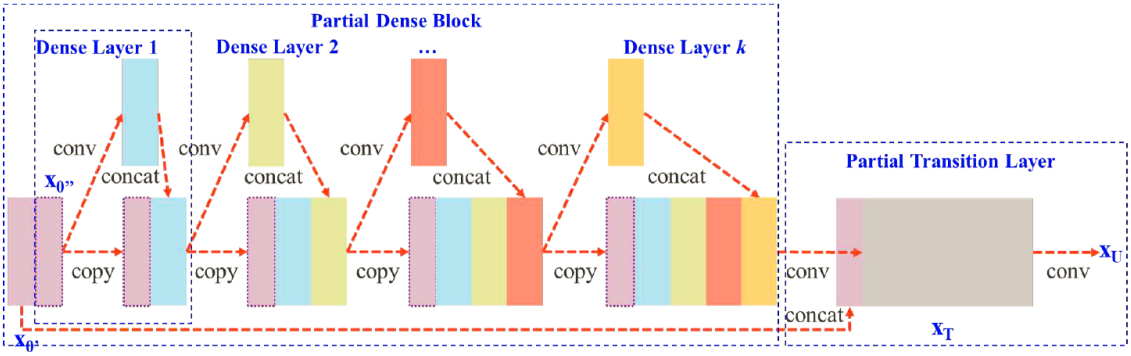

Figure 3.3: Partial CSPDarknet53 Architecture.[68]

The **Bag of Freebies** methods are the set of methods that only increase the cost of training or change the training strategy while leaving the cost of inference low. Data augmentation is one component of this that was explained earlier in this section. The other component is to reduce bias, implementing Focal Loss [46] and IoU loss[20], mentioned in section 2.1.4.

The **Bag of Specials** methods are the set of methods that increase inference cost by a small amount but can significantly improve the accuracy of object detection. The Mish activation function [20] is part of this bag for the Backbone. Due to preserving a small amount of negative information, Mish eliminated by design the preconditions necessary for the ReLU function.

After the formed features from the backbone, they need to be aggregated and prepared for the next step in detection. For the **Neck**, YOLOV4 uses PANet [47] for the feature aggregation of the network. A demonstration is seen in Figure 3.4, where each P_i represents a feature layer from the backbone, where the features are fused following a bottom-up and top-down pathway and connect only a few layers at the end of the convolutional network. Another layer used in the Neck is SPP [29]. The SPP layer is a type of pooling layer, as described in section 2.1.5, that allows generating fixed size features, whatever the size of the feature maps. To generate a fixed size, it will use pooling layers like Max Pooling, for example, and generate different representations of the feature maps.

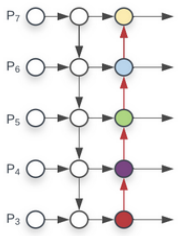

Figure 3.4: PANet demonstration. From:[66]

After the features are aggregated, the step of **Dense Prediction** begins. This component, typically called **Head** of the model, has the role, in the case of a one-stage detector, to perform dense predictions. The dense prediction is the final prediction which is composed of a vector containing the coordinates of the predicted bounding box (x_{center},y_{center},$width$,$height$), and the confidence score of the predicted location and label of the object. It follows the same Head as in YoloV3 [57] and previous versions of YOLO. General anchor-based detection follows these steps:

- Generate a high number of anchor boxes for each predictor that represent the ideal location, shape, and size of the object it specializes in predicting.
- For each anchor box, calculate which object's bounding box has the highest Intersection Over Union (IOU) over the ground truth box.
- Depending on the value of the IOU, tells the network if it should or not predict an object on that location or if it is ambiguous.

In the final layer for prediction, 3 anchor boxes are generated for each grid cell, as seen in figure 3.5, each anchor box containing the bounding box coordinates, the classes scores and the overall confidence score. The confidence score is calculated as follows:

$$P(Class_i) * IOU = P(Class_i|Object) * P(Object) * IOU$$

where $P(Class_i|Object)$ is the condition probability of the class i, where i ranges from 0 to total number of classes, knowing there is a object in the grid, $P(Object)$ the probability of an object being in the grid and the IOU. If there is no object in the grid cell, the confidence score will be 0.

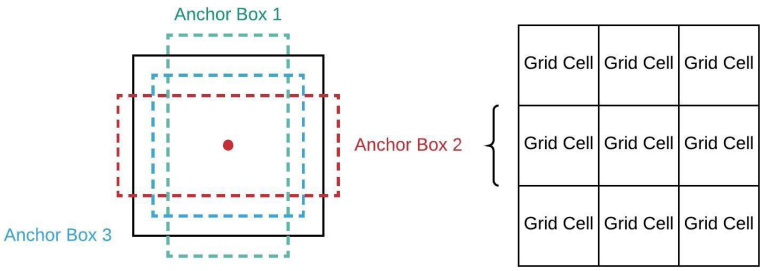

Figure 3.5: Anchor Boxes in Yolov4. From: [20]

Different from YoloV3 is the CIoU Loss [20] introduces two new concepts compared to IoU loss. The first concept is the concept of central point distance, which is the distance between the actual bounding box center point and the predicted bounding box center point. The second concept is the aspect ratio, comparing the aspect ratio of the true bounding box and the aspect ratio of the predicted bounding box. With these 3 measures, it is possible to measure the quality of the predicted bounding box. The CIoU Loss is described as follows:

$$\mathcal{L}_{CIoU} = 1 - IoU + \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{c^2} + \alpha v$$

where b and b^{gt} denote the central points of the predicted box and the ground truth box respectively, ρ is the Euclidean distance, c is the diagonal length of the smallest enclosing box covering the two boxes, α is a positive trade-off parameter, and v measures the consistency of aspect ratio.

Some other things are included in the bag of specials. One is Cross mini-Batch Normalization (CmBN) [19], a form of Batch Normalization explained in section 2.2.3, that consist of normalizing mini-batches across four steps. The other is DropBlock regularization [71],

in which sections of the image are hidden from the first layer to force the network to learn features that it may not otherwise rely upon.

Two-Stage Detectors

More recent Two-Stage detectors can also be applicable for object detection in real-time if optimized correctly. Figure 3.6 represent the meta-architecture for Two-Stage detectors. Another option for the models to be experimented with will be Faster-RCNN with ResNet [34], since we can reduce the number of Region Proposals generated for each image, reducing the precision but increasing computation speed. Since the experiments will be conducted in the future, this option is still open if we need more accuracy.

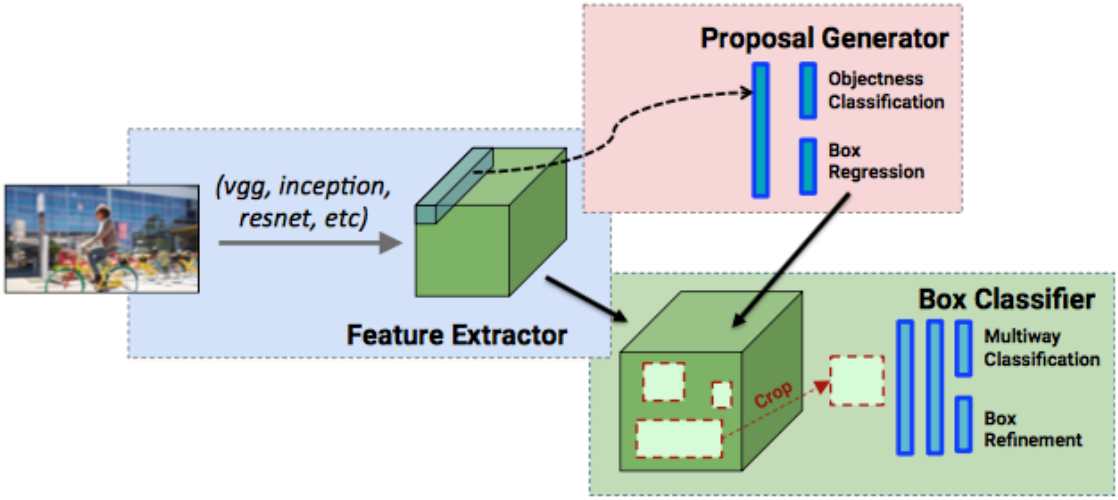

Figure 3.6: Meta-architecture for Two Stage Detectors. From: [34].

3.2 Datasets

For this project, data is needed to train the model as well as for validation. Since the project requires real images of different objects so we can train the model to identify and classify cars in a variety of objects, some research as made for datasets containing vehicles. Three options of public datasets were found that were explored for the project.

The first is the COCO Dataset [2]. The COCO Dataset is a public dataset already labeled, which has more than two-hundred thousand labeled instances with objects classified in eighty super categories, including vehicles. The dataset is split by Training Set, Validation Set e Test Set. The training set contains the images to train the model. The validation set contains the images with the ground truths annotations, i.e, each image contains the information about the ground truth values of the bounding box of each object, the different classes present in an image, and the number of objects of a certain class present. The Test Set will be used to evaluate the model's accuracy on unseen data and without the ground truth annotations.

The second is the Stanford Cars Dataset [39]. It is a public dataset that contains 16,185 images of 196 classes of cars. The data is split into 8,144 training images and 8,041 testing images, where each class has been split roughly in a 50-50 split. Classes are typically at the level of Make, Model, Year. While the COCO dataset is more focused on object detection,

this dataset can also be used for object detection but it's more focused on the fine-grained classification of the vehicles.

The last choice is the Boxi Vehicle Dataset [17] is a recent public dataset with 1.99 million annotated vehicles in 200,000 images, including sunny, rainy, and nighttime driving. This dataset offers another view for annotations, all vehicles are split into their visible sides, which creates a 3D-like bounding box impression, instead of the axis-aligned and segmentation seen in other datasets such as COCO.

From the different options, the COCO Dataset [2] will be used to train and validate the models since it offers good annotations for the images, including segmentation and bounding box coordinates. It also offers a coded structure to evaluate the models on the dataset, making comparing the models easier. On the other hand, the Test Set will only be used in the project's initial phase since we want to test the model on the frames captured by the camera on the edge hardware.

3.3 Evaluation of Models

To evaluate the models, this task will be divided between two types of evaluation: Evaluation of the models on edge hardware and evaluation on COCO Test Set. The primary task will be the evaluation and comparison between the different models on the COCO dataset, followed by an evaluation on the edge device. In the following subsections are presented the metrics for the evaluation.

3.3.1 Intersection Over Union

The IOU metric is often used to judge object detections in images using their bounding box overlap with the ground-truth object. Detections usually are considered correct when the area of overlap A between the predicted bounding box B_p and ground-truth B_{gt} exceeds 50%.

$$A = \frac{area(B_p \cap B_{gt})}{area(B_p \cup B_{gt})}$$

3.3.2 Precision and Recall

Precision is a measurement of the proportion of results retrieved that are considered to be correct. Recall is a measurement of the proportion of all possible correct results that the classifier actually detects. Precision and recall are inverse correlated, that is, as precision increases, recall generally decreases, and vice versa.

For each image, the information about the ground truth values of each object's bounding box, the different classes, and the number of objects of a certain class present come in the form of annotation.

By computing IOU for each detection in an image, we set a threshold for converting those real-valued scores into classifications, where IOU values above this threshold are considered positive predictions and those below are considered to be false predictions. More precisely:

- **TP (True Positive)**: The object is in the image and the model detects it, with IoU >= 0,5

- **FP (False Positive)**: Two cases, the object is in the image , but IoU $< 0{,}5$ or the object is not in the image, but the model detects it.
- **FN (False Negative)**: The object is in the image, but the model doesn't detect it.
- **TN (True Negative)**: The object is not in the image, and the model doesn't detect it. They are not used for object detection specifically.

Precision and Recall are calculated for each class present in an image.

$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN}$$

The F1 score is an overall measure of a model's accuracy that combines precision and recall, and it represents the weighted average of Precision and Recall. This score takes both false positives and false negatives into account and can be calculated as follows:

$$F1 = 2 \times (\frac{Precision \times Recall}{Precision + Recall})$$

3.3.3 Average Precision and Mean Average Precision

The Average Precision (AP) is based on the Precision/Recall Curve that varies according to a certain threshold. Changing the threshold affects the values of precision and recall, which may lead to an increase in precision or recall. It is calculated for each class the model is trying to predict. The formula is as follows:

$$\text{AP} = \sum_{n} (R_n - R_{n-1}) P_n$$

where P_n and R_n are the precision and recall at the n-th threshold. The Mean Average Precision (mAP) is simply the AP for all classes. It can be described as follows:

$$mAP = \frac{1}{N} \sum_{c}^{N} AP_c$$

The mAP is associated with a threshold value, for example, the IOU at 0.5 or 0.75. This allows for models to be compared on the same parameters, and it will be useful for the project to compare the different models on the same dataset.

3.3.4 Evaluation on Hardware

For the evaluation of the model on the hardware it will be considered the following parameters:

- **Frames per Second (FPS)**: The rate at which the model detects the objects.
- **Inference Time (IT)**: The time the model takes to make a prediction (in ms).

- **Central Processing Unit (CPU) and Graphics Processing Unit (GPU)**: If the model consumes all the limited memory and bottlenecks the GPU or the CPU. Values are qualitative and range from Low, Medium and High.

Comparing between different models solely on its performance on hardware, the preferable model will have the highest FPS rate, the lowest IT, and low memory requirements.

3.4 Requirements

In this section, the non-functional requirements for the solution that was developed will be presented. Throughout the internship, the requirements were subject to several iterations, with the participation of both advisors. Following are the priorities for the requirements needed for this project:

- **M (Must have)**: mandatory to be included.
- **S (Should have)**: are not essential, but add significant value.
- **N (Nice to have)**: helpful to have, but not essential if left out.

3.4.1 Functional Requirements

This type of requirements state the main functionalities the final solution should offer by describing the potential features that need to be implemented for it to be classified as a viable solution.

FR1 - Detection and Classification
Description - The final inference model should be able to correctly detect and classify vehicles.
Priority - **M**

FR2 - Hardware
Description - The final inference model should be able to run on the specified hardware
Priority - **M**

FR3 - Count Vehicles
Description - The overall system should be able to correctly count the number of vehicles.
Priority - **N**

FR4 - Robustness
Description - The overall system should be fault-tolerant. In events in which the camera is obscured or blocked, or the system can't perform detection, it should handle these events accordingly.
Priority - **N**

3.4.2 Non-Functional Requirements

The non-functional requirements are the requirements that define the criteria that can be used to evaluate the final solution as a whole. Associated with properties such as response time, they can impose constraints or restrictions on the design of the solution. For this purpose, quality attributes that describe how the solution should behave are enumerated in this section:

NFR1 - Confidence Score
Description - The final solution be able to detect objects with a minimum confidence score of 50%.
Priority - **M**

NFR2 - Performance
Description - The final solution should be able to detect objects with a 6 to 10 minimum rate of FPS
Priority - **M**

NFR3 - Hardware Capacity
Description - The final solution should be able to run on the specified hardware without exceeding its CPU and GPU capabilities.
Priority - **M**

NFR4 - Average Precision
Description - In comparison with the other models, the chosen model must have a minimum of 50% Average Precision.
Priority - **S**

NFR5 - Memory
Description - The model's memory should be within the bounds of the hardware's memory capacity.
Priority - **S**

3.4.3 Constraints

The only constraints imposed by Ubiwhere were related to business constraints that arise from business decisions and must be satisfied with the system's architecture. In this case, every technology, except the hardware used, involved in developing the final solution had to be open source. The listing of the constraints are as follow:

C1 - Licenses(Commercial Level)
Description - The licenses of the tools used had to be under a license that allows their utilization for commercial purposes.

C2 - Licenses(Free Use)
Description - The licenses of the tools and data had to be should be available without any fee associated.

C3 - Schedule
Description - The final solution must be finished by 07/09 and the thesis delivered on September 07.

C4 - External Data
Description - All the datasets used for training and testing of the models had to be open source and reproducible for commercial use.

3.5 Review

This chapter was instrumental for allowing the definition of the approaches to be followed, the choice of a dataset from the options available, the definition of metrics from which to evaluate the models, and the requirements.

From the approaches, we were able to guide the development process, starting with the creation of the model and subsequently using pre-trained models, that lead to the implementation of a One-Stage Detector, more precisely, YOLOV4 and its variant YOLOV4-Tiny.

For the choice of a dataset, we looked at the one that had good annotations for the images, including segmentation and bounding box coordinates, and the COCO dataset presented itself as a better choice.

For the evaluation of models, it was defined the different metric to evaluate the models both on precision in the test set of COCO, as well as the evaluation of the model on hardware.

Finally, this chapter also defined the requirements for the work at this internship, including the functional and non-functional requirements and the possible constraints for when developing.

Based on all the knowledge from this chapter and everything defined, we can transition to the development phase.

This page is intentionally left blank.

Chapter 4

Development

Following the approach defined in the previous section, the first step in development is to prepare the dataset and extract the information from it. After the data extraction is completed, the first approach of creating the model from scratch is initiated, where the steps in section 3.1.1 are followed. After the first approach, we move to using pre-train models, from which 3 choices were taken, You Only Look Once V4 (YOLOV4), YOLOV4-Tiny and Single Shot MultiBox Detector (SSD) with MobileNetV2. After this phase, the models were evaluated in using the parameters described in section 3.3. Finally, in order to evaluate the models in the hardware, as described in 3.3.4, the models were tested in the hardware for inference. This chapter will also present some of the difficulties that were encountered during this process.

4.1 Data Extraction from COCO

For the Common Objects in Context (COCO) dataset [2], the Cocoapi [3] was used to easily load the images annotations, as well as the different categories IDs. From the different 80 categories, we want to extract, both from the training set and testing set, the images related to the category "vehicles". It was decided to use only the 5 different categories,"car","motorcycle","bicycle","bus" and "truck", that are present in the COCO dataset both for training and validation, to reduce the training time and size of the network (number of parameters).

For the different images selected, a data structure was created, where each element contains a dictionary where each image has the associated values for the bounding boxes of all object in that image as well as the the class ID for each of those objects. Some images that repeatedly appeared in the structure were removed when selecting images that contained the same class of objects. This process was common for all the different approaches and models. The bounding boxes are represented for each object with the values [x_{min},y_{min},$width$,$height$], where x_{min} and y_{min} represent the initial points of the bounding box and the $width$ and $height$ its size. The class IDs are represented by an integer associated with the class.

Depending on the approach and model, further different steps were taken to pre-process the data since other models required different processes. These steps are described for each model in the following sections.

4.2 Creating the Model

The first approach of the work was to create a model from scratch, including the neural network to extract the features from the images and the subsequent layers for the classification and the inference for the objects. One of those steps described in 3.1.1, was to create the Convolutional Neural Network (CNN) for feature extraction from scratch. This step was found to be inefficient since it is more productive to use a neural network pre-trained in a large dataset (such as ImageNet), available in the Keras Application library, and only construct the layers that will receive the output of the backbone network. This allows for shorter training time and less time spent adjusting and fine-tuning the different parameters of the network used for training with different models using transfer learning. For this naive model, the CNN MobileNetV2 [31] was chosen for the feature extraction, with previously trained weights on ImageNet.

After the training process was completed, the results for this model was unsatisfactory, and problems for the optimization of the model and evaluation on the COCO were encountered. After the use of techniques such as data augmentation to optimize the model, it unexpectedly behaved the same way. After some time constraints and later deployment issues, this motivated the search for other alternatives and the use of pre-trained models to obtain good detection results. It was suspected the bad results were derivative from the complexity involved in correctly predicting multiple objects in a single image, at different distances, shapes and shadows, and how this simple model would require more optimization strategies and a rethink-ed model structure, that was not feasible at the time in the development phase due to time constraints.

4.2.1 Pre-Processing

After the extraction described in the previous section 4.1 for COCO, the images were loaded and converted to a resolution of 224x224 since the network requires this resolution for the input images. This is a common practice since different networks work with different resolutions.

The images were normalized by a single step of dividing the array that represents the images by 255 (maximum value for the Red, Green and Blue (RGB) channels), creating values between 0 and 1. The bounding boxes for each image were also resized to the image's new dimension, and this was done by dividing the values of the bounding boxes by the width and height of the original image, scaling the values between 0 and 1. Furthermore, a Label Encoder was used to transform the values of the labels to a range of 0 to N-1, where N is the number of classes.

For training, 500 images were selected from the train set in COCO, and for validation, 300 images from the validation set.

4.2.2 Model Structure and Parameters

Since the pre-trained MobileNetV2 will be used, it is required to freeze the body of the network such that the weights will not be updated during the fine-tuning process, removing the last Fully Connected (FC) layer, to construct a new layer head responsible for multiple-output predictions (class and bounding boxes). The output from the last layer of the convolution network, the Rectified Linear Unit (ReLU) activation layer, is forwarded to a flatten layer, and its output will be forwarded to the next two layers, as seen in figure 4.1.

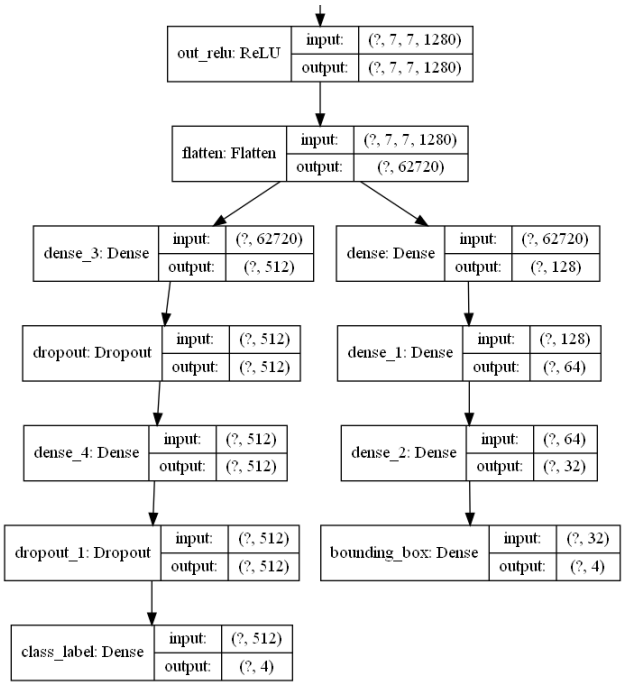

Figure 4.1: Naive model arquitecture after the the last output from MobileNetV2.

The first branch of this head layer is for the Bounding Box Regression, this branch is a simple fully-connected sub-network, where each fully connected layer consists of 128, 64, 32, and 4 nodes, respectively. In the last FC layer, each node represents the coordinates of the predicted bounding boxes. Each layer has an activation function of ReLU with the exception of the last one, which has a Sigmoid activation function to ensure our output predicted values are in the range 0 to 1.

The second branch in the header is for classification, which is a set of two FClayers with activation of ReLU, two Dropout layers with 0.5 probability of dropout, and a final FC layer with softmax activation. This naive model follows a sparse prediction architecture since two different loss functions, and two different branches are used for detection and classification.

For the optimizer, Adam was chosen for its adaptive learning rate, with an initial learning rate of $1e^{-4}$, number of epochs equal to 40, and batch size of 32. For the loss functions, categorical cross-entropy for the classification task and Mean Squared Error (MSE) for the Bounding Box regression. The training process was done in the local machine and not on the hardware for more GPU capability.

4.3 Pre-Trained Models

Using pre-trained models came out of a necessity to obtain good detection results and overcome the challenges encountered on the previous approach. From this second approach on using the pre-trained models, it was debated in section 3.1.2 whether to use One-Stage Detectors or Two-Stage Detectors. According to the literature, it was decided to focus on One-Stage detectors since the most recent models give a higher speed than Two-Stage detectors and can maintain a relatively good level of accuracy.

For the pre-trained One-Stage models, three options were chosen. YOLOV4 and YOLOV4-

Tiny with CSPDarkNet53 as backbone, and SSD with MobileNet-V2.

As described earlier, Two-Stage Detectors can also be applicable for object detection in real-time if optimized correctly and if needed more accuracy. However, after developing with the One-Stage Detectors for this work and looking at the results on inference in the Jetson Nano in section 5.2, and the need to convert YOLOV4 to YOLOV4-Tiny to run on the device, using Two-Stage Detectors was unnecessary for the task at hand.

4.3.1 YoloV4 and YoloV4-Tiny

Following the structure on section 3.1.2,YOLOV4 was re-trained using the COCO dataset. During this process, several parameters were changed to re-train the model. Some data preprocessing was needed in addition to the methods described in the previous section. This was due to calibrating the model so that it was possible to obtain good results. These changes are explained further.

After training the model, the saved weights and the model configuration were then loaded into the Jetson for inference, and after testing, it was discovered that a smaller model was needed in order to obtain good inference times. YOLOV4-Tiny presents the same structure as YOLOV4, but having reduced network size, the number of convolutional layers in the CSPDarknet53 backbone are compressed, only 2 detection convolutional layers(layers responsible for generating the anchor boxes for each grid cell) instead of 3, and fewer anchor boxes for each grid cell. For this reason, it presented a good alternative.

Pre-Processing

After the extraction described in the previous section 4.1 for COCO, a script was made to create the ground truth files for the YOLOV4. It requires that the bounding boxes are normalized, divided by the *width* and *height* of the images, and the x_{min} and y_{min} to be transformed to x_{center} and y_{center} respectively. The labels for each object must also be in the range from 0 to N-1 classes. Followed by these are the steps of data preprocessing described in section 3.1.2. The size of the images is also changed to 416x416 (multiples of 32). This preprocessing step was the same for YOLOV4 and YOLOV4-tiny.

Parameters and Training

The same structure described in section 3.1.2 was used in this phase, except for some parameters of the network that needed to be changed for the training process with fewer classes. The training process was done in the local machine and not on the hardware for more GPU capability. The list of parameters changed are the following:

- **Number of batches**: The number of batches was changed to 64, having the maximum number of batches equal to 2000 per class.
- **Steps**: The number of steps was changed to 80% of the maximum batch size. The number of steps is the number of iterations at which the scales will be applied to the learning rate, and consequently, the rate at which the weights are updated.
- **Initial learning rate**: was changed to $1e^{-3}$, and will decrease over training.

- **Filters**: The number of filters of the convolution layers before each convolutional detection layer, to $(5 + C) * B$, with C equal to the number of classes,and B the number of predicted boxes for each cell in the grid cell. In this case a total of 30 filters, $C = 5$ and $B = 3$.

These parameters were changed so that the model can be optimized to train for the dataset chosen. The number of batches determined the samples that the network received before updating the weights, and the maximum of batches was reduced, so it doesn't cause bad generalization. The steps and learning were also reduced so that the model could have better convergence to the local minimum. Finally, the number of filters was reduced to give the model fewer parameters to learn since it was trained for only five classes.

4.3.2 SSD-MobileNet-V2

The SSD-MobileNet-V2 model was a late addition in the choices of detection models that took place after the difficulties presented in the deployment of models to the Jetson Nano. This model served as a comparison model to test the inference on hardware and, therefore, was not trained or tested for vehicle detection. However, it could still be used for this purpose since it has weights pre-trained for all COCO classes.

The choice for this model was based on the state of the art research, which indicates it is an optimal choice for detection on mobile or edge hardware, as referred to in section 2.3. The family of MobileNets models has good results in edge devices, and therefore this was the first choice. This model was also available in the Nvidia repository, which made it possible to skip all the configuration processes that take place when running detection models on edge hardware.

4.3.3 Hardware Configuration and Deployment

This section refers to the initial configuration process that took place at the beginning of the development work. The initial configuration on the hardware was done with the JetPack Software Development Kit (SDK) from Nvidia, which included the installation of all the packages necessary to run the inference models. For the YOLOV4 models, extra steps were required to overcome some compatibility errors that weren't allowing for the detection to work. This was surpassed by loading the configurations on the local machine to the hardware and the model's weights. When it comes to deployment on the Jetson Nano, other alternatives could be applied, such as a conversion of the models with TensorRT, an SDK available on Jetson Nano, for high-performance deep learning inference, since it includes an inference optimizer that delivers low latency and high throughput for deep learning inference applications. This will be taken into consideration for future models to be deployed.

This page is intentionally left blank.

Chapter 5

Results

In this section, we present the results from the development process section and the results on the hardware. We are starting with the training results from the models and their validation results on the dataset, proceeding to show the models' detection on Jetson Nano. To conclude, an analysis of the results were done, as well as na overall conclusion about the results and a comparison between models.

5.1 Training and Validation Results

This section presents the training and validation results both for the created model and for You Only Look Once V4 (YOLOV4) and YOLOV4-Tiny. For the last two, it also presents an evaluation on the Common Objects in Context (COCO) dataset, where it is possible to compare both models.

5.1.1 Created Model

For the created model, as explained in section 4.2, only the results for the training process will be presented because of the difficulties experienced during the development process described before. In the figures presented next, the Training and Validation losses will be presented for the classification, regression, and overall loss.

In Figure 5.1, the loss for Bounding Box regression is presented. From it, it is possible to see the model behaved correctly for the localization task since the validation loss decreased and stabilized.

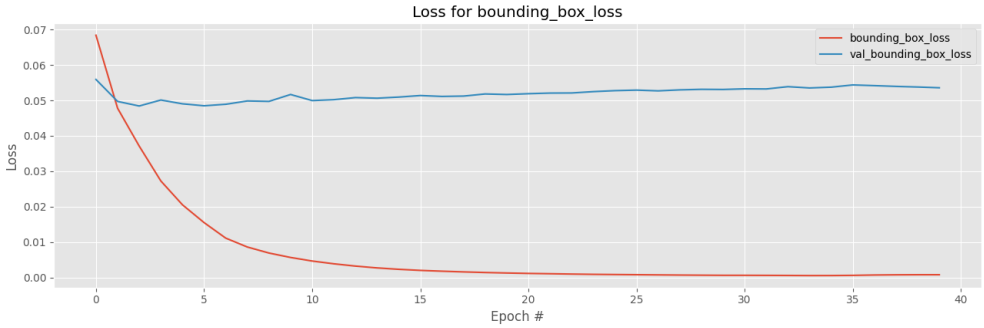

Figure 5.1: Created Model Bounding Box Regression Loss.

In Figure 5.2, the loss for the classification task is presented. It is possible to observe that the model overfitted the training data and performed badly on the validation set since the validation loss increased for each epoch.

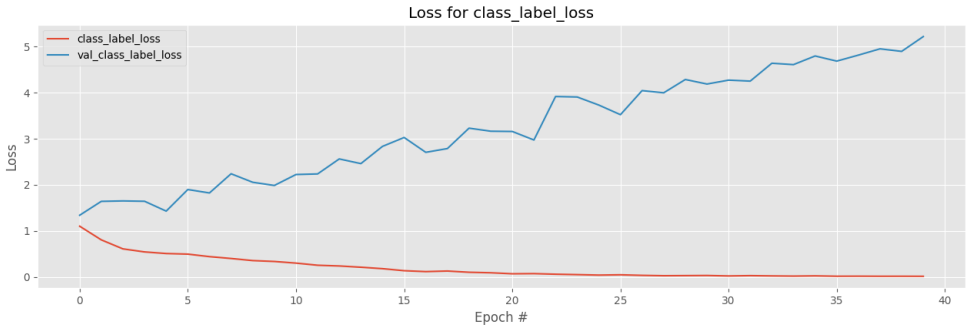

Figure 5.2: Created Model Classification Loss.

In Figure 5.3, the total loss for the created model is presented. From it, it is possible to conclude that the model's overall performance was not good and performed badly on the validation set.

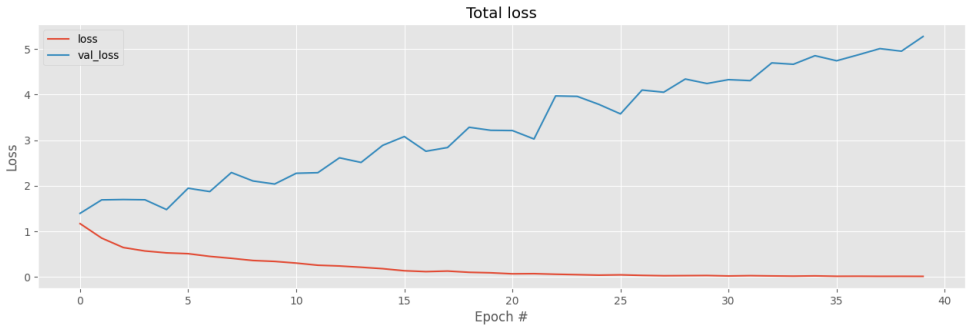

Figure 5.3: Created Model Total Loss.

5.1.2 YOLOV4 and YOLOV4-Tiny

For YOLOV4, the results for the training loss can be visualized in figure 5.4. After some iterations, the loss started to stabilize, and training was stopped, saving the weights. The Mean Average Precision (mAP) decreased a bit as well in comparison to regular YOLOV4, and in this figure was calculated on the validation set and not the testing set on which we'll be guiding the results.

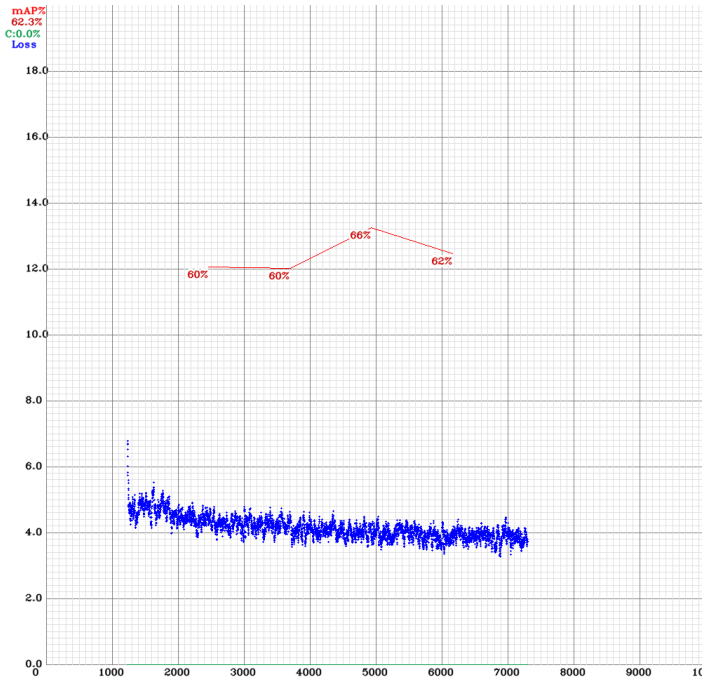

Figure 5.4: YoloV4 Training Loss and mAP.

In figure 5.5 is the results for the training loss for YOLOV4-Tiny. It is possible to see an initial decrease for the training loss and an increase for the validation with MAP. It stabilizes after 4000 iterations, so training was stopped, and the weights were saved.

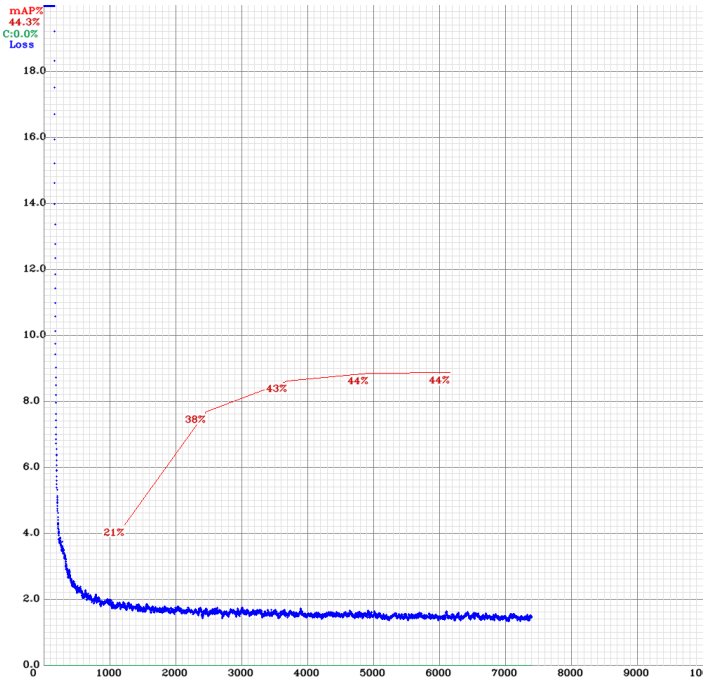

Figure 5.5: YoloV4-tiny Training Loss and mAP.

In the COCO dataset, the subset of testing images can be used to extract the results of the models and compare their performances using the metrics described before in section 3.3. To remember that the evaluation is made on on the COCO dataset, and therefore values of mAP are subject to the Intersection Over Union (IOU) threshold.

For the results on the test set, the following table 5.1 gives a summary of the values obtained for average IOU, precision, recall, and F1-score, for a confidence threshold of 25%. The average IOU represents the average percentage of times the overlap area exceeds the IOU threshold of 50%. This data was obtained following the calculations presented in the section 3.3.

Model	**Precision**	**Recall**	**F1 Score**	**Average IoU**
YoloV4	55%	55%	55%	40.8%
YoloV4-Tiny	49%	37%	42%	36.56%

Table 5.1: Models overall precision and recall values for a confidence threshold of 25%.

After the obtained values in the previous table, we were able to calculate the Average Precision (AP) and mAP for each class for both models. The results are shown in table 5.2.

Class	**YoloV4 Average Precision**	**YoloV4-Tiny Average Precision**
Bycicle	37.51%	27.56%
Car	61.52%	41.37%
Motorbike	58.35%	45%
Bus	75.50%	53.02%
Truck	49.70%	29.51%
mAP@50	56.5%	39.36%

Table 5.2: AP for each class and mAP for both models, for an IOU threshold = 50%.

5.2 Inference on Jetson

To test the inference on Jetson, a video in Ubiwhere's parking lot was taken from the place where the SmartLampost is placed since it was not possible to test on-site. This way, it is possible to simulate as if it was detecting in real-time on the post. The following figures are the results of the detection video for each of the models.

For YOLOV4, figure 5.6 shows a the predictions made in the the edge hardware, with correct predictions.

Figure 5.6: YOLOV4 detection on Jetson.

For YOLOV4-Tiny, the figure 5.7 shows the inference on Jetson, also a correct prediction, with slight less certainty on the most distant vehicle.

Figure 5.7: YOLOV4-Tiny detection on jetson.

For SSD-MobileNetV2, figure 5.8 shows the inference on the hardware. It has good values in terms of performance but struggles due to the distance of the objects and presents an incorrect prediction.

Figure 5.8: SSD-MobileNetV2 detection on Jetson.

The following table 5.3 shows the final comparison between the model's inference on the hardware. According to the evaluation in section 3.3.4, the values were extracted from running the models on the device. To note that Inference Time (IT) is calculated on a single image. The results will be discussed in the following section.

Model	**Average FPS**	**Inference Time**	**GPU and CPU (resource consumption)**
YoloV4	0.5	5494.381ms	High
YoloV4-Tiny	13,2	963.532ms	Low
SSD-MobileNetV2	22	724.453ms	Low

Table 5.3: Comparison between model's inference on Jetson Nano.

5.3 Results Analysis

Since the initial phase of this internship, one of the challenges associated with this approach was the high degree of complexity faced. Consequently, the lack of ideal results in the section 4.2 created a tremendous challenge in terms of complexity and planning, and both implementation and validation phases were directly influenced.

From the results presented, it is possible to draw conclusions about the models and the topic of the trade-off between speed/accuracy. By looking at the training results from the created model in section 5.1.1, we draw the conclusion that the model needed several optimization processes and techniques before becoming a viable option for inference on the Jetson hardware, and as mentioned before in 1.3, the task of optimizing a Convolutional Neural Network (CNN) and fine-tune its parameters can be very complex, as there would not be enough time for this project.

For the training and testing of both YOLOV4 and YOLOV4-Tiny, its possible to con-

clude that YOLOV4 presented better results for mAP and average IOU than YOLOV4-Tiny when tested on the local machine, both with good performances in terms of Frames per Second (FPS). When looking at their performance in the hardware in section 5.2, and comparing both to a very fast but inaccurate Single Shot MultiBox Detector (SSD)-MobileNetV2, we can conclude on the importance of trading speed and accuracy when designing models for edge devices with limited GPU and memory capabilities. The very deep models with more learning parameters and network size are more accurate. Still, speed is essential on deployed models since a minimum required fps are needed in order to obtain some real-time goal.

By looking at these results, it was concluded that the model that it is a better fit to be running on the edge hardware is YOLOV4-Tiny, for its good accuracy and performance. Both YOLOV4 and YOLOV4-Tiny presented good results, however, YOLOV4-Tiny had better values for inference on the Jetson Nano.

This page is intentionally left blank.

Chapter 6

Conclusions and Future Work

This chapter considers the main conclusions related to the work that was developed and the different approaches that were experimented with. It will also be mentioned as contributions that resulted from the work and discuss the future work.

In an initial phase of this project, the learning task of the basic Deep Learning (DL) concepts necessary for this work took place. The concepts of Image Classification and Object Detection, specifically about the Deep Neural Network (DNN) used for these tasks, were studied. It was studied the fundamentals of Convolutional Neural Network (CNN), how to build and train them, the different optimization strategies, and how they learn the features and learning about the tools that were used. The project revealed a very investigative component had to be taken, so a study of the state of the art was made, where it studied the most recent and different models and devices used for this project.

Following the approaches defined, it started by creating a simple model that would be improved over time for the task at hand. While developing this model, some challenges on how to improve the model developed as well as time constraints were encountered during this process. After that, it was decided to make use of transfer learning and use pre-trained models that were studied previously had optimized architectures, such as You Only Look Once V4 (YOLOV4) and subsequently YOLOV4 for the need of more performance on hardware.

Some other challenges were overcome, such as the correct data extraction from the large dataset Common Objects in Context (COCO) and the learning curve that needed to be crossed when first enlisting for this internship.

The main goal of this internship project was to implement an object detection model that would be applying real-time vehicle defections on edge hardware. In other words, the goal was to train a DNN capable of accomplishing the task mentioned before with a high confidence score, low consumption of CPU and GPU resources, and high performance on the edge device.

This goal was concluded in a way that a final model capable of detecting vehicles was deployed in the edge hardware able to perform well, but could be improved if more time is taken into improving the model. During this process, different models were tested to be chosen to be running on the hardware. Although one part of the goal was to test the final model on the Smart Lamppost itself, it was not possible for external motives. The alternative chosen was a video recorded from the SmartLamppost placed in Ubiwhere's parking lot, which was important to simulating the video captured as if it were in real-time.

6.1 Contributions

The main contributions resulting from this work were:

- The Background Knowledge and State of the Art in the field of Object Detection, Image Classification and Detection on edge devices.
- Comparison between different models based on accuracy and inference speed.
- At the end of the work, an object detection model was deployed and can be used on the hardware, detecting at 13FPS and low inference time.
- For Ubiwhere, the configured and ready to be used device for object detection.

6.2 Future Work

Although the chosen model's results were good, it's possible to improve a lot in both detection tasks and performance in the edge hardware. It is possible to build a better model that has more accuracy and more speed. Creating a custom dataset for testing vehicle detection from the Smart Lampposts point of view could improve the models' generic focal perceptions and improve defections for objects at different distances.

The automatic capture and processing of information about the vehicles detected, such as the count, type of vehicle, empty parking spots in a parking lot, and other specifics, to be used by Ubiwhere Urban Platform, would be a good improvement.

References

[1] Cifar10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[2] Coco dataset. https://cocodataset.org/#home.

[3] Cocoapi. https://github.com/cocodataset/cocoapi.

[4] Comparison between frameworks. https://towardsdatascience.com/pytorch-vs-tensorflow-in-2020-fe237862fae1.

[5] Comparison between frameworks. https://towardsdatascience.com/which-deep-learning-framework-is-the-best-eb51431c39a/.

[6] Comparison between frameworks. https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b.

[7] Imagenet challenge. http://www.image-net.org/challenges/LSVRC/.

[8] Imagenet dataset. http://image-net.org/index.

[9] Keras api. https://keras.io/about/.

[10] Nvidia jetson nano. https://developer.nvidia.com/buy-jetson.

[11] Nvidia jetson nano vs google coral vs intel ncs: a comparison. https://towardsdatascience.com/nvidia-jetson-nano-vs-google-coral-vs-intel-ncs-a-comparison-9f950ee88f0d.

[12] Pytorch. https://pytorch.org/.

[13] Tensorflow. https://www.tensorflow.org/.

[14] Urban platform. https://urbanplatform.city.

[15] Luis Rodrigo Barba Guamán, José Naranjo, and Anthony Ortiz. Deep learning framework for vehicle and pedestrian detection in rural roads on an embedded gpu. *Electronics*, 9:589, 03 2020.

[16] C. M. Bautista, C. A. Dy, M. I. Mañalac, R. A. Orbe, and M. Cordel. Convolutional neural network for vehicle detection in low resolution traffic videos. In *2016 IEEE Region 10 Symposium (TENSYMP)*, pages 277–281, 2016.

[17] Karsten Behrendt. Boxy vehicle detection in large images. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, Oct 2019.

[18] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[19] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-yuan Liao. Yolov4: Optimal speed and accuracy of object detection, 04 2020.

[20] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[21] Dami Choi, Christopher J Shallue, Zachary Nado, Jaehoon Lee, Chris J Maddison, and George E Dahl. On empirical comparisons of optimizers for deep learning. *arXiv preprint arXiv:1910.05446*, 2019.

[22] Dan C Cireşan, Ueli Meier, Jonathan Masci, Luca M Gambardella, and Jürgen Schmidhuber. High-performance neural networks for visual object classification. *arXiv preprint arXiv:1102.0183*, 2011.

[23] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. IEEE, 2005.

[24] Z. Dong, Y. Wu, M. Pei, and Y. Jia. Vehicle type classification using a semisupervised convolutional neural network. *IEEE Transactions on Intelligent Transportation Systems*, 16(4):2247–2256, 2015.

[25] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[26] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[27] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[28] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, 2016.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactions on pattern analysis and machine intelligence*, 37(9):1904–1916, 2015.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[32] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[33] Jiayuan Huang and Robert L Nowack. Machine learning using u-net convolutional neural networks for the imaging of sparse seismic data. *Pure and Applied Geophysics*, pages 1–16, 2020.

[34] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7310–7311, 2017.

[35] Mohamed Ibrahim, James Haworth, and T. Cheng. Understanding cities with machine eyes: A review of deep computer vision in urban analytics. *Cities*, 96, 11 2019.

[36] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[37] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. A survey of deep learning-based object detection. *IEEE Access*, 7:128837–128868, 2019.

[38] Diederik P Kingma and J Adam Ba. A method for stochastic optimization. arxiv 2014. *arXiv preprint arXiv:1412.6980*, 434, 2019.

[39] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*, Sydney, Australia, 2013.

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[41] Jan Kukačka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*, 2017.

[42] R. Laroca, E. Severo, L. A. Zanlorensi, L. S. Oliveira, G. R. Gonçalves, W. R. Schwartz, and D. Menotti. A robust real-time automatic license plate recognition based on the yolo detector. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, 2018.

[43] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[44] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[45] Zhengang Li, Geng Yuan, Wei Niu, Yanyu Li, Pu Zhao, Yuxuan Cai, Xuan Shen, Zheng Zhan, Zhenglun Kong, Qing Jin, et al. 6.7 ms on mobile with over 78% imagenet accuracy: Unified network pruning and architecture search for beyond real-time mobile acceleration. *arXiv preprint arXiv:2012.00596*, 2020.

[46] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[47] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8759–8768, 2018.

[48] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[49] London and Zafeirios Fountas. Imperial college spiking neural networks for human-like avatar control in a simulated environment. 01 2021.

[50] David G Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

[51] Kien Nguyen, Clinton Fookes, Arun Ross, and Sridha Sridharan. Iris recognition with off-the-shelf cnn features: A deep learning perspective. *IEEE Access*, PP:1–1, 12 2017.

[52] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1717–1724, 2014.

[53] Niall O'Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco Hernandez, Lenka Krpalkova, Daniel Riordan, and Joseph Walsh. Deep learning vs. traditional computer vision. In *Science and Information Conference*, pages 128–144. Springer, 2019.

[54] K. K. Pal and K. S. Sudeep. Preprocessing for image classification by convolutional neural networks. In *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pages 1778–1781, 2016.

[55] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

[56] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[57] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[58] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[59] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*, pages 92–101. Springer, 2010.

[60] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[61] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

[62] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[63] Petru Soviany and Radu Tudor Ionescu. Optimizing the trade-off between single-stage and two-stage object detectors using image difficulty prediction. *CoRR*, abs/1803.08707, 2018.

[64] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[65] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[66] Mingxing Tan, Ruoming Pang, and Quoc V Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10781–10790, 2020.

[67] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001.

[68] Chien-Yao Wang, Hong-Yuan Mark Liao, Yueh-Hua Wu, Ping-Yang Chen, Jun-Wei Hsieh, and I-Hau Yeh. Cspnet: A new backbone that can enhance learning capability of cnn. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 390–391, 2020.

[69] Alexander Womg, Mohammad Javad Shafiee, Francis Li, and Brendan Chwyl. Tiny ssd: A tiny single-shot detection deep convolutional neural network for real-time embedded object detection. In *2018 15th Conference on Computer and Robot Vision (CRV)*, pages 95–101. IEEE, 2018.

[70] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[71] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6023–6032, 2019.

[72] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object detection in 20 years: A survey. *arXiv preprint arXiv:1905.05055*, 2019.