Universidade de Aveiro Departamento de Engenharia Mecânica
2018

Tomás Sampaio de Freitas Freixo Osório

Deteção de Objectos para Carros e Pedestres Através de Deep Learning

Object Detection for Cars and Pedestrians Using Deep Learning

DOCUMENTO PROVISÓRIO

Universidade de Aveiro Departamento de Engenharia Mecânica
2018

Tomás Sampaio de Freitas Freixo Osório

Deteção de Objectos para Carros e Pedestres Através de Deep Learning

Object Detection for Cars and Pedestrians Using Deep Learning

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestrado em Engenharia Mecânica, realizada sob orientação científica de Miguel Armando Riem de Oliveira, Professor Auxiliar do Departamento de Engenharia Mecânica da Universidade de Aveiro e de Vítor Manuel Ferreira dos Santos, Professor Associado do Departamento de Engenharia Mecânica da Universidade de Aveiro.

DOCUMENTO PROVISÓRIO

O júri / The jury

Presidente / President

Prof. Doutora Ana Maria Perfeito Tomé
Professora Associada da Universidade de Aveiro

Prof. Doutora Margarida Isabel Cabrita Marques Coelho
Professora Auxiliar da Universidade de Aveiro

Vogais / Committee

Prof. Doutor Miguel Armando Riem de Oliveira
Professor Auxiliar da Universidade de Aveiro (orientador)

Prof. Doutor Vítor Manuel Ferreira dos Santos
Professor Associado da Universidade de Aveiro (co-orientador)

Agradecimentos / Acknowledgements

Gostaria de agradecer todo o apoio dado pela minha família e colegas. Agradeço ao Professor Vitor Santos, e particularmente ao Professor Miguel Riem Oliveira por todo o apoio e tempo disponibilizado e por me terem dado a oportunidade de realizar a Dissertação de Mestrado na área de *machine learning*. Também gostaria de agradecer ao Eurico Pedrosa por todo o tempo disponibilizado a configurar o servidor do IRIS Lab, e claro ao IRIS Lab por ter nos emprestado o seu servidor.

Palavras-chave Inteligência Artificial; Machine Learning; Deep Learning; Convolutional Neural Networks, CNN, Detecção de Objectos, Carros Autonomos

Resumo Os carros autónomos são o futuro do sector de transportes, mas a deteção de objetos nas estradas é ainda um enorme desafio e encontra-se em permanente desenvolvimento. O nosso objetivo foi criar e desenvolver um detetor de objetos para carros autonómos que consiga localizar carros e pedestres. Este detetor deve conseguir adaptar-se rapidamente às mudanças de cenário, pelo que tem de processar um elevado número de imagens por segundo e identificar os objetos com uma elevada precisão e sensibilidade. Para tal, recorremos ao *deep learning*, uma técnica de *machine learning* que nos últimos anos tem obtido grande sucesso em diversas áreas, de entre as quais a visão por computador. Com o detetor desenvolvido conseguimos atingir 46 imagens processadas por segundo, uma precisão média de 43,2% e 43,9% de sensibilidade media, pelo que consideramos que o modelo desenvolvido apresenta potencial e é competitivo. Os resultados obtidos indicam que o detetor de objetos desenvolvido poderá vir a ser implementado no ATLAS-CAR 2, o carro autônomo desenvolvido pelo Departamento de Engenharia Mecânica da Universidade de Aveiro.

Keywords Artificial Intelligence; Machine Learning; Deep Learning; Convolutional Neural Networks, CNN, Object Detection, Self-Driving Cars

Abstract Self-driving cars will be the transportation sector future, but nowadays there still exists multiple challenges to solve, such as detecting objects in roads, and further research on multiple fields is needed. The aim of our work was to create an object detector for self-driving cars that detects cars and pedestrians. This object detector must be able to perform at high frame rates and achieve high precisions and recalls, to answer to the fast scenario changes. We used a machine learning technique called *deep learning*, which in the last years was able to achieve state-of-the-art in multiples fields, including computer vision. Our best result was able to perform at 46 frames per second with a mean average precision of 43.2% and a mean average recall of 43.9%, a competitive performance. Our results indicate that our object detector could be implemented in ATLASCAR 2, a self-driving car developed by the Department of Mechanical Engineer of the University of Aveiro.

Contents

List of Tables

List of Figures

Acronyms

AI Artificial Intelligence

ML Machine Learning

IoU Intersection over Union

NMS Non-Maximum Suppression

RoI Region of Interest

YOLO You Only Look Once

SSD Single Shot Detection

AP Average Precision

mAP mean Average Precision

mAR mean Average Recall

FPS Frame per Second

GT Ground Truth

L1 Mean Absolute Error

softmax normalized exponential function

DPM Deformable Part Model

RPN Region Proposal Network

TDM Top-Down Modulation

FCN Fully Convolution Network

SVM Support Vector Machine

AdaBoost Adaptive Boost

SLAM Simultaneous Localization and Mapping

TP True Positive

TN True Negative

FP False Positive

FN False Negative

GPU Graphics Processing Unit

CPU Central Processing Unit

GB Gigabyte

SGD Stochastic Gradient Descent

GT Ground Truth

RAM Random-Access Memory

R-CNN Regions with Convolutional Neural Networks features

CNN Convolutional Neural Network

SVM Support Vector Machine

RPN Region Proposal Network

SGD Stochastic Gradient Descent

Mb Megabyte

ANN Artificial Neural Network

DNN Deep Neural Network

FCNN Fully Connected Neural Network

ReLU Rectified Linear Unit

MSE Mean Square Error

MAE Mean Absolute Error

MSLE Mean Squared Logarithmic Error

Chapter 1

Introduction

Technology has always developed alongside with society [6], shaping it in different ways. Currently, we are entering in the Artificial Intelligence (AI) revolution, which could impact society and firms in different ways [69], as for example the transport sector.

According to the World Health Organization (WHO) road traffic accidents are a leading cause of preventable death. Over 1.2 million people die each year on the world's roads, being in the top ten causes of death among people aged between 15-29 years [112]. Human error has been a big factor in the causes of road accidents and could be minimized using vehicle control systems [96].

The aim of this thesis is to explore an AI application to prevent human error-related road accidents. We will use AI to perform object detection (namely cars and pedestrians) on images of real-world scenarios.

1.1 Vehicle Automation

According to Society of Automotive Engineers (SAE) there are six levels of driving automation on a car [53]. If the human driver performs a part of the car driving task the level of drive automation is 2 or lower, on the other hand, if the autonomous driving systems perform the total control of the vehicle the level of car automation is 3 or above. These levels differ if the next tasks are performed by a Human Driver and/or a System:

- Execution of Steering an Acceleration/Deceleration (level 2 and above);
- Monitoring of Driving Environment (level 3 and above);
- Fallback Performance of Dynamic Driving Task (level 5 and above);
- System Capability on driving in multiple environments (level 5).

The **Dynamic driving task** consists in the operational aspects of the driving task (steering, braking, accelerating, monitoring the vehicle and the roadway) and in the tactical aspects of driving task (responding to events, determining when to change lanes, turn, use signals, etc.), but not in the strategic task (determining destinations and waypoints). **Driving mode** is a type of driving scenario in which are performed dynamic driving task (e.g., expressway merging, high-speed cruising, low-speed traffic jam, closed-campus operations, etc.).

The automation of cars will change the way we live [124], reducing the cost of travel, increasing travel safety, allowing to optimize the route, etc. Increasing the quality of living in the cities. To achieve the highest levels of automation is necessary the usage of learning techniques [5], which allows completing multiple tasks without the need to hard-code all the rules. This suggest that AI systems need the ability to gain their own knowledge, by extracting patterns from raw data [38].

1.2 Artificial Intelligence

The field of AI attempts to understand intelligent entities and create them. Intelligence entities have a huge impact on our everyday lives and will affect the course of civilization [94].

In its early days, the AI was used to rapidly solve problems that are intellectually difficult for human beings, which could be described by mathematical rules and are relatively straightforward for a computer. However, the true challenge for AI are the tasks that are easy for the humans to perform but hard to describe, such as language, vision, recognizing spoken words, etc.

Several AI projects have tried to solve problems through hard-coded knowledge, which is known as a **knowledge base systems**, but none of these projects led to a major success. The difficulties faced by systems just relying on hard-coded knowledge led the scientific community to realize that there is the need of implementing systems that are able of acquire its own knowledge through patterns from raw data. This capability is known as *machine learning*.

1.2.1 Machine learning

Machine Learning (ML) gives computers the ability to learn without being explicitly programmed [98]. This transformative technology allows computers to make decisions according to the provided data, performing very well for problems that either are too complex or does not have a known algorithm to solve it [34].

The first ML techniques depended heavily on the representation of the provided data, these pieces of information are known as a feature. ML techniques, such as logistic regression, could correlate different features, however, cannot alter them. Many tasks were solved by designing the right set of features for the task but is difficult to know which should be extracted.

With the development of ML techniques, they were not only able to correlate different features but also extracting their own features. This is known as *representation learning*, resulting in higher performances comparing to *knowledge base systems*. However these techniques generally did not perform well in real-world applications, due to the high level of variety that occurs in these scenarios. Deep Learning solved this problem, allowing the computer to build complex concepts out of simpler concepts, being more robust to noise [38].

1.2.2 Deep Learning

The easiest way to explain Deep Learning is with a **feed-forward deep network** or Multilayer Perceptron (MLP) model. First, the input layer transmits the input data to

the hidden layer. The unit in the hidden layer accumulates and process the weighted inputs before sending their outputs to the output units. This structure is trained to learn by repeated exposure to examples until produces the correct output [97]. In our next chapter, these concepts will be discussed.

Despite the fact that Deep Learning has been characterized as a re-branding of Artificial Neural Network (ANN) [37], various deep learning architectures have been applied to fields like computer vision, speech recognition, natural language processing, as in many other fields and they have been producing the state-of-the-art results in different tasks [79].

Using deep learning techniques, in 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [56], Hinton and his student Krizhevsky were able to surpass all other competitors in image classification and achieved top-5 error 15% vs 26.2% of the conventional methods [58]. From that time onwards deep learning is being applied to many areas of computer vision, achieving great results. One of the field that had great success using deep learning techniques was object detection [127].

Object Detection

Object detection consists of localizing and classifying an object on an image. Typically the localization is represented by a bounding box [93], [65], as shown in Figure 1.1.

Initially the object detection was based on hand-crafted features, but because of the complex background, noise disturbance, occlusion, illumination changes and other circumstances, it was very hard to create a robust system with a good generalization [127].

Deep learning was successfully used in object detection by Ross Girshick in 2013, which proposed the R-CNN [36]. This method was a milestone and further methods came after this one as well other approaches that will be detailed in State-of-the-Art in Object Detection.

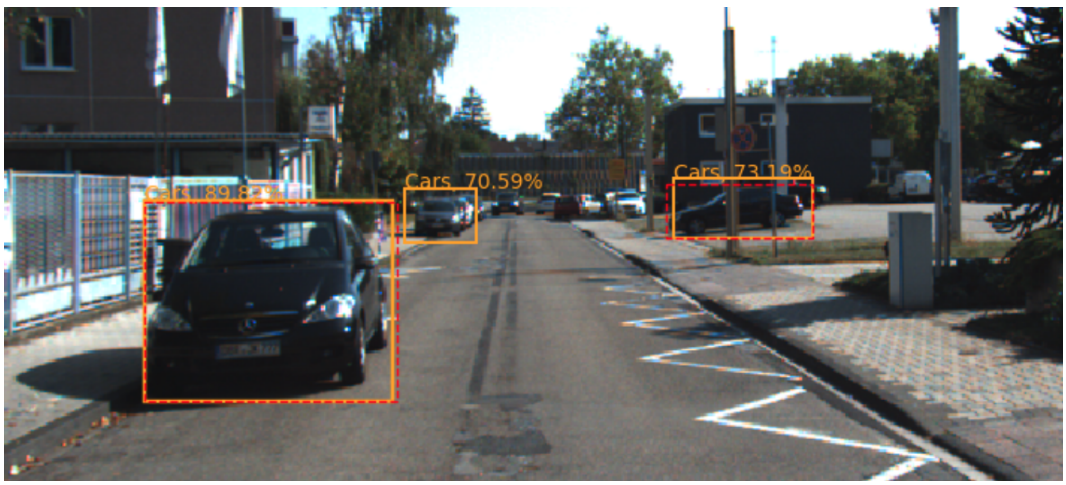


Figure 1.1: Object detector predicting the position of cars in a real-world environment, in red is represented the objects ground truth and in orange the predictions.

1.3 ATLAS Project

ATLAS project[1] started in 2002/2003 in the Laboratory of Automation and Robotics on the Department of Mechanical Engineering of the University of Aveiro. ATLAS focused its resources into autonomous driving subjects, which first started developing small-scale prototypes to enrol in the Portuguese Robotics Open (PRO). After successfully winning several awards,in 2010 the project team begun developing a real-scale vehicle ATLASCAR, containing several cutting-edge sensors. With this car they accomplished several breakthroughs at the time, such as localizing zebra crossings, pedestrians, or for example following a person within a safe distance [78].

Currently, this car already has installed several sensors such as cameras, lasers, GPS, among others [16].

The aim of the present work is to create an object detector that could be later on implement in ATLASCAR 2.

[1] http://atlas.web.ua.pt/

Chapter 2

Background

This chapter explains basic concepts about ANNs, and at the end of this section, it will be complemented it with some concepts that are more focused on object detection problems.

2.1 Artificial Neural Networks

ANNs have been around since 1943, were neurophysiologist Warren McCulloch and mathematician Walter Pitts introduced a simplified computational model of how biological neurons might work in animals [70]. However, this field is no longer inspired by biological neurons due to the difficulty to understand how the biological brain works.

ANNs had its ups and downs [71] due to exaggerated claims regarding their biological plausibility, but lately have been under a lot of attention again due to their powerful abilities in image processing, speech recognition, natural language processing, etc. The increase of computational capability and the large amount of data available nowadays allowed the ANNs to out-perform other frameworks in these fields [10].

In the next subsection, it will be explained in detail how feed-forward Neural Networks work and also some variations that exist. These concepts were based on Bishops [4] and Buduma [7] books.

2.1.1 Feed-Forward Neural Networks

ANN are very similar to linear models used for regression and classification problems, using an identical basic function. ANNs associate a linear combination of inputs allowing that each basis function be a non-linear function since the coefficients are adaptive parameters. Equation 2.1 constructs M linear combinations of the input variables $x_1, ..., x_D$ [4]:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}, \tag{2.1}$$

where the superscript (1) refers to the corresponding layer, being in this case, the first layer of the network, the $j = 1, ..., M$ indicates the position in that layer. $w_{ji}^{(1)} x_i$ are

the *weights*, $w_{j0}^{(1)}$ the *biases* and a_j are the activations. Then, these activations are transformed using a differentiable non-linear activation function $h(\cdot)$:

$$z_j = h(a_j), \tag{2.2}$$

these are called hidden units. A sigmoidal function is generally used as the non-linear function $h(\cdot)$, such as a logistic sigmoid or hyperbolic tangent function. These values are again linearly combined in order to give output activation, resulting in the following equation:

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}, \tag{2.3}$$

where $k = 1, ..., K$ and K is the total number of outputs. These computations corresponds to the second layer of the network, where the output unit activations use an appropriate activation function to give a set of network outputs y_k, which varies according to the provided data and objective of the network.

For multiple classification problems, each output unit activation is transformed using a logistic sigmoid function according to the following equation:

$$y_k = \sigma(a_k), \tag{2.4}$$

where,

$$\sigma(a) = \frac{1}{1 + \exp(-a)}. \tag{2.5}$$

The layers that just described are called fully-connected layers. These computations will result in a two-layer feed-forward ANN that can be represented as shown by Figure 2.1. Notice that the number of the network's inputs is the same amount as provided during training, however, the number of outputs varies according to the chosen task.

2.1.2 Activations

Activations functions used in ANN are univariate and applied to each element of the input feature providing the non-linearity needed to learn complex distributions [115]. In the next subsections will be explained in further detail the most commonly used activation functions in the current state-of-the-art.

Logistic Sigmoid Function

The logistic sigmoid function is one of the most commonly used activation functions in feed-forward neural networks due to their non-linearity and the computational simplicity. Its smoothness is also helpful to soft limiting non-linearities, which are common in ANN [40]. Logistic sigmoid functions return values vary between 0 and 1, which is commonly needed for the network's output, this function is computed as follows:

$$\sigma(a) = \frac{1}{1 + exp(-a)}, \tag{2.6}$$

Figure 2.2 shows logistic sigmoid function graphical representation.

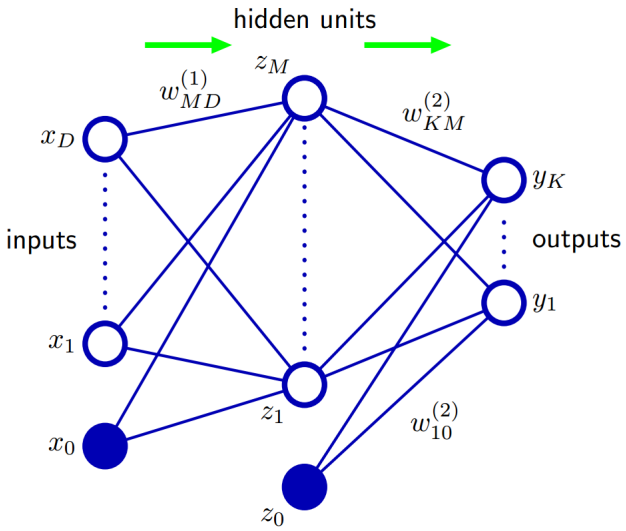


Figure 2.1: Feed-Forward Neural Network with two-layers, where each circle represents a unit (from [4]).

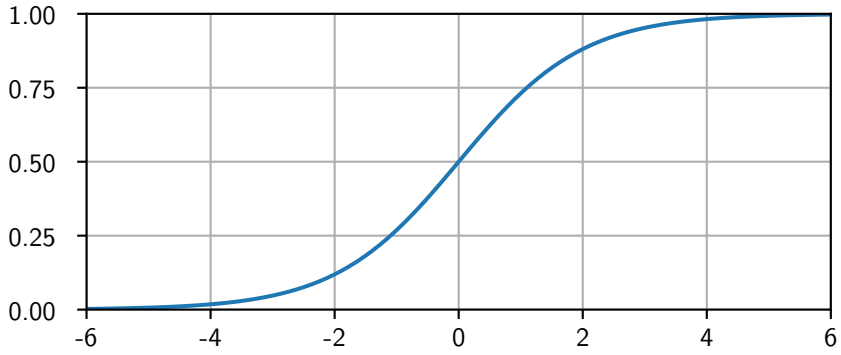


Figure 2.2: Logistic sigmoid function curve.

Hyperbolic Tangent Function

Hyperbolic tangent function, abbreviated as *Tanh*, is a continuous and differentiable sigmoidal function. This function outputs values between -1 and 1, making each layers output more normalized, often helping to obtain faster converges [34]. Figure 2.3 shows the hyperbolic tangent function graphical representation.

Rectified Linear Units Function

ReLUs [74] had been essential to the recent success of Deep Neural Networks (DNNs), allowing the decrease of training time due to their computational simplicity, generally obtaining similar or higher performances comparing to sigmoidal functions [126], [17].

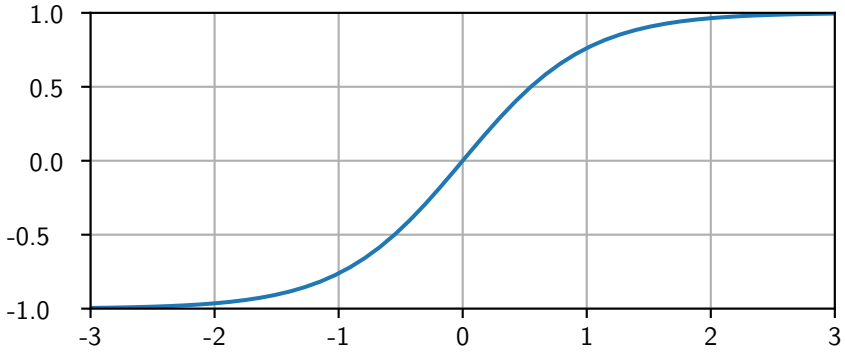


Figure 2.3: Hyperbolic tangent function curve.

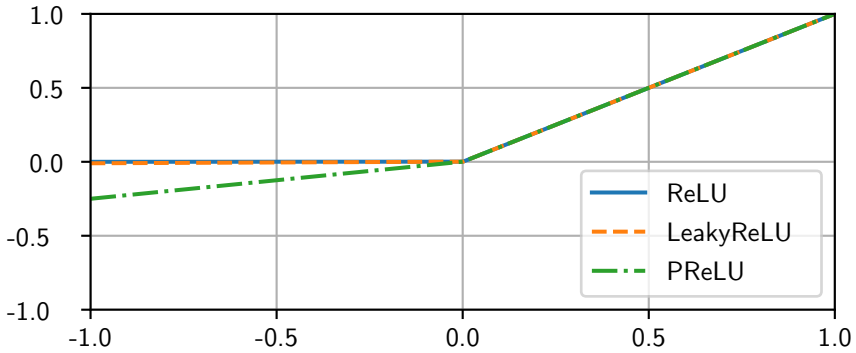


Figure 2.4: ReLUs graphical representation.

The ReLU is computed as follows:

$$f(a) \equiv \begin{cases} a, & \text{if} \quad a > 0, \\ 0 & \text{if} \quad a \leqslant 0, \end{cases} \tag{2.7}$$

After ReLU activation function emerged, various version had appeared trying to improve further its performance. One of the most used is Leaky ReLU [68], which is formulated as follows:

$$f(a) \equiv \begin{cases} a, & \text{if} \quad a > 0, \\ az & \text{if} \quad a \leqslant 0, \end{cases} \tag{2.8}$$

where z is a parameter to adjust the slope, which generally is set to 0.01. This tweak allows having gradient through all its values, unlike the original version, however, does not improve the results substantially [68].

In 2015 another variant emerged, the Parametric Rectified Linear Unit [44], which its formulation is similar to Leaky ReLU but with a higher slope ($z = 0.25$). This tweak allows improving the network's performance at a negligible extra computational cost with little overfitting risk. Figure 2.4 shows the graphical representation of the activation functions described before.

Normalized Exponential Function

The Normalized Exponential Function, most known as Softmax function, allows to reduce extreme-values (outliers) in the data without removing them from the dataset [84]. This function is widely used in classification frameworks as the output activation function because it associates a "probability" for each trained class [4]. Its formulation is expressed as follows:

$$f(a) = \frac{exp(a_k)}{\sum_j exp(a_j)}, \tag{2.9}$$

this formulation satisfies $0 \leqslant f \leqslant 1$ and $\sum_k f = 1$, where k represents the k-th unit and j the index for that vector components.

2.1.3 Loss Function

The Loss function quantifies how close the prediction is from the GT, so it is always dependent on the focused task [81]. This is very important due to the impacts that may occur in the choice of a loss function. *For example, in a system in a medical field, a false negative might lead to a patient not receiving treatment for a serious disease, however, a false positive might lead to additional tests or even unnecessary treatments* [91]. So, we must choose a trade-off that makes sense.

The loss function in ANNs will be used to help find the parameters (*weights* and *biases*) that will minimize the loss incurred from the errors. Generally, this can not be solved in an analytical form, so, commonly an iterative optimization algorithm is used to solve it.

The following subsections we will denote that a given training set comprising a set of input vector $\{x_n\}$, where $n = 1, ..., N$ and the corresponding set of target vectors will be $\{t_n\}$.

Loss function for Regression

Mean Square Error

The Mean Square Error (MSE) is one of the most used loss functions for regression due to its simplicity, however is sensitive to outliers. The MSE is calculated by squaring the error in a prediction and averaging over the entire dataset, as denoted in the following equation:

$$E(w) = \frac{1}{N}\sum_{n=1}^{N}(y(x_n, w) - t_n)^2. \tag{2.10}$$

Mean Absolute Error

The Mean Absolute Error (MAE) consists of simply averaging the absolute error over the number of data points:

$$E(w) = \frac{1}{2N} \sum_{n=1}^{N} \left| y(x_n, w) - t_n \right|. \tag{2.11}$$

Mean Squared Logarithmic Error

For regression is also used Mean Squared Logarithmic Error (MSLE), which is quite similar to MSE but we calculate the logarithm of the target and prediction. The formulation of this loss function is denoted as follows:

$$E(w) = \frac{1}{N} \sum_{n=1}^{N} (\log(y(x_n, w)) - \log(t_n))^2. \tag{2.12}$$

Loss function for Regression: Discussion

The functions described above are frequently used, but each has its advantages and disadvantages. The MSE is widely used, just as MAE, however, when the outputs varies a lot generally is used MSLE. For example, if we have two outputs, one that varies between [0, 10] and the other between [0, 100], MSE and MAE would penalize more the second output than the first, however, MSLE would take into account its values sizes.

Loss function for Classification

The ANNs commonly use for classification problems loss functions that associate a "probability" to its outputs, allowing to know with some degree of certainty which class is most likely to be. In the following subsections its explained how these loss functions are formulated.

Hinge Loss

Hinge loss is generally used for binary classification, mostly when the network must be optimized for a hard classification, for example in detecting frauds, where fraud = 1 and no fraud = 0. This kind of classification is by convention called as a 0-1 classifier. However, when using Hinge loss the data points must be -1 or 1, where this function is computed as follows:

$$E(w) = \frac{1}{N} \sum_{n=1}^{N} max(0, 1 - y(x_n, w) \times t_n). \tag{2.13}$$

Negative Logarithmic Likelihood

Negative Logarithmic Likelihood is frequently in classification problems with multiple classes. This function is mathematically equivalent to what is called the cross-entropy

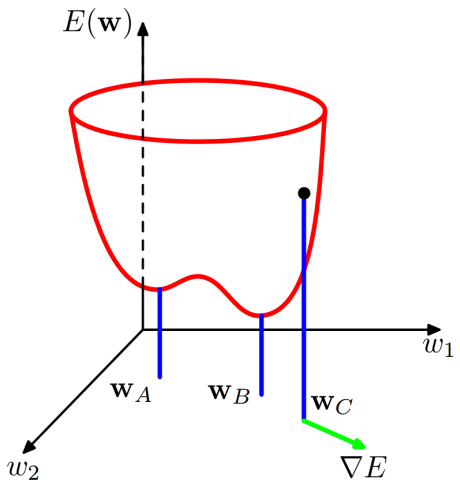


Figure 2.5: Surface representing an error function $E(w)$ over a weight space, where point w_A is a local minimum and w_B is the global minimum. w_C represents the local gradient of the error given by ΔE (from [4]).

between two probability distributions. The formulation of Negative Logarithmic Likelihood is denoted as follows:

$$E(w) = -\sum_{n=1}^{N}\{t_n \ln y_n + (1-t_n)\ln(1-y_n)\}. \tag{2.14}$$

This loss function leads to a faster training and good generalizations [102].

2.1.4 Optimization

After obtaining the error from the loss function $E(w)$, it is needed to find suitable weight and biases (w) to minimize it. To better explain this process, Figure 2.5 contains a geometrical representation of the error function in order to the weights and biases, with an example of two weights.

To optimize a network, it is made a small step in the weight space from w to $w+\delta$ which will change the error function $\delta E \simeq \delta w^T \Delta E(w)$. Notice that the vector $\Delta E(w)$ points in the direction where the error function will increase the most. Since $E(w)$ is a continuous function of w, the smallest value will occur when the weights will make that the gradient of the error function vanishes, so that:

$$\Delta E(w) = 0, \tag{2.15}$$

however, when taken a small step in the opposite direction $-\Delta E(w)$ the error will reduce. When the gradient vanishes these points are called stationary points, which may be minima, maxima or a saddle point.

For a good optimization it is necessary to find a vector w that will get the smallest value in the error function $E(w)$ corresponding to the global minimum. In ANN it is normally not possible to verify if a minimization reached the global minimum or if we will ever find it, so it is necessary to compare several local minima to find a solution. To

find this minimum it would be nearly impossible doing it using an analytical solution so, instead, an iterative approach is used. The techniques that use this approach typically choose some initial value $w^{(0)}$ for the weight vector and move it around the space in a succession of steps; the next equation denotes this process:

$$w^{(\tau+1)} = w^{(\tau)} + \Delta w^{(\tau)}, \tag{2.16}$$

where τ labels the iteration step. Not all techniques update the weight vector $\Delta w^{(\tau)}$ in the same way, however many of them use the gradient information. So, it is required that after each step the value of $\Delta E(w)$ is evaluated at the new weight vector $w^{(\tau+1)}$ [4].

Gradient Descent

Gradient descent uses the gradient information to choose the weight update in (2.16) taking a small step in the direction of the negative gradient; this computed is as follows:

$$w^{(\tau+1)} = w^{(\tau)} - \eta \Delta E(w^{(\tau)}), \tag{2.17}$$

where the parameter $\eta > 0$ is known as the learning rate. This computation happens after each weight update, so that the gradient be re-evaluated for the new weight vector w. After each weight update is also necessary to revaluate the error function ΔE for all the training data. These techniques that use the whole dataset at once are called batch methods, which every step the weight vector moves towards the biggest decrease in the error function. The approach previously described is called gradient descent or steepest descent. This process intuitively may seem optimal, however, turns out to be a poor algorithm.

There are other batch methods which are more efficient than gradient descent, such as conjugate gradients and quasi-Newton methods, which are more robust and faster [82], [29], [75]. These algorithms are built to decrease the loss at each iteration, if they did not already reach a local or global minimum. However, it is necessary to run multiple times in different random starting points to find a sufficiently good minimum and then test its performance on an independent dataset, causing these methods to be highly time-consuming.

In 1998, LeCun et al. [62] were able to develop an online version of gradient descent that was useful to train ANNs on large datasets. For that, the error function is based on a maximum likelihood for a set of independent observations that its terms are summed, one for each data point. This is be computed as follows:

$$E(w) = \sum_{n=1}^{N} E_n(w). \tag{2.18}$$

This method, online gradient descent, most commonly known as Stochastic Gradient Descent (SGD), updates the weight vector one data point at a time, as denoted in the following equation:

$$E(w^{(\tau+1)}) = w^{(\tau)} - \eta \Delta E_n(w^{(\tau)}). \tag{2.19}$$

These updates are repeated through cycling all data. Each iteration, or cycle, could be gathered by choosing random points or by sequential points.

SGD and other online methods deal with data much more efficiently than batch methods, because the online methods do not deal with all data at once. This way, the dataset size will not affect the computational effort of the online methods. Another advantage of using online methods is the **possibility** of escaping from local minima, since a stationary point with respect to the error function for the whole data set will generally not be a stationary point for each data point individually.

Backpropagation

ANN are trained through an iterative process that adjusts the weights in a sequence of step, which is is defined by two main phases. The first phase evaluates the derivatives of the error function with respect to the weight. This process is called backpropagation that uses a local message passing scheme where the information in passed backwards. Then the weights are adjusted accordingly with the computed derivatives. One of the simplest techniques initially considered was [18] which involves gradient descent.

Evaluation of error-function derivatives

To better understand how backpropagation works lets consider an arbitrary feed-forward topology, using an arbitrary differentiable non-linear activation function and a broad class of error functions. This will result in a single layer of a sigmoidal hidden unit, which the cost function will be a sum-of-squared error. The error function can is defined as follows:

$$E(w) = \sum_{n=1}^{N} E_n(w). \tag{2.20}$$

Considering a simple linear model that the outputs y_k are linear combinations of its input variables x_i, which is denoted as follows:

$$y_k = \sum_i w_{ki} x_i. \tag{2.21}$$

With a particular input pattern n and the output of the linear combination y_k, the error function will be:

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2, \tag{2.22}$$

where $y_{nk} = y_k(x_n, w)$. The gradient with respect to a weight (w_{ji}) of this error function is given by the following equation:

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni}, \tag{2.23}$$

this equation can be viewed as a "local" computation, which involves the product of an "error signal" ($y_{nj} - t_{nj}$) associated with the output end of the link (w_{ji}) and the variable x_{ni} associated with the input end of the link.

For more complex systems, for example, a multilayer feed-forward network, each unit from this network computes a weighted sum of its inputs:

$$a_j = \sum_i w_{ji} z_i, \tag{2.24}$$

where z_i is the activation of a unit, or input, which sends a connection to j-th unit and to the weight associated with that connection is w_{ji}. Biases can be included in this sum by introducing an extra input with activation fixed at $+1$, therefore it does not need to be dealt with it explicitly. The sum resulted in (2.24) is transformed by a non-linear activation function $h(.)$ resulting in the activation z_j of the unit j. This is computed as follows:

$$z_j = h(a_j). \tag{2.25}$$

Notice that in (2.24) the sum could be done by one or more variables z_i, also in (2.25) the unit j could be an output.

This process is called *forward propagation*, it consists of passing the data information through the network and then calculate the activations of all of the hidden and output units in the network by successive applying (2.24) and (2.25).

After the *forward propagation* its applied the backpropagation, however first it is necessary to evaluate the derivatives of E_n with respect to a weight w_{ji}. The outputs of the units will depend on the input data n, however, to keep the notation in a cleaner way it is omitted the subscript n from the network variables. Since E_n depends on the weight w_{ji} only via the summed input a_j to unit j it is applied the chain rule for partial derivatives, denoted as follows:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}. \tag{2.26}$$

using the following notation:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j}, \tag{2.27}$$

where the δ's are often referred to as *errors*. Using (2.24) results in:

$$\frac{\partial a_j}{\partial w_j i} = z_i. \tag{2.28}$$

Substituting 2.27 and into 2.26, outcomes the following equation:

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i. \tag{2.29}$$

This equation indicates that the required derivative is obtained by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight (where $z = 1$ in the case of a bias). Notice that this is similar to the simple linear model when it was explained the Feed-Forward Networks. To evaluate the derivatives it is needed to be calculated the value of δ_j for each hidden and output unit in the network, then its applied (2.29), which results in:

$$\delta_k = y_k - t_k. \tag{2.30}$$

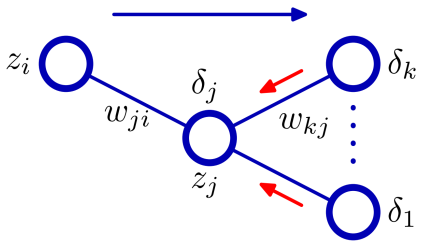


Figure 2.6: Backpropagation for a hidden unit j, where the blue arrow indicates the forward propagation direction and the red arrow denotes the backward propagation error (from [4]).

Now, it is applied the chain rule for partial derivatives in order to evaluate the δ's for hidden units, resulting in:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}. \tag{2.31}$$

Notice that the sum runs over all units k to which unit j sends connections, this is better shown in Figure 2.6.

To formulate the *backpropagation* the definition of δ given by (2.27) its substituted into (2.31) and making use of (2.24) and (2.25), results in:

$$\delta = h'(a_j) \sum_k w_{kj} \delta_k, \tag{2.32}$$

which the value of δ of a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network. Since it is known the value of δ's for the output units, we can evaluate it for all the hidden units by recursively applying (2.32).

Summary of the **backpropagation** [4]:

1. Apply an input vector x_n to the network and forward propagate through the network using (2.24) (2.25) to find the activations of all the hidden and output units.

2. Evaluate the δ_k for all the output units using (2.30).

3. Backpropagate the δ's using (2.32) to obtain δ_j for each hidden unit in the network.

4. Use (2.29) to evaluate the required derivatives.

In batch methods the derivative of the total error E could be obtained through repeating the step above and then summing all up, as it follows:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}, \tag{2.33}$$

where is assumed that all units have the same activation function $h(.)$. However, if we wanted different activations functions for some units, it would simply be needed to keep track of it of which form of $h(.)$ goes with which unit.

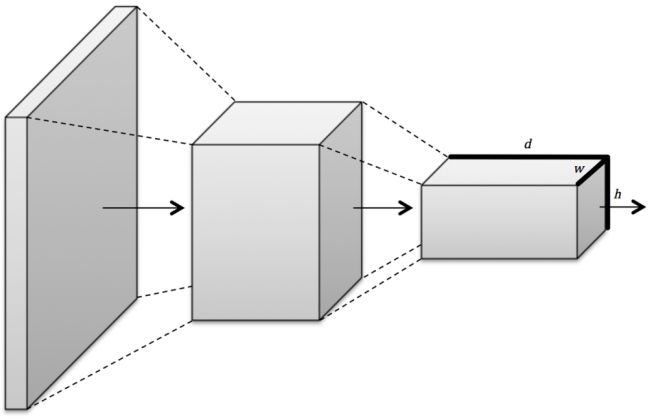


Figure 2.7: Convolutional layer arrange neurons in three dimensions width (w), height (h) and depth (d) (from [7]).

2.1.5 Convolution Neural Networks

If we select a random image and try to identify the containing objects, for a human, that would be easy and extremely natural to do it so. However, for a machine, it is extremely difficult due to the high variance and variety present in an image. Even by using traditional machine learning algorithms would struggle with the given challenge. This happens because the signal-to-noise ratio is too high for any useful learning to occur.

Another attempt made, obtained a trade-off between traditional computer program, where it was defined all the logic, and then the machine learning would do the computational generalizations. For that, the features had to be hand picked and could reach to hundreds or even thousands. This would produce a lower-dimensional representation of the problem and then the machine learning would use these features to make a decision. Since this feature extraction improves the signal-to-noise ratio, if the features were appropriated picked, it would accomplish the task with a higher performance comparing it to other state-of-the-art solutions at the time. This technique is very slow and time-consuming and did not obtain any extraordinary results, so other solutions were attempted.

With the increased power of the computers, DNNs started to emerge. This machine learning method limited the feature selection process since each layer is responsible for learning and building up features to represent the received input data. However, in 1990 Convolutional Neural Networks (CNNs) appeared, which were inspired by how biological vision works arranging the input layers in three dimensions width, height and depth as shown in Figure 2.7. These networks use the "raw" data as its input and rely on backpropagation to appropriately extract features for the first layers. Figure 2.8 shows one of the first Fully Convolution Networks (FCNs) successfully used in image processing. CNNs combine three key features to ensure some degree of shift and distortion invariance, such as (1) local receptive fields, (2) shared weights (or weight replication), and (3) spatial or temporal subsampling [60]:

1. **Local receptive fields:** allows units (or neurons) to extract elementary visual features such as oriented edges, end-points, corners.

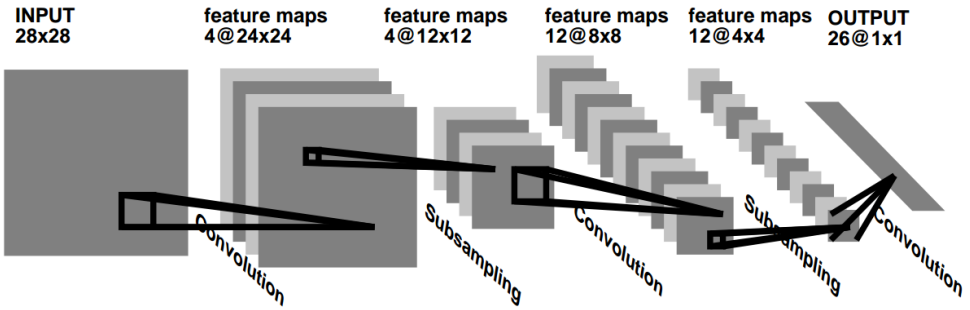


Figure 2.8: Fully Convolutional Neural Network for handwritten digit recognition. (from [60]).

2. **Shared weights:** allows to reduce the number of parameters needed by a large factor, still being able to extract a lot of useful information.

3. **Spatial/Temporal subsampling:** performs a local averaging and a subsampling, reducing the resolution of the feature map, and the sensitivity of the output to shifts and distortions.

Filters and Features Maps

The filters and feature maps are the key concept of convolution layer, in which a filter consists essentially of a feature detector. To better understand how they work, let's consider the Figure 2.9. To detect the vertical and horizontal lines in 2.9 it must be used a feature extractor like the one shown in Figure 2.10. To detect the vertical lines, the feature presented in the top is used, which is slid through the image, and when it matches a vertical line was detected. The results are shown in the matrix presented in the top right corner; in black are marked the detected vertical lines, which is the resulting feature map. For the horizontal lines, the results are accordingly presented in the bottom right side of the Figure 2.10.

The operation described above is a convolution, which consists **essentially** of a filter that is multiplied over the entire area of the image in question. In feed-forward neural networks, "*a unit in a feature map (layer) is activated if the filter contributing to its activity detects an appropriate feature at the corresponding position in the previous layer*" [7].

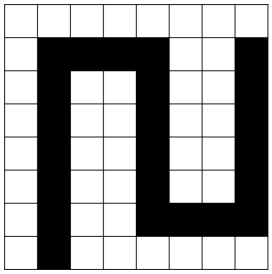


Figure 2.9: Simple back-and-white example image (from [7]).

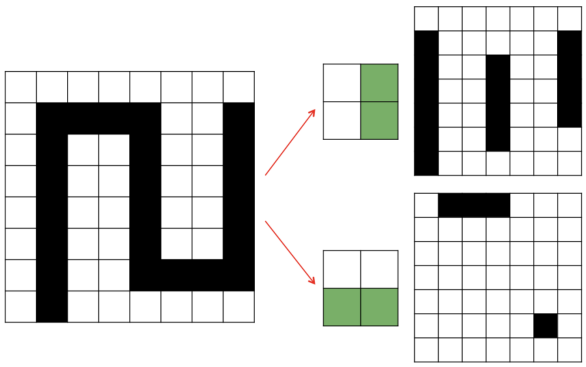


Figure 2.10: Applying filters that detect vertical and horizontal lines on a simple black-and-white example image (from [7]).

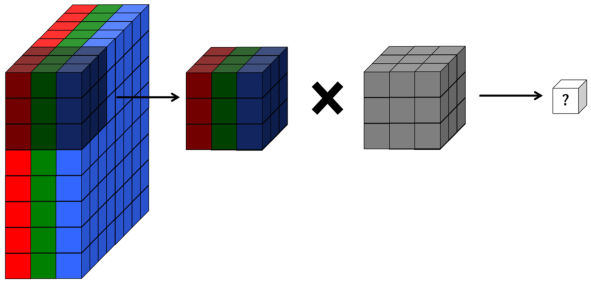


Figure 2.11: Representation of a RGB image as a input volume and applying a volumetric convolutional filter, resulting in a output volume (from [7]).

To express the feature map let's denote the k^{th} feature map in a layer m as m^k and the corresponding filters are defined by the values of its weights w_{ji}^k. If we assume that the units in the feature map have bias w_{j0}^k and that are kept identical for all units in a feature map, we can the express the feature map with the following equation:

$$m_{ji}^k = f(w_{ji}^k x_i + w_{j0}^k), \tag{2.34}$$

despite the equation being simple and concise, it does not describe how the filter work in CNNs. Filters operate on the entire volume of the feature map from a particular layer. If we have an image represented by RGB (Red-Green-Blue) values, that means that we have an input volume that contains three slices. Since feature maps are able to operate over volumes, not just areas, as shown in Figure 2.11, each cell in the input volume is a unit.

Summing up how a convolution layer works:

A convolutional layer consists of a set of filters that converts one volume of values into another volume of values. The depth of each filter corresponds to the depth of the input volume, allowing filters to combine information from all features that have been learned. The depth of the output layer corresponds to the number of filters in that layer because each filter produces its own slice.

Full description of the Convolutional Layer

In a convolutional layer, the first step consists of taking an input volume. So first let's characterize the input volume:

- Its width w_{in};
- Its height h_{in};
- Its depth d_{in};
- Its zero padding p.

This input volume is then processed by k filters, which represents the weights and connections in the CNN. These filters have a number of parameters that define them, which are described as follows:

- Spatial extent is a scalar e, which is the dimensions of the filter: height and width.
- Strides s, represent how many shifts it moves every time the filter is applied.
- Besides weights w, Convolutions also have bias b, that are learned in the same way, which is added to each component of the convolution.

The output volume that results in applying the previous filters can be characterized as follows:

- Its width $w_{\text{out}} = [\frac{w_{\text{in}} - e + 2p}{s}] + 1$.
- Its height $h_{\text{out}} = [\frac{h_{\text{in}} - e + 2p}{s}] + 1$.
- Its depth $d_{\text{out}} = k$.
- Its activation function f is applied to every unit in the output value to determinate its final output value.

The depth slice of the output volume is denoted as m^{th}, where $1 \leq m \leq k$, corresponds to the function f applied to the sum of the m^{th} filter convoluted over the input volume and the bias b^m. This means that per filter we have $d_{\text{in}}e^2$ parameters, which mean that the layer has $kd_{\text{in}}e^2$ parameters and k biases.

In real applications, the filter sizes are usually kept small (3×3 or 5×5), allowing high representations while using a reduced number of parameters, however, is common to use at the first convolutional layer bigger filters (7×7). The filters, generally, also use a stride of 1 to capture all useful information from the feature maps and zero padding to keep output volume, width and height, the same as the input volume.

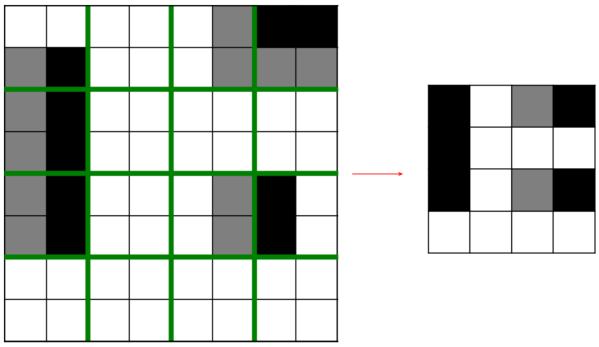


Figure 2.12: Max-pooling applied to a feature map (from [7]).

Max-Pooling Layer

Map-pooling layers are used to reduce the dimensionality of the feature maps and sharpen the local features, these layers generally appear after convolution layers. The idea behind this kind of layer is to break up each feature map into equally size tiles and then create a condensed feature map. To better understand this process we suggest to look into Figure 2.12.

Max-pooling layers have to main parameters, the spatial extent e and the stride s. Generally, when using pooling layers only two major variations are used. The first is the non-overlapping pooling layer with $e = 2$ and $s = 2$, the second is the overlapping pooling layer with $e = 3$ nad $s = 2$. The output dimension of each feature map after applying a pooling layer takes the following form:

- Its width $w_{\text{out}} = [\frac{w_{\text{in}} - e}{s}] + 1$
- Its height $h_{\text{out}} = [\frac{h_{\text{in}} - e}{s}] + 1$

Max-pooling is a very interesting technique due to being locally invariant, which means that even if the inputs shifts a bit around, the output stays constant, this is extremely useful for visual algorithms. However, if it is enforced too much the network might struggle to carry useful information, so, the spatial extension must be kept quite small.

2.1.6 Transfer Learning

Transfer Learning is used to obtain better results while taking less time to train a model, this is achieved due to reusing previously trained weights. Transfer learning is especially useful when working with small datasets since it enables to use a larger network while minimizing the risk of overfitting [20], [125].

Transfer learning is performed by copying the weights of n layers of the base network to n layers of the target network, the remaining layers from the target network are randomly initialised [123].

The Transfer Learning pipeline can be described as follows:

1. Train a base network on a base dataset and task;
2. Transfer the features learned to a second target network;

3. Train the target network with an target dataset and task.

This way, instead of creating new features from scratch, the network will reuse the generalised features from the previously trained weights [123].

Transfer learning could be done using two different approaches: frozen features and fine-tuned features. In the frozen features method, the n layers that were copied to the target network are not trained, only the newly added are trained. In the second method, all the layers are trained.

2.1.7 Regularize Layers

Regularization is used in order to control the overfitting problem, helping to restrict the out-of-control parameters. In the following subsections, it will be explored the most common techniques used in this field.

Batch Normalization

During training, the distributions of each layer's input changes, which will affect the parameters of the following layers, difficulting the training of neural networks. To overcome this issue, generally, lower learning rates are used and careful parameter initialization is done, which will affect the training time. However, batch normalization acts like a regularizer, allowing higher learning rates and fewer issues from the parameter initialization. This will permit to obtain faster training while achieving the same accuracy. Batch normalization is applied to every convolutional layer before the activations being applied, which is computed as follow:

$$y = \frac{x - E[x]}{\sqrt{\text{Var}[E] + \epsilon}} \times \gamma + \beta, \tag{2.35}$$

where the mean and variance are calculated per-dimension over the mini-batches, γ and β are a learnable parameter from the vector of size C, where C is the input size. During training, these layers keep running estimates of its computed mean and variance, which are then used for normalization during evaluation [54].

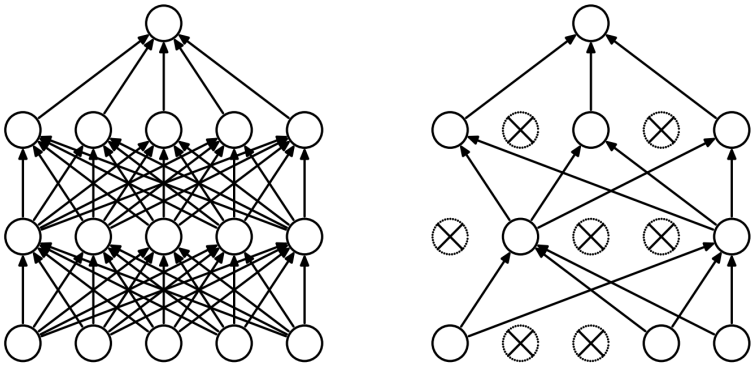
Dropout

DNN have a large number of parameters, enabling them to generalise by analysing large amounts of data, however, overfitting is a common problem. To solve this issue, the Dropout technique [104] is frequently used.

Dropout is only used during training and generally only on the last layer(s) of the model. This technique will randomly drop units from the layers (Figure 2.13) by a previously set percentage, which prevents units from co-adapting too much. This normalization technique has helped major improvements in the model performances.

2.1.8 Early Stopping

A common alternative to regularization is the procedure of *early stopping*. During the training phase, the error relative to the training is non-increasing, possibly causing the system to overfit. To overcome this problem, a solution is to use an independent dataset,



(a) Standard Neural Net (b) After applying dropout.

Figure 2.13: Dropout exemplification on a two hidden layer Neural Network Model: (a) standard version during the training phase; (b) dropout applied during the training phase, some random units have been dropped (from [104]).

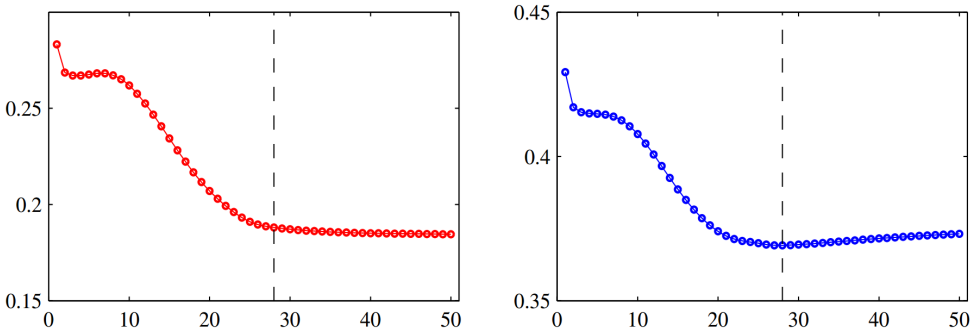


Figure 2.14: Demonstration of the behaviour of training set error (left) and validation set error (right) during the training phase of a network. According to the early stopping method, the training should be stopped around the vertical dashed lines, which corresponds to the point where the error from the validation set reached its minimum (from [7]).

called the validation set, in which the error will be also evaluated. This way, when the error from the validation set starts to increase and the error from the training set continues to decrease (Figure 2.14) the system starts to overfit the data, so the training can be stopped at this point. Generally, the checkpoint associated with the minimum error obtained from the validation set is chosen.

2.2 General Concepts

This section explains some general concepts that are useful to better understand how object detectors work and some techniques that are commonly used.

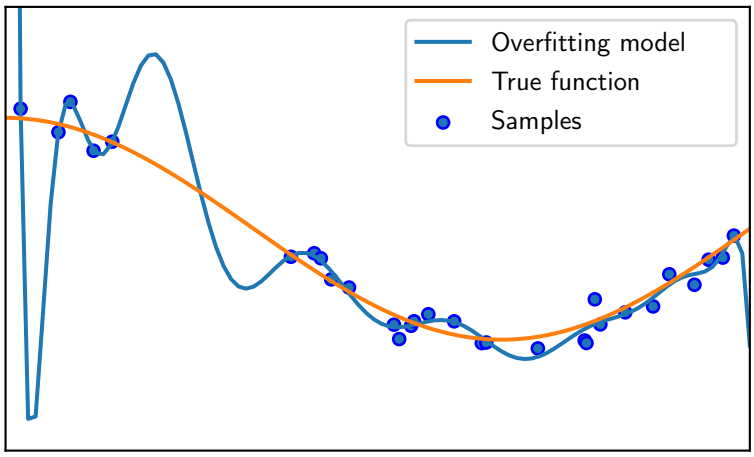


Figure 2.15: In blue we have a model that is overfitting the sample data (blue dots), that was generated by adding noise to the true function (orange) (inspired by[1]).

2.2.1 Overfitting

Overfitting occurs when a system has a low error during training, but it works poorly in the real-world or when evaluated using independent data. This happens due to poor generalization by the system (Figure 2.15), which instead "memorizes" all the feature of the data [3]. Overfitting could happen for many reasons, such as:

- Small training dataset;
- Model to complex;
- Training data too noise (eg. data errors and outliers).

2.2.2 Intersection over Union

Intersection over Union (IoU), also known as Jaccard index or Jaccard similarity coefficient, is a statistical method of measure similarity of samples sets, which is widely used in data mining [111], this metric is also used in object detection quite frequently.

IoU is computed as follows [25]:

$$IoU(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}, \tag{2.36}$$

which consists in dividing the intersected by the union area of two bounding boxes as shown in Figure 2.16.

[1]http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html

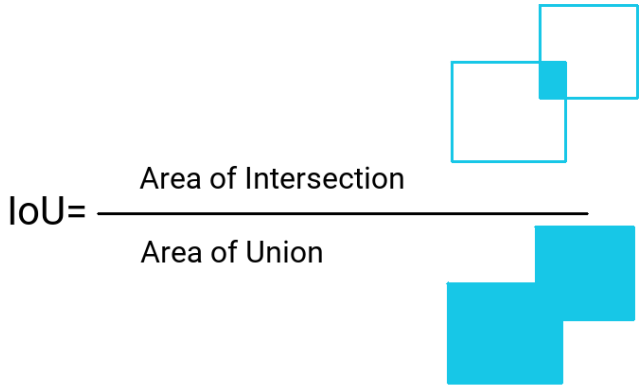


Figure 2.16: Visual exemplification of Intersection over Union metric.(inspired by[2])

2.2.3 Non-Maximum Suppressor

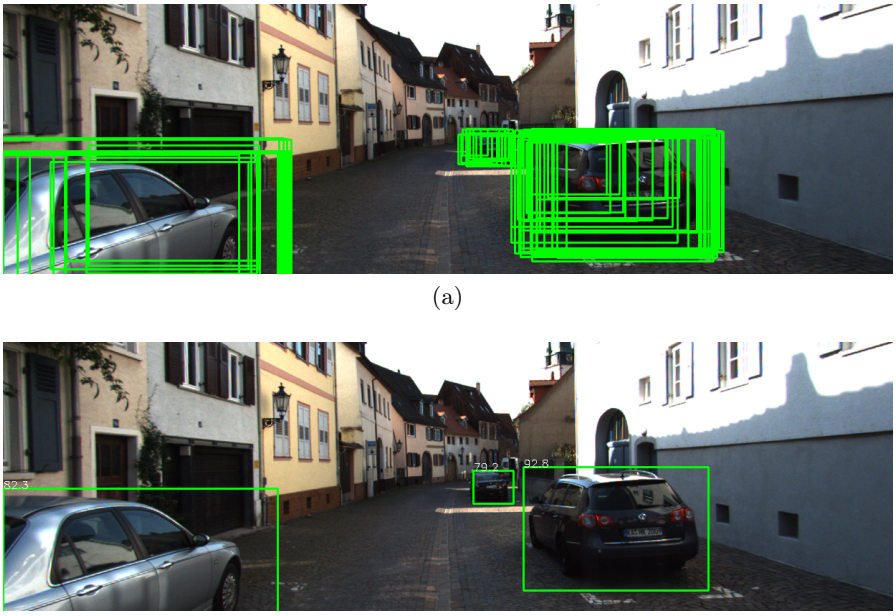
Non-Maximum Suppression (NMS) is a commonly used tool in object detection. That occurs because, object detectors produce multiple boxes with high confidence scores near the target objects, which happens because of the ability of these detectors to generalise. Since, ideally, its only desirable a bounding box per object, the outputs must be filtered [92].

To filter the excess of bounding boxes, NMS first sorts them by the confidence given by the object detector. Then, it picks the bounding boxes with the highest confidences and discard the ones that overlap them, avoiding multiple boxes overlapping the same object. Generally, the maximum IoU allowed between boxes is 50%; however, this could be adjusted according to the dataset [28]. Figure 2.17 shows the effect of applying or not the NMS.

2.2.4 *k*-means Clustering

k-means Clustering is an unsupervised learning technique, which means that it does not need labels for learning. This technique is commonly used to partition data into homogeneous groups called clusters, where k refers to the number of clusters that the data will be divided [59] , [1]. Figure 2.18 shows a k-mean Clustering result.

[2]https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/



(a)

(b)

Figure 2.17: Comparison between applying or not NMS in an output image from an Object Detector (a) Input image without applying NMS; (b) Input image applying NMS. This image is from KITTI Dataset.

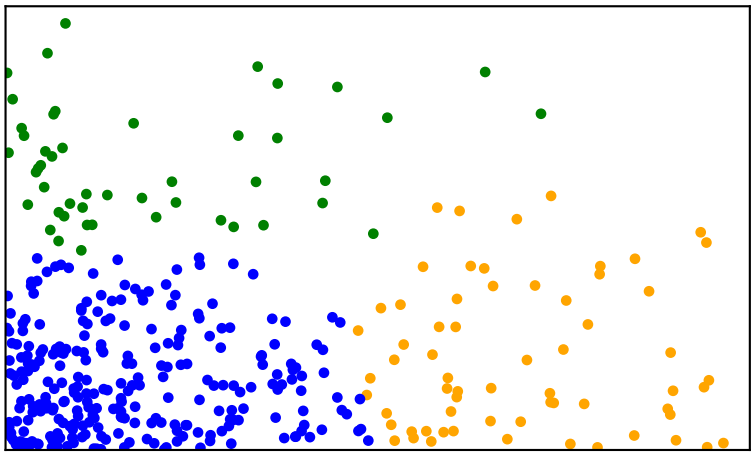


Figure 2.18: k-means Clustering with k equal to 3, each cluster is represented by a different colour.

Chapter 3

State-of-the-Art in Object Detection

The object detectors can be defined by three main approaches: Classic Object Detectors, Two-Stage Detectors and One-Stage Detectors.

Classic Object Detectors was the first successful approach using machine learning techniques. In 2012, after the resurgence of neural networks, with deeper and wider networks, Two-Stage Detectors appeared, overpassing the results obtained by classical methods, with better precisions, but achieving a lower frame rates. Then, One-Stage Detectors appeared, with a different approach, an end-to-end solution, taking advantage of CNN performance and speed. This enabled high frame rates, maintaining high precisions, comparing with other methods.

In the next section, these methods will be introduced and we will go a bit deeper in the One-Stage Detectors since it is the approach taken by this thesis.

3.1 Classic Object Detectors

Classic Object Detectors consists of four stages: (1) generating candidate regions on the given image by the sliding window technique, (2) extracting relevant features from these regions, (3) classifying and identifying regions through a trained classifier, (4) revise the detections and optimize their results:

1. **Generating Candidate Regions:** In this stage, the objective is to obtain the objects' location; for that, a sliding-window is passed through the image. The sliding window should have different dimensions since objects could appear in any place and with any dimension. This process represents a high computation cost due to the high number of windows that are redundant.

2. **Feature Extraction:** After obtaining the objects' location it is necessary to classify them, but before, must be extracted the features that will be used by the classifier. These features will influence the performance of the classifier, however, they are difficult to design. This difficulty derivates from the variety of external factor that happen in real-world scenarios (*e.g.* change of angles, illumination, moving objects, etc.), that makes difficult to generalise. In contrast, controlled environments are a lot easier to extract features from, due to their lack of variety.

3. **Classification:** At this stage, the extracted features will be used by the classifier, which generally is SVM or Adaptive Boost (AdaBoost) model.

4. **Revise Detection Results:** After performing the previous steps, the framework will output multiple windows identifying the same object. To filter them, in order to obtain only one per object, generally NMS is used.

Classic object detector resort to multiple steps, making it highly complex and time-consuming, which results in poor performances [127].

One of the first research that used this method was Viola and Jones [114], which used boosted object detectors for face detection; this lead to a general adoption of this technique. In [77] this method was used to detect cars, which achieved, in a custom dataset, a hit ratio of 78% but a false detection ratio of 75%.

3.1.1 Rapid Object Detection by Viola and Jones

Viola and Jones [114] used boosted object detectors, obtaining a system that detects faces at 15 Frame per Second (FPS), with a detection rate of 95% and a false positive rate of 1 in 14084. This research is distinguished by three key factors:

1. **Integral Image:** a new image representation, allowing the detector to computer rapidly the features used.

2. **AdaBoost:** a learning algorithm which would select key visual features from the dataset, allowing to obtain an extremely efficient classifier [31].

3. **Cascade Method:** combines increasingly more complex classifiers, allowing to rapidly discard image regions that were background.

This allowed object detectors to be used in real-time applications.

In 2012, with the resurgence of deep learning [58], Two-Stage Detectors quickly came and lead the detection paradigm. In the next section, we will explain what them.

3.2 Two-Stage Detectors

As referred before, Two-Stage Detectors became a method widely used in object detection. One of the first researches using this method was Selective Search for Object Recognition [113]. The process of Two-Stage Detectors consists of generating candidate regions (1) and then of classifying them (2):

1. **Generating Candidate Regions:** In this stage a Region of Interest (RoI) is generated which should contain the objects present in the image.

2. **Classification:** After obtaining the RoI, it is needed to classify the proposed regions into foreground/background classes [1]. R-CNN [36], was a breakthrough in the Object Detectors field, since it used CNN to perform this stage, allowing major gains in precision, surpassing other methods, as for example Classic Object Detectors.

[1] where foreground corresponds to the regions with objects or RoI, and background regions without objects.

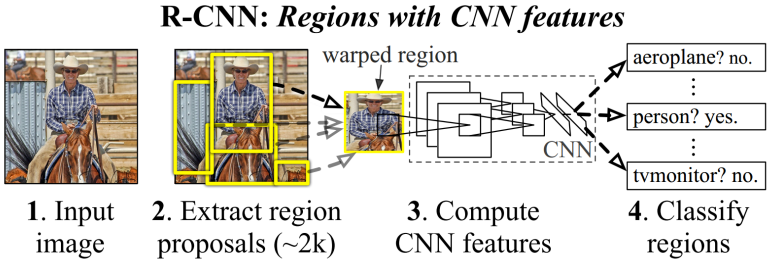


Figure 3.1: R-CNN pipeline: (1) the system takes an input image, (2) extracts around 2000 region proposals, (3) computes features for each proposal using a CNN, and (4) classifies each region using a class-specific linear SVM (from [36]).

R-CNN was an inspiration for many researchers, being upgraded along the years in terms of speed [35], [43], [89] and by using learned region proposals [22], [83], [35]. Another upgrade was done by joining Region Proposal Network (RPN) the first-stage (generation of candidate regions) with the second-stage (region classification) into a single convolution network, creating the Faster R-CNN framework [35], which numerous extensions of this framework have been made, *e.g.* [64], [101], [43].

3.2.1 Regions with Convolutional Neural Networks features

Regions with Convolutional Neural Networks features (R-CNN) [36] surpassed the results gotten in previous works at the time, still being an inspiration for many researches in this field.

At the time R-CNN was developed, object detectors generally used regressions or sliding windows, but these did not obtain great results [23]. So, they opted by using a region proposal area method [15], which had already obtained good results for object detection [113] and semantic segmentation [11].

R-CNN was one of the first implementations that used deep CNN to improve object detection performance. For this they combined two key factors: (1) high-capacity CNN and (2) transfer learning:

1. **CNN:** The high-capacity CNN deals with the region proposals, localization and segmentation of the objects.

2. **Transfer Learning:** Since the data is limited and is expensive to label new data, it was used a pre-trained network, to auxiliary the task, followed by a domain-specific fine-tuning.

Figure 3.1 represents the networks pipeline [2], where first is given the input image to the system, which will be extracted region proposal areas. After, a CNN will compute features for each proposal and then an SVM will classify each region. R-CNN achieved a mean Average Precision (mAP) of 53% on PASCAL VOC [26].

[2]Pipeline: is the infrastructure surrounding the algorithm. This includes gathering the data, use it for training, training and export the model for further tests or for production.

3.2.2 Faster R-CNN

Faster R-CNN [89] is a continuation of previous works, Fast R-CNN [35] and R-CNN [36], which was able to surpass in precision and frame rate from its predecessors.

To better understand Faster R-CNN, lets first take a quick view on Fast R-CNN. The main propose of Fast R-CNN was to reduce the detection running time, where they determinate that R-CNN biggest bottleneck was the region proposal computation. To improve its speed, they used RPN, which consist of a Fully CNN that simultaneously predicted bounding boxes and confidence scores for each position. This enabled an end-to-end approach for training, which generated high-quality region proposal.

Faster R-CNN to overpass Fast R-CNN, merged RPN further into its system, this resulted in a single network that shares the convolutional features in both stages [43], [35], which the first stage is a Fully CNN that proposes regions and the second stage is the Fast R-CNN detector that uses the proposed regions. To improve further its performance, they introduce a new feature, "anchor" boxes. "Anchor" boxes are used as references for the objects, which have different scales and aspect ratios.

To train the model, they alternate between fine-tuning the region proposal task and the object detection task. This unifies the RPN and Faster R-CNN object detection network, which will converge quickly and where the convolutional features will be shared by both tasks, as stated before.

Faster R-CNN can achieve 5 FPS on a Graphics Processing Unit (GPU), with state-of-the-art precisions in multiple datasets, obtaining only 300 proposals per image, instead of 3000 as R-CNN. To achieve this results, Faster R-CNN takes advantage of the GPU reimplementing methods that generally are implemented on the Central Processing Unit (CPU). This made the computational cost of the proposal areas have a substantial decrease, where each iteration only took 10ms per image, cutting the time taken by 50% compared to its predecessor.

With Faster R-CNN they were able to obtain a 69.9% mAP in PASCAL VOC test set [26], with VGG-16 architecture [121], and in MS COCO [65], with the same architecture, they obtained a 73% mAP, at 15 FPS.

3.3 One-Stage Detectors

One-Stage Detectors merges the steps of Two-Stage Detectors further, resulting just in a network that receives an input image and returns their bounding boxes, with almost no mid-steps. These detectors can reach much higher FPS than other solutions.

One of the first frameworks using this method was OverFeat [30], which inspired a lot of researches in this field. Recent work on One-Stage Detectors, such as YOLO [87], YOLO 9000 [88] and Single Shot Detection (SSD) [66], demonstrates promising results, obtaining much faster detectors comparing to Two-Stage Detectors or other approaches taken, but they obtain a worse precision then Two-Stage Detectors. However, RetinaNet [64] tried a different path, instead of focusing on the speed they tried to get higher precisions, which they were able to achieve, beating at the time the benchmarks of Two-Stage Detectors.

Since our aim is to detect cars and pedestrians in real-time, we will need high frame rates. For that reason, our design will coincide with One-Stage Detectors like YOLO [88], [87] and SSD [66] or other researches in this field. The next subsections will go into

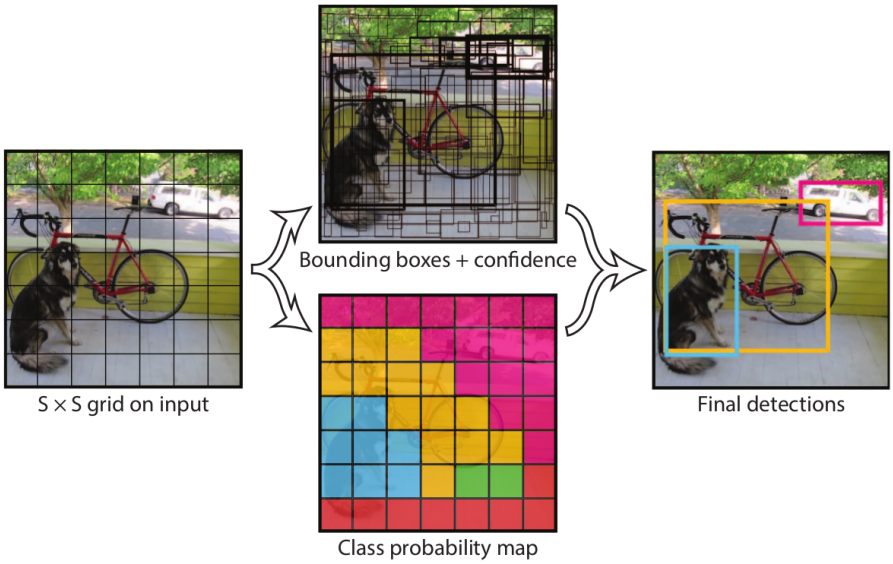


Figure 3.2: YOLO first divides the image into a $S \times S$ grid, which each grid cell will predict B bounding boxes, with an associated confidence C per bounding box, and the class probabilities c for each cell (from [87]).

more detail on the papers since they are more relevant to our work.

3.3.1 You Only Look Once

Previous works on One-Stage Detectors repurposed classifiers to perform detection, however, YOLO [87] opts to take a different approach, where they "*frame object detection as a regression problem to spatially separate bounding boxes and associated class probabilities*" [87].

Since the YOLOs approach consists in using an end-to-end system, relying mainly on the network that has a constant output, however, the number of objects varies. To deal with the variation on objects per image, YOLO opts by dividing the input image into an $S \times S$ grid, which each cell of the grid will have n predictors [3], where an object is assigned to a grid cell if its bounding box centre falls inside the cell. Each cell will also output the object class if the cell contains an object. This grid is designed to enforce spatial diversity, making the predictors uniformly distributed, Figure 3.2 illustrates this pipeline.

To train, YOLO uses full-size images as input data, which the system tries to predict the object positions. Then, with the usage of a loss function, it is compared the predicted position with the GT, which computes an error that will be optimized accordingly. To

[3] A predictor is responsible by localizing objects in the image, outputting the position and dimensions of the bounding box and a confidence of its prediction, in some systems it also classifies the object in question.

optimize this network during the training phase, the following loss function is used:

$$\begin{aligned}\text{Loss} =& \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x_i})^2 + (y_i - \hat{y_i})^2 \right] + \\ & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w_i}} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h_i}} \right)^2 \right] + \\ & \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 + \\ & \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 + \\ & \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p_i}(c) \right)^2, \end{aligned} \tag{3.1}$$

where the variables with a hat represent the predictions and without the GT, x_i and y_i are the centre coordinates of a bounding box, and w_i, h_i are the according width and height of the i-th cell; $p_i(c)$ is the class probability of a given cell, which the GT of c is represented by a one-hot vector [4]; C_i is the confidence that each predictor outputs, allowing to measure how well the predictor "thinks" the prediction was made; $\mathbb{1}$ is a mask that will only allow computations to be made on the predictors that had an allocated GT, if the superscript is equal to *obj*, or the remaining ones if the superscript is equal to *noobj*. The parameters λ_{coord}, λ_{noobj} are a scalar factor that increases or decreases the importance of an element in the loss calculation.

During training, the scarce of label data is a common problem, so, during YOLOs development, it was used two methods to overcome this issue. Instead of training the model from scratch, first was trained a custom model for classification with ImageNet dataset [56], [93] with an input image of 224 by 224 pixel, then they double its input resolution and convert the network to perform object detection by adding convolution layers to the pre-trained network. The process of adding new layers was shown that helps networks to learn new tasks by [90].

YOLO [87] is a One-Stage Detectors which largely focus on delivering high frame rates, but it lacks in precision comparing to Two-Stage Detectors. Within the first released version, they have 3 different architectures: Fast YOLO (155 FPS, 63.4% mAP), YOLO (45 FPS, 63.4% mAP) and YOLO VGG-16 [121] (21 FPS, 66.4% mAP) trained on PASCAL VOC [24].

Limitations of YOLO

Since YOLO imposes strong spatial constraints on bounding box predictions, this limits the number of objects that it can detect. Another issue with this model is that it learns to predict bounding boxes from data and, therefore, struggles to generalize new objects, in other words, it does not detect well objects of a trained class that are a lot different

[4] A one-hot vector is a $1 \times N$ matrix, where N is, in this case, the number of classes, which the index of the GT is equal to 1 and the rest of the vector is zero.

from the trained ones. It also performs poorly on detecting small objects, this occurs because the loss function treats in the same way errors small or large objects.

3.3.2 YOLO 9000

YOLO 9000 [88] consists of an improved version of YOLO [87], which allowed to detect over 9000 object categories, achieving a higher precision with similar FPS.

YOLO 9000 authors observed that the datasets for classification were more complete than for object detection, where classification datasets commonly have thousands of classes and millions of examples [64], [128], however, object detection datasets generally have dozens of classes and a few thousands of examples [26], [93], [64]. This happens mainly because the labelling cost is very different, being a lot cheaper to just classify images. For that reason, YOLO 9000 takes advantage of classification datasets proposing a joint training between detection and classification. This enabled YOLO 9000 achieving this perk.

As we said before, YOLO 9000 also surpasses the precision levels of its predecessor. To achieve that, instead of applying a deeper and larger network as commonly is done [121], [107], [43], since they also wanted to maintain their frame rates high, they opted by simplifying the networks' training, making the representations easier to learn. For that they used the following techniques: (1) Batch Normalization, (2) High-Resolution Classifier, (3) Convolution With Anchor Boxes, (4) Dimension Cluster, (5) Fine-Grained Features, (6) Multi-Scale Training, which can be defined as follows:

1. **Batch Normalization**: This method not only helps to regularize the model but also eliminates the need for other forms of regularization [54]. Using this method on all the convolution layers YOLO got an improvement in mAP of more than 2%.

2. **High-Resolution Classifier:** Generally, state-of-the-art detection methods use a pre-trained model on ImageNet [56], so, they first trained a network with an input resolution of 224x224 pixel, when it finished, they increased it to 448x448 pixel and fine-tuned it, then they converted this network to perform detections. With this process, they were able to gain increased in mAP of almost 4%.

3. **Fine-Grained Features:** In order to improve one of the biggest limitations of YOLO, detecting small objects, they used finer-grained features for localizing smaller objects, this increased the performance by 1%.

4. **Convolution With Anchor Boxes:** In YOLO 9000 it was removed the fully-connected layers, turning the model to an FCN. They also used anchor boxes to predict the bounding boxes, allowing to classify each prediction in each cell independently. Despite using anchor boxes, the mAP decreased, however, the recall increased.

5. **Dimension Cluster:** To be easier for the network to learn, the anchor dimensions, instead of being picked by hand, were obtained by running a k-means cluster, which got the priors automatically.

6. **Multi-Scale Training:** To make the model more robust, they train the model with different input sizes, this could be achieved because the model only uses convolutional and pooling layers, which can be resized on the fly.

With all these improvements was possible to increase the mAP to about 76.8% on PASCAL VOC 2007 [25], and still having 67 FPS with a resolution of 416 by 416 pixels.

3.3.3 SSD: Single Shot MultiBox Detector

SSD appeared after the first version of YOLO, which resulted, naturally, in higher precisions and frame rates. To increase its performance they used a standard architecture for image classification and then added an auxiliary structure to the network. But their key difference, comparing with YOLOs approach, was that SSD combines multiple feature maps with different resolutions, which helped to deal with different size objects [100], [41], [85]. Some other key features were also developed, such as:

Convolution predictors for detection: the feature layers that were added can produce a fixed number of predictions using a set of convolution filters.

Default boxes and aspect ratios: the default anchors/priors were associated with each feature map cell.

Matching strategy: to allocate the GT with the anchors, it is calculated the IoU between them. If the IoU is higher or equal than 0.5 they are matched, which will mean that a higher number of predictors will be trained at each iteration [5]. This allowed the network to predict with high scores for multiple overlapping default boxes.

Training objective: SSD loss function is inspired from the MultiBox objective function [22], [109], however, can deal with multiple object categories, which is defined by a weighted sum of the localization loss (L_{loc}) and the confidence loss (L_{conf}):

$$L(x, c, l, g) = \frac{1}{N}(L_{\text{conf}}(x, c) + L_{\text{loc}}(x, l, g)), \tag{3.2}$$

where x is an indicator for matching the anchors to the GT and N is the number of labels matched to the default boxes, however, if $N = 0$ then the loss is set to zero. This way creating some kind of mask to only calculate the loss in the positions where exist an object. For the localization loss L_{loc}, it was used a smooth Mean Absolute Error (L1) [35] between the predicted box (l) and the GT box (g) parameters. The confidence loss L_{conf} is a normalized exponential function (softmax) loss over multiple classes confidences (c).

Hard negative mining: Generally the foreground represents a smaller area than the background, this creates an imbalance between true positives and true negatives. For this reason, SSD opts by only using the highest confidence loss for each anchor and chooses the top ones so that the ratio between them is at least 3:1 (positives, negatives). They claim that this approach leads to a faster and more stable optimization.

Data augmentation: As a way to make the model more robust, was performed virtual data augmentation. For that, besides vary the input object sizes by changing shapes and perform flips, they also have used one of the following options during training:

- Use the entire original input image;
- Sample a patch so that the minimum IoU is 0.1, 0.3, 0.5, 0.7 or 0.9;
- Randomly sample a patch.

[5] An Iteration is the process doing a forward and then a backward propagation.

The size of each patch was $[0.1, 1]$ of the original image size and the aspect ratio was between $\frac{1}{2}$ and 2. After this, they resized the image to a fixed size and applied some photo-metric distortions similar to the ones described by [47].

After applying these methods, they got two versions, a network with a 300x300 pixel input image, which achieves on PASCAL VOC 2007 74.3% mAP at 59 FPS and another network with 512x512 input, which got 76.9% mAP and 22 FPS.

3.3.4 Focal Loss for Dense Object Detection

Focal Loss for Dense Object Detection [64] resulted in a breakthrough in One-Stage Detectors, since until now, the leading models in precision where Two-Stage Detectors, however, they were able to surpass all methods.

For this achievement, RetinaNet first discovered that the biggest issue that One-Stage Detectors face is the foreground/background class imbalance, as boosted detectors [114], [19] and Deformable Part Model (DPM) [27] or more recent works, SSD [66], dealt. This imbalance causes an inefficiency in training since a lot of background areas do not contribute with useful data to train, which leads to overwhelming training and degenerated models.

A common solution for foreground/background class imbalance is to perform hard negative mining [114], [106], [27], [101], [66], which sample hard examples during training or using more complex sampling/reweighing schemes [8], [42]. However, this paper created the Focal Loss, which allowed to efficiently train on all examples. To compute this loss, they reshaped the cross-entropy loss function in order to down-weight easy examples, focusing on the hard ones. For this, they created a modulating factor:

$$(1 - p_t)^{\gamma}, \tag{3.3}$$

where γ is a tunable parameter which must be higher or equal to zero, p_t can be defined by:

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & otherwise, \end{cases} \tag{3.4}$$

where p is the model estimated probability for the class with label $y = 1$.

So, with (3.3) and (3.4), the Focal Loss is computed as follows:

$$FL(p_t) = -\alpha_t (1 - p_t)^{\gamma} log(p_t), \tag{3.5}$$

where α and γ are tunable parameters. The focusing parameter γ smoothly adjusts the rate at which easy examples are down-weighted (Figure 3.3); α is used to balance the loss function, which also helped improve accuracy. It was also observed that if this loss is combined with a sigmoid operation for computing p, also results in a better numerical stability.

RetinaNet surpassed all other object detectors on MS COCO [65], achieving a 40.8% mAP, where YOLO 9000 obtained 21.6%, SSD 513x513 31.2% mAP and the state-of-the-art at the time, Faster R-CNN with Top-Down Modulation (TDM) [101] achieved a 36.8% mAP.

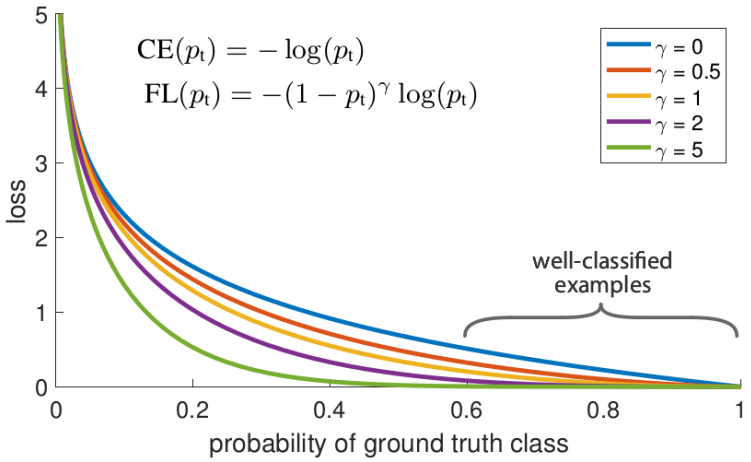


Figure 3.3: Different variations of γ in the Focal loss (from [64]).

3.3.5 SqueezeDet

SqueezeDet is a small FCN specialized in object detection, which is inspired in YOLO [87] but uses SqueezeNet architecture [52] as his backbone. This architecture can achieve AlexNet precision, however, using far fewer parameters.

SqueezeDet achieved on KITTI dataset [33] a mAP of 76.7%, reaching up to 57.2 FPS.

3.4 Summary

Object detector suffered a lot of changes throughout time. Nowadays, the research is mainly focused on Two-Stage Detectors and One-Stage Detectors, where Two-Stage Detectors generally obtain better precision but lower frames rates comparing to One-Stage Detectors, with the exception of RetinaNet [64] that has similar frame rates as Two-Stage Detectors but obtains higher precisions.

For this thesis it will be used some ideas from the papers described here, sometimes to tweak our model, others just to prove some concepts. However, the main inspiration came from YOLO [87], YOLO 9000 [88] and SqueezeDet [116] to construct our model.

Chapter 4

Model Characterization

This section explains the options taken during the process of creating the base structure of the model. For that, it will be described the model's architecture and the loss function.

4.1 Architecture

The architecture plays a major role in the systems' performance. The majority of the recent researches on CNNs is focused on increasing precision in computer vision problems. This fact can be observed on one of the most known competitions of image classification, ImageNet [56], [93], in which the results were surpassed year after year [93]. The majority of the newly proposed models' architecture tend to become deeper and larger, meaning that they would have a higher number of parameters, resulting in a longer inference time[1].

Our goal is to have a fast feedback from the network, which means to obtain a low inference time. This is needed because our system will be used in a self-driving car, so it must perform in real-time. To understand what it means to run in real-time its needed a benchmark. The psycho-motor reaction of a human-being to trigger a brake pedal, in a real-world scenario, falls between 0.42 and 0.92 seconds [57]. However, this includes the process of evaluating the scenario and reacting to it. Considering that this networks only detects the objects, our inference time must be the lowest possible. So we assume that for an object detector to run in real-time it must reach to at least 30 FPS, which is the same frame rate as the most common cameras available in the market.

A model will work in real time if it contains a low amount of parameters. This characteristic will lower the computational cost, achieving higher frame rates, however, for an object detector be used in a self-driving car it must achieve high precision as well.

A Model can achieve high precision even with low amount of parameters, since for a given precision exists multiple architectures [51]. For example, Network In Network (NiN) [64] has about 8x fewer parameters comparing to AlexNet [58], but they have similar precision. Furthermore, GoogLeNet [107] and VGG [121], share a similar precision, even so, GoogLeNet as 10x fewer parameters. This is especially possible due to the usage of smaller filters sizes in the convolutions and by using a FCN. This way, it was possible to

[1]Inference time corresponds to the time that the network takes to make the forward propagation during evaluation. In other words, is the time that the network takes since it is given its input data until outputs predicted data.

Layer name/type	**Kernel Size / Stride**	s_{1x1}	e_{1x1}	e_{3x3}
Input Image				
Convolution	3x3/2 (96)			
MaxPool	3x3/2			
Fire 1		16	64	64
Fire 2		16	64	64
MaxPool	3x3/2			
Fire 3		32	128	128
Fire 4		32	128	128
MaxPool	3x3/2			
Fire 5		48	192	192
Fire 6		48	192	192
Fire 7		64	256	256
Fire 8		64	256	256
Convolution	1x1/1 (1000)			
AvgPool	13x13/1			

Table 4.1: SqueezeNet Architecture.

reduce the number of parameters and connections but still maintaining high precisions.

A convolutional network with fewer parameters has significant advantages, such as training faster than larger networks. This happens because the parameters communicate between them, then with fewer parameters, will also have fewer connections [51], resulting in a faster development and training time. For these reasons, we decided to select SqueezeNet [52] as our architecture.

4.1.1 SqueezeNet

SqueezeNet is an FCN, which the goal is to have the same precision as AlexNet [58] but with fewer parameters. They were able to achieve this with about 50x fewer parameters then AlexNet.

Lowering the parameters is essential, since this allows a more efficient distributed training, less overhead when exporting new models to clients and in embedded platforms. SqueezeNet resorts to many strategies to achieve this goal, such as:

- **Replace $e_{3\times3}$ filters with 1×1 filters:** this allows decreasing the number of parameters by a factor of 9.
- **Decrease the number of input channels to $e_{3\times3}$ filters:** since the number of parameters of a layer is equal to the number of input channels times the number of filters times the filter size. It is not only essential to decrease the filters sizes, but also the number of input channels. For this reason, SqueezeNet decreases the number of input channels to $e_{3\times3}$ filters by using *squeeze layers*, which will be described further.
- **Downsample the output from the layers late in the network so that convolution layers have large activation maps:** each convolutional layer in a

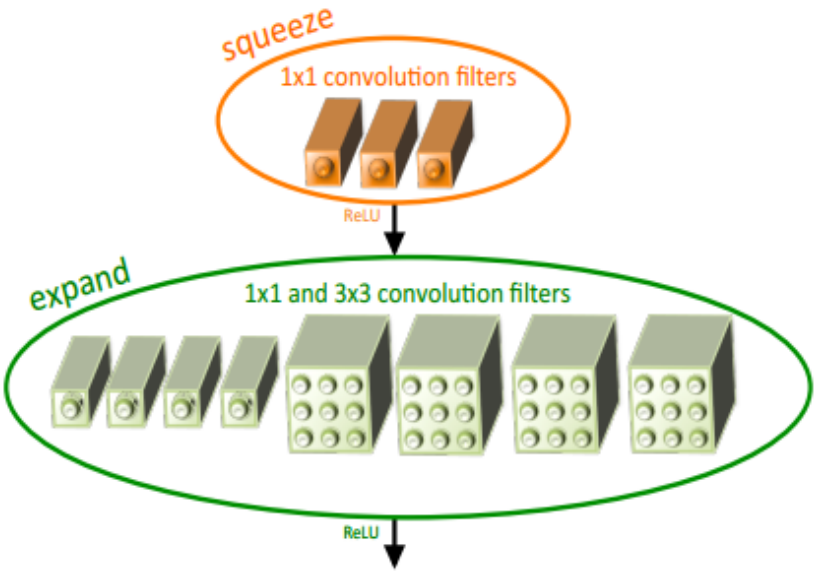


Figure 4.1: Fire Module, which is the building block of SqueezeNet (from [52]).

> CNN outputs an activation map with a spatial resolution that is at least 1x1, this width and height are determined by the size of the input data and the choice of the downsampling applied to the layers. Generally, this downsampling is done by the stride in the convolution and pooling layers [58], [107], [121], since large activation maps can lead to higher classification precision, with all else held equal, as shown by [43], the large strides are concentrated at the end of the network.

The first and second strategies are more concerned with minimizing the number of parameters to have a small network and the last strategy was chosen to achieve higher precisions.

To deploy the strategies earlier described, SqueezeNet created what they called Fire modules (Figure 4.1). A Fire module is comprised of: a *squeeze* convolution layer (which has only 1×1 filters), feeding into an *expand* layer that has a mix of 1×1 and 3×3 convolution filters. These layers have three hyper-parameters $s_{1\times1}$, $e_{1\times1}$ and $e_{3\times3}$, which represent:

- $s_{1\times1}$**:** number of channels in the squeeze layer with 1×1 filters.
- $e_{1\times1}$**:** number of channels in the expand layer with 1×1 filters.
- $e_{3\times3}$**:** number of channels in the expand layer with 3×3 filters.

For the Fire module to work, it is needed that the squeeze layer has fewer channels than the summed expand layers ($e_{1\times1} + e_{3\times3}$), limiting the number of input channels of the expand layer with 3×3 filters.

SqueezeNet does not contain only Fire modules and also the Fire modules are not all equal. So, SqueezeNet architecture starts with a convolution layer, followed by 8 Fire modules, ending with a final convolutional layer. The number of filters per Fire module is gradually increased from the beginning to the end of the network. It also contains a max-pooling layer with a stride of 2 after the first convolution, the fourth and eight Fire module and in final layers, as shown in Table 4.1.

Layer name/type	**Kernel Size / Stride**	s_{1x1}	e_{1x1}	e_{3x3}
Input Image				
Convolution	3x3/2 (96)			
MaxPool	3x3/2			
Fire 1		16	64	64
Fire 2		16	64	64
MaxPool	3x3/2			
Fire 3		32	128	128
Fire 4		32	128	128
MaxPool	3x3/2			
Fire 5		48	192	192
Fire 6		48	192	192
Fire 7		64	256	256
Fire 8		64	256	256
Fire 9		96	384	386
Fire 10		96	384	386
Convolution	3x3/1 (n)			

Table 4.2: SqueezeDet Architecture.

Other strategies applied

Besides the main strategies and methods described before, some other techniques were also used, such as:

- For the outputs of the layers, with 1×1 or 3×3 filters, having all the same size, it was added a 1-pixel border of zero-padding in the input data of the expansion layers with 3×3 filters.
- The activations applied to the layers of SqueezeNet were ReLU activations [74].
- To prevent early stopping due to overfitting Dropout [104] a ratio of 50% was used as a regularization method.
- SqueezeNet did not use fully-connect layers, because is inspired on the NiN [64] architecture also to lower the number of parameters.
- The starting learning rate used in SqueezeNet was 0.04, which was decreased linearly throughout the training phase as described in [72].

hese techniques made possible that SqueezeNet to have the same precision as AlexNet but with far fewer parameters.

4.1.2 Vanilla SqueezeDet Architecture

SqueezeNet was created to address the ImageNet Competition [93], which means that the output layer gives 1000 values to address each class represented in this competition.

Since our objective is to not only classify but also to detect objects, we need to modify this architecture to perform our task.

Before starting modify the network,firstly the desired output must be defined. As said in Section 3.3, each image, generally, has a different number of objects. To deal with this, the most common way is to divide the image with an x by y grid [64],[116], [88], where it will be performed i predictions in each of these cells. These predictions will output five scalars plus the number of classes, resulting on an output vector of output $= (n_x, n_y, n_{\text{predictors}}, 5+n_{\text{classes}})$. When the desirable output given by the network is defined, it is possible to start modifying architecture.

First, the last convolution and average-pooling layer that was specialized in classification it will be removed, likewise as with YOLO 9000 [88]; then two Fire modules and a convolution layer it is added, similar to other models [90], [116]. The following hyper-parameters can characterize these Fire modules: $s_{1\times1}$ with 96 channels, $e_{1\times1}$ with 384 channels and e_{3x3} with 386 channels, as shown in Table 4.2. The convolution layer contains a kernel size of 3×3 and a stride of 1. The number of channels of the last convolution layer will depend according to the number of classes and predictors per cell, being equal to $n_{\text{predictors}} \times (5 + n_{\text{classes}})$.

4.2 Objective Function

Deep learning, using supervised learning, involves optimization, which could vary between minimization or maximization. To accomplish this optimization is needed an objective function or criterion, that may also be called cost, loss or error function when minimization is used. This function measures how well the model is doing accordingly with the predictions and the GT, this way, an objective function guides the model through the training, improving his performance [38].

For our framework, we created the vanilla cost function, which is described in the next subsection. This objective function it is mainly an interpretation of YOLO [87] and SqueezeDet [116], being a fusion of both concepts.

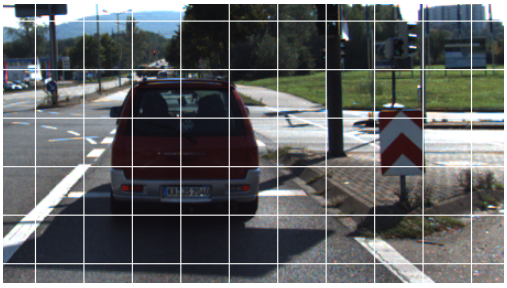
4.2.1 Vanilla Cost Function

The vanilla cost function will be the used basic formulation in the cost function. In this version the standard methods to detect objects were used. Throughout the course of the experiments small tweaks in the vanilla cost function will be performed affecting its performance. This way we can understand how these tweaks affect the performance and which ones are favourable to our framework. Since our objective is to create a One-Stage detector, all the computation will have to be done by the network, so the cost function must deal with an end-to-end solution.

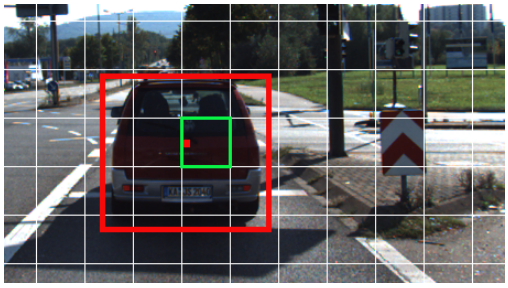
Object Allocation

To better understand how the vanilla cost function works it will be first explained how GT was generated.

The GT provided to the network represents the way it is allocated labels to the predictors. Since the number of objects differs from image to image, but the output does not, it must be found a way to work around. If, for example, it is given n labels



(a) Representation of grid "division" .



(b)

Figure 4.2: In (a) is represented the grid that is applied on each image to create a spatial distribution, (b) shows how it is allocated a cell to a specific GT. The red dot is the centre of the bounding box, the green box is the limited area of the cell and the red box is GT.

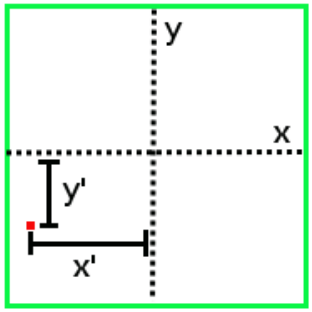


Figure 4.3: Offset between the cell and bounding box centre.

to the model, and they are not spatial constrict, the network will perform poorly. This happens for multiple reasons, for instance, it is not guaranteed a uniform distribution of the predictors, which would agglomerate around the easier to detect while ignore the hard ones. To overcome this problem, it must be ensured a uniform spatial distribution of the predictors, so to each predictor was allocated an specific area.

To create this uniform spatial distribution, it is generated an S_x by S_y grid on the image, as shown in Figure 4.2 (a). Each grid cell will ensure an uniform distribution, only "allowing" the predictors to look at the objects that falls inside them. The allocation of an object to a specific cell is determined by its centre position, as shows Figure 4.2 (b). These objects are represented by a bounding box, which is defined by x, y centre coordinates, the width w and height h.

The labels spatial positions are encoded to easier the network learning . As the positions of the cells are well defined, when an object falls in a cell it is regressed from where it was felt inside the cell, instead of from the global position. For that, it is calculated the offset between the cell and the object (Figure 4.3), which is computed as follows:

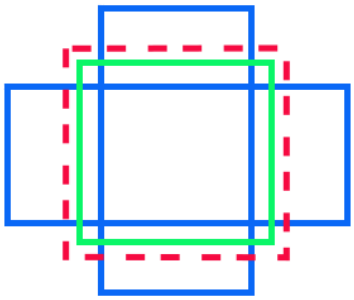


Figure 4.4: Anchor allocation to an object. Solid lines represent anchors, and the dashed line the object. The green anchor has the highest overlap with the object.

$$\begin{aligned} x'_i &= x^G - x_i, \\ y'_j &= y^G - y_j, \end{aligned} \tag{4.1}$$

where x'_i and y'_j are the offset position, x^G and y^G the GT position given by the label, x_i and y_j the coordinates of cell centre position. Which i and j correspond to the cell localization within the grid.

To width and height GT, it is used a different process. As explained in subsection 4.1.2, the output of the network will be output $= (n_x, n_y, n_{\text{predictors}}, 5+n_{\text{classes}})$. Meaning that each cell from the grid will output n predictors. To facilitate the model's training, each predictor should specialize in a particular object shape, since each class has its own particular shape, for example, cars tend to be wider and the pedestrians thinner. Using this analogy, it can be easily understood that an object from a particular class has a predisposition to have a given shape. To allocate these shapes to the predictors are used the priors, also known as anchors [89].

The priors, or anchors, can be created in different ways. The easiest one is by hand pick, when the researcher already has an idea of the object shapes and selects the most common ones. Another alternative is to cluster the data, recurring, for example, to a k-means Cluster.

To allocate an object to a prior, the IoU between them is calculated in order to know which is more fitted to learn about that object. Figure 4.4 shows how this process is made, considering that the dashed line is the object and the solid lines are the anchors.It can be observed that the green anchor has the highest overlap of the object and it will be the most fitted to learn its features.

Similar to the positional allocation, the width and height can also be encoded, as shown in the next equation:

$$\begin{aligned} w'_k &= \frac{w^G}{w_k}, \\ h'_k &= \frac{h^G}{h_k}, \end{aligned} \tag{4.2}$$

where w'_k and h'_k represent the ratio between the GT width and height (w^G, h^G) and the priors width and height (w_k, h_k), where k is the prior.

After (4.1) and (4.2), it is performed one more transformation, where x'_i and y'_jcoordinates are divided by the allocated anchor:

$$\begin{aligned} \delta x^G_{ijk} &= \frac{x'_i}{w_k}, \\ \delta y^G_{ijk} &= \frac{y'_j}{h_k}, \end{aligned} \tag{4.3}$$

And for w' and h' it is computed their logarithms:

$$\begin{aligned} \delta w^G_{ijk} &= log(w'_k), \\ \delta h^G_{ijk} &= log(w'_k), \end{aligned} \tag{4.4}$$

where δx^G_{ijk}, δy^G_{ijk}, δw^G_{ijk} and δh^G_{ijk} represent values given to the network during training.

These computations, (4.3) and (4.4), allows the network deal in a better way with different sized objects due to give different weights according to the object size.

Loss Function

After generating the GT, it must be calculated the loss function, due to the high level of its complexity is divided into three segments: position, confidence and classification loss.

Position Loss

Position loss will compute the error corresponding to the localization of objects. For this, its calculated the square error between the prediction and GT of δx_{ijk}, δy_{ijk}, δw_{ijk} and δh_{ijk}, and then it is multiplied by the mask $\mathbb{1}^{obj}_{ijk}$, this way only computing the loss for predictors that had an associated object, and finally, sum all these values. These computations are denoted as follows:

$$\begin{aligned} \text{Pos.Loss} = &\sum_{i=0}^{S_x}\sum_{j=0}^{S_y}\sum_{k=0}^{B} \mathbb{1}^{\text{obj}}_{ijk}\left[\left(\delta x^G_{ijk} - \delta x^P_{ijk}\right)^2 + \left(\delta y^G_{ijk} - \delta x^P_{ijk}\right)^2\right] + \\ &\sum_{i=0}^{S_x}\sum_{j=0}^{S_y}\sum_{k=0}^{B} \mathbb{1}^{\text{obj}}_{ijk}\left[\left(\delta w^G_{ijk} - \delta w^P_{ijk}\right)^2 + \left(\delta h^G_{ijk} - \delta h^P_{ijk}\right)^2\right], \end{aligned} \tag{4.5}$$

where the superscript P represents the predicted values, and G the GT, S_x and S_y represent grid size, and B the number of predictors per cell.

Confidence Loss

The confidence that the network outputs (C^P), consists in the predicted value of the IoU between the predictor and predicted object. Since IoU varies between [0, 1], it is applied a sigmoid filter to the confidence outputted by the network, enforcing this interval.

The confidence loss was divided into two steps, when a predictor has an allocated object and when it does not. To calculate the first step, after each iteration it must be

calculated the IoU (C^G) between the predictors and the allocated objects. Afterwards, it is used the following equation to compute its loss:

$$\text{Conf.Loss}_{\text{obj}} = \sum_{i=0}^{S_x} \sum_{j=0}^{S_y} \sum_{k=0}^{B} \mathbb{1}_{ijk}^{\text{obj}} \left(C_{ijk}^{G} - \sigma(C_{ijk}^{P}) \right)^2, \tag{4.6}$$

where σ refers to the applied sigmoid function.

If a predictor does not have any object allocated, its confidence should be equal to zero, however its common to be higher than that, so it is also computed the following loss:

$$\text{Conf.Loss}_{\text{noobj}} = \sum_{i=0}^{S_x} \sum_{j=0}^{S_y} \sum_{k=0}^{B} \mathbb{1}_{ijk}^{\text{noobj}} \left(\sigma(C_{ijk}^{P}) \right)^2, \tag{4.7}$$

Classification Loss

Classification loss will calculate the error made when predicting an object class. For that, it is computed the cross-entropy between the predicted classes and GT, similar to positional loss. It will be only calculated the loss for the predictors that had an allocated object. This computation is denoted as follows:

$$\text{Class.Loss}(c, \text{class}) = \sum_{i=0}^{S_x} \sum_{j=0}^{S_y} \sum_{k=0}^{B} \mathbb{1}_{ijk}^{\text{obj}} \text{- } \log \left(\frac{\exp(c[\text{class}])}{\sum_{l} \exp(c[l])} \right), \tag{4.8}$$

where *class* is the GT, c are the predictions made.

Total Loss

After calculate all the necessary losses, they must be concatenated . For that, it is used the following function:

$$\begin{aligned} \text{Loss} = & \frac{\lambda_{\text{coord}}}{\sum \mathbb{1}_{ijk}^{\text{obj}}} \text{Pos.Loss} + \\ & \frac{\lambda_{\text{conf}}}{\sum \mathbb{1}_{ijk}^{\text{obj}}} \text{Conf.Loss}_{\text{obj}} + \\ & \frac{\lambda_{\text{conf}}}{\sum \mathbb{1}_{ijk}^{\text{noobj}}} \text{Conf.Loss}_{\text{noobj}} + \\ & \frac{1}{\sum \mathbb{1}_{ijk}^{\text{obj}}} \text{Class.Loss}, \end{aligned} \tag{4.9}$$

where λ_{coord} and λ_{coord} are balance parameters. Notice that each summed loss is divided by the number of elements of their mask, obtaining this way, the mean error per predictor.

During training, the balance parameters were set to $\lambda_{\text{coord}} = 75$ and $\lambda_{\text{conf}} = 100$.

4.2.2 Inference

Since the model was trained using an encoded GT, during the inference, its output values must be decoded by inverting the encoding equations.

To obtain the positional coordinates referent to a predictor, it must be performed the following computation:

$$\begin{aligned} x^P &= x_i + w_k \delta^P_{ijk}, \\ y^P &= y_j + h_k \delta^P_{ijk}, \end{aligned} \tag{4.10}$$

where x^P, y^P are its the decoded x and y coordinates resulted from inverting (4.1) and (4.3). To decode the width and height of the object, it is computed the following equation:

$$\begin{aligned} w^P &= w_k exp(\delta w^P_{ijk}), \\ h^P &= h_k exp(\delta h^P_{ijk}), \end{aligned} \tag{4.11}$$

where w^P, h^P are the decoded coordinates *width* and *height*, which resulted from inverting (4.2) and (4.4).

To obtain the confidence value of a prediction the sigmoid function must be applied as well, otherwise, the output interval will be different than [0, 1]. For the class probabilities, it is also applied the softmax.

At test time the conditional class probability it is multiplied by the individual box confidence predictions:

$$Pr(\text{Class}_i|\text{Object}) \times \Pr(\text{Object}) \times \text{IOU}^{\text{truth}}_{\text{pred}} = \Pr(\text{Class}_\text{i}) \times \text{IOU}^{\text{truth}}_{\text{pred}}, \tag{4.12}$$

which gives the class-specific confidence scores for each box.

After decoding all the network outputs, the predicted bounding boxes are passed throughout an NMS algorithm.

Chapter 5

Data Handling

This chapter discusses the KITII Dataset [33], it is also explained how the augmentation data was performed during training. 0

5.1 KITTI Dataset

Datasets are a key factor for any machine learning method, without them, it would not be possible to perform these techniques. Since we want to create an object detector for a self-driving car, it must be chosen a dedicated dataset to this objective, so we opt by KITTIs dataset [33].

KITTI Dataset was developed by Karlsruhe Institute of Technology and Toyota Technological Institute of Chicago, which consists of real-world computer vision benchmarks of roads, being one of the most known in this field. This dataset counts with benchmarks for multiple tasks such as optical flow, visual odometry/Simultaneous Localization and Mapping (SLAM), 3D object detection, 2D object detection and road segmentation, these benchmarks also have an associated evaluation metric.

To collect the data, they used an autonomous driving platform, which contains four high-resolution cameras, a Velodyne laser scanner and a localization system. The data was captured around the mid-size city of Karlsruhe, in rural areas and highways. The data is provided in a raw format, containing the corresponding labels for each task, where each image has up to 15 cars and 30 pedestrians.

For our experiments, it was used the 2D object detection benchmark, which contained 7481 labelled images from real-world scenarios, which its labels had the following information:

- **Bounding box:** left, top, right, and bottom pixel coordinates of the bounding box.
- **Object class:** which could be "Car", "Van", "Truck", "Pedestrian", "Person_sitting", "Cyclist", "Tram", "Misc" or "DontCare".
- **Truncation:** percentage of the object that leaves image boundaries, in other words, when the bounding box does not correspond to the full extent of the object.
- **Object occlusion**: which could be fully visible, partly visible, largely occluded or unknown.

KITTIs benchmark for 2D object detection only evaluates the detection of cars, pedestrians and cyclists, despite existing other classes. This benchmark is also divided into three difficulty levels: easy, medium and hard, which are defined as follows:

- **Easy:** Minimum bounding box height 40 Px, Maximum occlusion level: Fully Visible, Maximum Truncation: 15%.
- **Medium:** Minimum bounding box height 25 Px, Maximum occlusion level: Fully Visible and Partly occluded, Maximum Truncation: 30%.
- **Hard:** Minimum bounding box height 25 Px, Maximum occlusion level: Fully Visible, Partly occluded and Largely occluded, Maximum Truncation: 50%.

5.1.1 Evaluation Methodology

The evaluation tool of KITTIs 2D benchmark follows PASCAL criteria [25], which needs seven parameters: image reference, object class, confidence, left, top, right, bottom pixel coordinates. Using these parameters it is calculated the precision-recall curve and the mAP.

Metrics

As said before, PASCAL criteria use the precision-recall curve and mAP to evaluate the models' performance. So, first let's understand these concepts:

- **Precision** is the fraction of relevant retrieved information over the retrieved information, in object detection, this will measure the percentage of detections that were correct.
- **Recall** is the fraction of relevant retrieved items over the total amount of relevant items, in object detection, this will measure the percentage of detect objects in an image.
- **mAP** is the mean of the average precision scores.
- **mean Average Recall (mAR)** is the mean of the average recall scores.

To calculate precision and recall True Positive (TP), False Positive (FP) and False Negative (FN) are needed, which are defined as:

- TP: number retrieved and relevant items.
- FP: number retrieved but not relevant items.
- FN: number not retrieved but relevant items.

The calculation of precision for a given class c, is computed as follows:

$$\text{Precision}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c}. \tag{5.1}$$

For calculating mAP:

$$\text{mAP} = \frac{\sum_{c}^{\text{classes}} \text{TP}_c}{\sum_{c}^{\text{classes}} (\text{TP}_c + \text{FP}_c)} \tag{5.2}$$

The recall of a given class c, is denoted as:

$$\text{Recall}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FN}_c}. \tag{5.3}$$

For calculating mAR:

$$\text{mAR} = \frac{\sum_{c}^{\text{classes}} \text{TP}_c}{\sum_{c}^{\text{classes}} (\text{TP}_c + \text{FN}_c)} \tag{5.4}$$

All these calculation are based on [95].

Calculating Metrics

To evaluate a model, first must be defined what will be considered a TP, FP or FN. According to PASCAL criteria, a detection is considered TP if the IoU between the object and prediction is higher than 50% and correctly classified, however, if one or both of these criteria failed, then the prediction is considered FP. If an object did not have any predictor correctly detecting them, then is considered an FN. If multiple detections occur for the same object, only one will be counted as TP, the rest being considered FP.

KITTIs evaluation considers in a different way TP for cars[1], which the IoU between the predictor and object must be higher than 70%. KITTIs also consider a TP if a van is detected as a car, however, we do not follow this rule because we want to only detect cars and pedestrians.

When we evaluate our models, besides precision and recall, it is also calculated the localization, background and repetition error using algorithm 1, that is setted with the KITTIs evaluation benchmark rules, however, when using PASCALs criteria, it must be changed the algorithm minimum IoU to 50% for all objects.

5.1.2 Dataset Proprieties

Explore the dataset is extremely important due to the information that it contains, however, to avoid bias towards data, it must be used only the training data. It will be only analysed cars and pedestrians, since its the aim of this thesis.

To analyse the KITTIs dataset, it will be gathered the following pieces of information: class balance, object positional and shape distribution, and occlusion balance.

[1] http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=2d

Algorithm 1 Evaluator Algorithm

Require: Objects; Predictions

```
detected = [False] × len(Objects)
n_correct = 0
n_missed = 0
loc_error = 0
bg_error = 0
repeated_error = 0
for Prediction in Predictions do
    if Prediction == 'car' then
        min_IoU = 0.7
    else min_IoU = 0.5
    IoUs = calculate_IoU(Objects, Prediction)
    max_IoU = max(IoUs)
    idx_IoU = argmax(IoUs)
    if max_IOU > 0.1 then
        if max_IoU > min_IoU then
            if not detected[idx_IoU] then
                n_correct += 1
                detected[idx_IoU] = True
            else repeated_error += 1
        else loc_error += 1
    else bg_error += 1
for det in detected do
    if  not det then
        n_missed += 1
TP = n_correct
FP = repeated_error + loc_error + bg_error
FN = n_missed
```

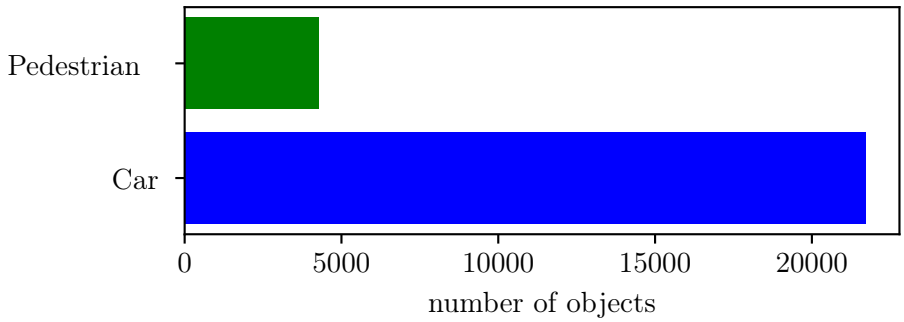


Figure 5.1: Class balance, between cars (21707) and pedestrians (4276), of KITTI Dataset.

Class Balance

Figure 5.1 shows the class balance between cars and pedestrians, where is observable that exists an imbalance [2] between the classes, which cars represent 83.5% of the total number of objects. This class imbalance plays a significant role on the performance attainable by learning methods, which generally assume a uniform distribution of classes.

Object Shape Distribution

Figure 5.2 shows the distribution of the objects shapes, which is observable the difference between pedestrians and cars, this is due to pedestrians, generally, be narrower than cars.

Since we are using anchors in our framework, would be opportune to use a clustering algorithm, this way, obtaining better generalization for the priors.

Object Positional Distribution

Figure 5.3 shows the positional distributions of the objects, which perspective is clearly visible, where the objects fall especially in the centre and borders of the images.

Despite the distribution between cars and pedestrians be quite similar, it is observable that on the left side appear more cars, yet, on the right side are more pedestrians. Since Karlsruhe is in Germany and cars drive at the right, its expectable this fact.

Occlusion Balance

The objects occlusion is a determinant factor during the training of the network, since, high occlusion level will results in a harder feature extraction due to less similarity between objects.

Figure 5.4 shows the balance of occlusion per class, which majority of the objects are fully or partially visible, especial among pedestrians. However, a significant portion of the cars are partially or largely occluded, this might be due to a high overlap, as for example in parking lots, or traffic queues.

[2] Class imbalance corresponds to domains where one class is widely represented, and the other only has a few examples [55].

5.2 Data Preparation

Before using KITTIs dataset, it is divided in three folders, train, validation and test set, corresponding to 74%, 14%, 12% of the data, which the test set only was used once

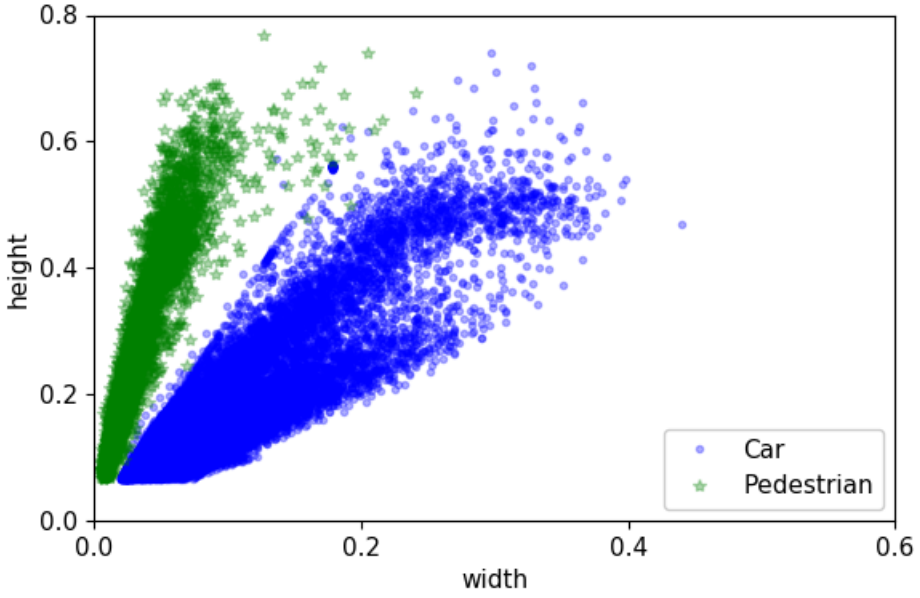


Figure 5.2: Normalized width and height scatter plot of the objects from KITTIs' dataset.

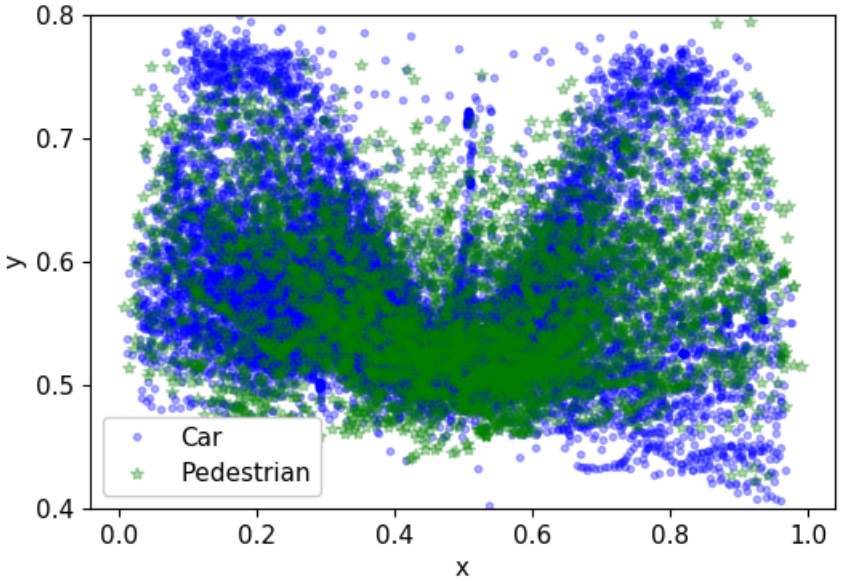


Figure 5.3: Normalized x and x coordinates scatter plot of the objects centre from KITTIs' dataset.

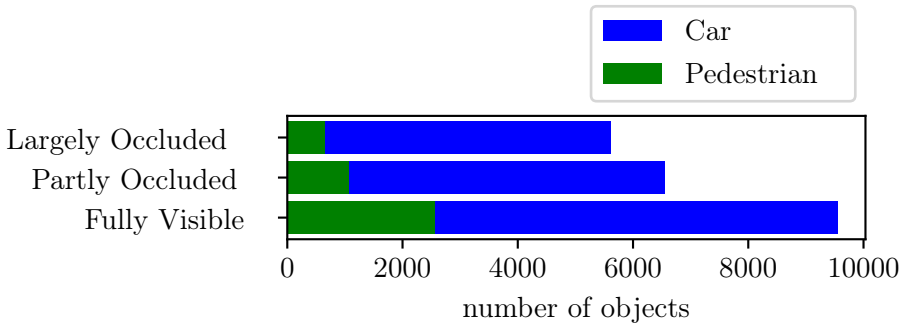


Figure 5.4: Number of objects per occlusion level (a) in cars and (b) in pedestrians, on KITTIs dataset.

per model during evaluation, avoiding bias towards data. Afterwards, data is filtered reducing unnecessary details from data [32]. For that it is filtered-out the non-important information from labels, such as, other classes information and objects that do not fit the **hard** difficulty set by KITTIs, minimizing possible errors. During training, the images must be converted from the range between [0, 255] to [0, 1], this is done because using high input values might affect the activations during training, resulting in a poor models' performance [81].

5.2.1 Data Augmentation

Data augmentation is a common technique practised in computer vision due to the high cost of manual labelling new data [67] and by reducing the potential risk of overfitting the data [46]. The use of data augmentation might even help improve results, as shown by [47], which reduced the top-1 error rate by over 2% in ImageNet using this technique. Consequently, many methods were developed for data augmentation such as flips, crops [58], [47], [122], [125], colour casting [117], blur [2], among others.

During our training we also perform data augmentation, using the following methods:

- **Horizontal Flips:** with a probability of 50%
- **Random Rotation:** between the angles [-2.5, +2.5]
- **Random Resize Crop:** random size crop between [0.1, 1] of the original image size and a random aspect ratio between [2, 4] of the original aspect ratio.
- **Colour Jitter:** random variation between [0%, 50%] of brightness, contrast and saturation and from [0%, 1%] on hue.

Figure 5.5 and Figure 5.6 shows these methods During training, it was used all these techniques at the same time. When performing these methods, it was also made the correspondent transformations on the bounding boxes to maintain its coherence. These methods are performed on-the-fly, reducing unnecessary Random-Access Memory (RAM) usage, allowing more variate transformations. Data augmentation is only done in the training set, maintaining the validation set unaltered.

(a)

(b)

(c)

(d)
colorjitter

Figure 5.5: Augmentation data techniques: (a) original image; (b) flipped image within vertical axis; (c) random rotation of the image; (d) random crop with a random scale of the image.

(a)

(b)

Figure 5.6: Augmentation data techniques: (a) random change in saturation, brightness and hue of the image; (b) applying all the previously described methods at the same time.

Chapter 6

Model Fitting

This chapter is divided into two parts, framework and experiments, first it will be explained which materials were used, next, it will be explained the experiences made.

6.1 Framework

Before training the models, some aspect must be taken into to account, for instance, what kind of machine should be used? which programming language? which libraries? In this section, it will be explored these concepts and why they were used.

6.1.1 Machine

During the 90's several attempts to create specific hardware to neural networks or to exploiting existing ones were attempted, but not successfully. In 2000' the GPUs started to become cheaper, being widely used for video games, which changed the GPUs market, being more competitive and bigger, making them cheaper.

GPUs are great at computing matrices and vector multiplications, a requirement to train ANN, which can speed up learning by a factor of 50 or more. A lot of recent successes in contests for pattern recognition, image segmentation and object detection, used ANN trained in GPUs [99].

Since we are training models for computer vision, which contain millions of parameters with thousands of images, it was needed a computer with some computational power. So, to train the models it was used the computer with the following specifications: Nvidia Tesla k40c with 12 Gigabyte (GB) of memory, 32 GB of RAM memory and an Intel Xeon E5-2640 @2.50Ghz CPU. The CPU is mainly used to pre-process the images and deploy them on the network, and the RAM memory is specially used to save the networks progress. The key factor in our computer is the GPU since it will deal with all the training process of ANN.

6.1.2 Programming Language

Python is a high-level programming language for general-purpose programming and its explorative nature leads to being widely used for data science, thus contains a large number of useful libraries developed by its community, making it highly versatile which is excellent for machine learning [91].

Python despite not being the fastest language to perform computation-intensive tasks, many Python libraries, extensively used in Data Science such as *NumPy*, *Pandas* and *SciPy* have been developed upon lower layer Fortran and C or C++ implementations to boost its speed [86]. This way we do not need to compromise its speed or computational power, making Python ideal for our purpose.

6.1.3 Library

Currently are many deep learning libraries that support Python, such as TensorFlow, Keras, Theano and Pytorch, which are the most used in this field [1]. Most of them can realize the same task with similar performances, but we opt by using Pytorch, which is the tool of choice for many researchers due too its flexibility and intuitive integration with Python. It is not considered Theano since is no longer supported, which could lead to many problems. It is also rejected the hypothesis of using Keras because it runs on top of TensorFlow which could give less flexibility in the experiments. Despite TensorFlow being highly used in this field, it is too low-level to use comfortably for rapid prototyping, being mainly used at a production level.

Pytorch supports reverse-mode [103] and automatic differentiation [39] of scalar functions as most of the deep learning libraries. But it distinguishes by some features, such as [80]:

- **Dynamic, define-by-run execution:** Generally, deep learning libraries use static graph structure, this differentiates symbolically ahead of time and then are run many times. Instead, Pytorch uses a dynamic framework which defines the function to be differentiated simply by running the desired computation.

- **Immediate, eager execution:** This will allow the framework to run tensor computations as it encounters them, enabling the execution of the CPU and GPU to be pipelined.

- **No Tape:** It enables the users to mix and match independent graphs how they want, without explicit synchronization. This makes possible when a graph or a portion of a graph becomes dead it to be automatically freed, making it possible to free the memory as quickly as possible for other tasks.

- **Core logic in C++:** Majority of Pytorch is written in C++, which makes possible to achieve much lower overhead compared to other frameworks.

- **Extensions:** It is possible in Pytorch to create custom differentiable operations by specifying the forward, which computes the operations, and backward functions, which extends the vector-Jacobian product. This allows the usage of Python libraries, making them differentiable.

- **Memory management:** Pytorch is essentially used for machine learning model on GPU, being the low GPU memory capacity one of the biggest limitations, Pytorch frees memory as soon as they became unneeded in order to combat this issue.

[1] Information obtained from: https://github.com/thedataincubator/data-science-blogs/blob/master/output/DL_libraries_final_Rankings.csv

Anchor nº	Width	Height
1	36	37
2	366	174
3	115	59
4	162	87
5	38	90
6	258	173
7	224	108
8	78	170
9	72	43

Table 6.1: Width and height defined in pixel for anchors sizes for object detection using KITTI dataset.

These points make Pytorch an ideal library to be chosen for our research, and that why we use it.

6.1.4 Programming the loss function

As was said in the section 6.1.1 and 6.1.3, GPU is what boosts the training of ANNs due to its efficiency on computing matrices and vector multiplications. For these reasons, it is important to do all computations using matrices to not create unwanted bottlenecks. So, during training all the calculations needed for the loss function should be made through matrices or inherent calculations, allowing to take full advantage of the GPU.

6.1.5 Training Hyper-Parameters

During the experiments, we want to maintain our models as similar as possible, this way it is observable what the tweaks made in their performances. For that reason, the hyper-parameter were maintained through the experiments.

As the optimization algorithm, it is used the SGD with a starting learning rate of 0.01, a momentum of 0.9 and weight decay of 0.005. It is also used a learning scheduler, that will change the learning with the pass of epochs, which every 60 epochs it multiply the learning rate by a factor of 0.25. It was used a batch size of 20 [2]. To be sure that our models trained to his maximum performance, they trained for 300 epochs [3]. The values that will be used for the anchors are presented in table 6.1, which are the same used in SqueezeDet [116] for detecting pedestrians, cars and cyclists.

[2]Batch size is the number of images, in this case, that will be trained at the same time.
[3]An epoch is when the full training set passed through the network during the training phase.

6.2 Experiments

This section describes the experiments that will be performed.

6.2.1 Vanilla Version

Vanilla Version consists of the control version, which will be trained according to the previous chapters. As result, it is expected a nice performance but nothing extraordinary, since it is not performed any special enhancement.

6.2.2 No Augmentation Data

In this experiment, it is opted by not using any technique of augmentation data, showing problem of using small datasets and the influence that these techniques have on the models' performance. Accordingly, the performance is expected to substantially lower when comparing it to the Vanilla Version.

6.2.3 Batch Normalization

Batch Normalization will be integrated with architecture for this experiment, making possible to remove the previous regularization methods, dropout. Since in YOLO the usage of batch normalization boosted its mAP over 2%, it is also expected some increase in the resulting performance of this experiment.

6.2.4 Focal Loss

In object detection one of the most significant problems described by [64] is the extreme foreground-background class imbalance during training. So they formulate a loss function that focuses on hard examples and down-weights the contribution of easy examples, they call this loss the *Focal Loss* (in the section 3.3.4 it is explained how the focal loss works). To see what improvements could be achieved by using the *Focal Loss*, it was integrated it into the vanilla cost function in this experiment.

Integrating focal loss in the loss, it expectable a major improvement from the network's performance, since RetinaNet was able to outperform other object detectors especially due to this tweak.

6.2.5 Matching Strategy

In Vanilla version, a label will only be allocated to an anchor if this anchor has the biggest IoU among all (the IoU is calculated between the anchor and label). However, SSD does a different approach, associating a label to any anchor that obtains a IoU higher or equal than 50%. This strategy will probably result in higher performance.

6.2.6 No transfer Learning

This experiment will train the network from scratch, initializing the layers weights and bias through a uniform distribution, which probably will result in a poor performance. This way, proving that the use of transfer learning is necessary for an optimal result.

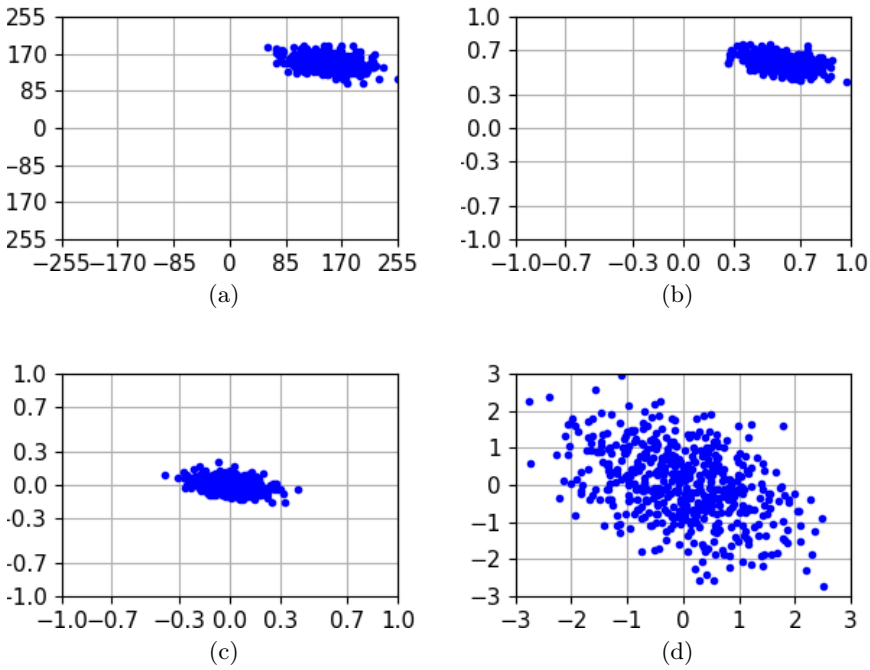


Figure 6.1: Different representation of data: (a) Original data plotted in 2-dimensional graph; (b) Normalization of the original data, by min-max scaling shift and rescale; (c) Usage of the process data represented in (b) and zero-centred by subtracting mean in each dimension. The data is now centred around the origin; (d) Normalize and centre the data according to the previous steps plus standardizing by dividing the standard deviation of the data by each dimension.

6.2.7 Frozen layers

Frozen layer is a technique commonly used when training networks on the same task but with different data. So, it be will frozen the layers that are common between both architectures, only training the newly added. This change will result in a faster training since it has fewer weights to update, however probably will perform worse that fine-tune all the layers.

6.2.8 Data Standardization

In Vanilla model training, the input data is normalized according to min-max scaling. Min-max scaling shifts and rescales the values, so that they be between [0, 1], instead of the original interval [0, 255]. However,standardization consists of first subtracting the mean values and then divide them by its standard deviation. In Figure 6.1 its represented these transformations. In this experiment the input data will be normalized and then standardized. We believe that this process could make small improvements to the network performance.

Anchor nº	Width	Height
1	109	58
2	276	156
3	78	175
4	36	36
5	212	108
6	378	184
7	36	94
8	70	44
9	155	81

Table 6.2: Width and height defined in pixel for anchors sizes for object detection using KITTI dataset and k-means Cluster.

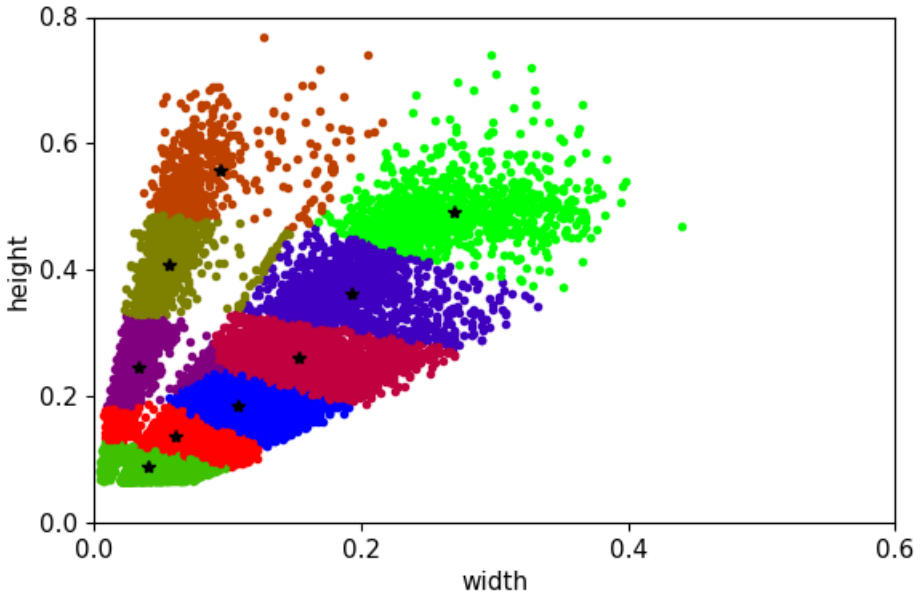


Figure 6.2: k-means cluster with k equal to 9, using the normalized aspect ratios from KITTIs training data; the stars represent the centre of each cluster.

6.2.9 k-means Cluster getting the anchors

Object detectors generally use priors, or anchors, to improve their performance. Since we only want to detect car and pedestrians, a good way to obtain these anchors is to cluster the most common width and height of these classes. For that, it is used a k-means cluster algorithm that computed 9 clusters (Table 6.2), which are shown in Figure 6.2. These anchors should have the optimal size to detect these classes, which will probably improve the network performance.

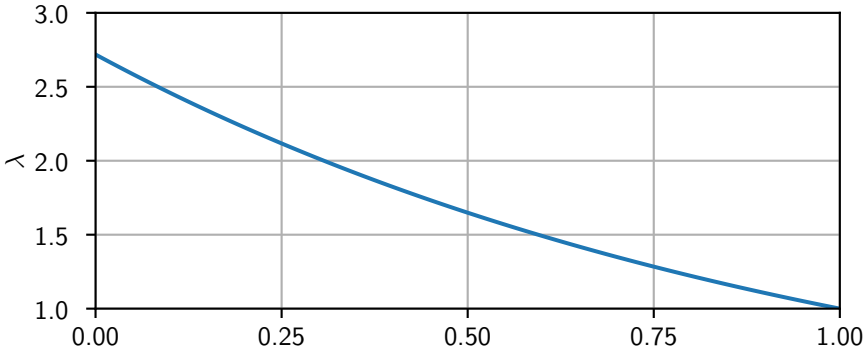


Figure 6.3: Graphic of λ in function to IoU.

6.2.10 λ in function of IoU

To try to improve the results from Vanilla Version, we formulated a dynamic balance variable λ that will be multiplied by the positional and confidence loss. The idea of this variable is to increase the weight of the predictors that achieved a low IoU and decrease the others ones. This way, optimizing the localization of the predictors, improving the overall network's performance. This variable λ can be defined by:

$$\lambda(\text{IoU}) = \exp(1 - \text{IoU}). \tag{6.1}$$

Figure 6.3 shows its graphical representation.

6.2.11 Lower Input

This experiment will decrease the size of each image (375×375) that is used to train. This will increase the speed of the model during inference time and decrease the training time, however, it is expectable a lower precision.

6.3 Other Trained Models

To compare the experiments other models were trained. This will allows to compare the results of the experiments with these models using the same evaluation tools. For this,it was opted to train two frameworks, SSD, which is a one-stage detector like ours, and Faster R-CNN which is state-of-the-art of the Two-Stage Detectors. These models will use previously trained weights on COCOs dataset [64].

6.3.1 SSD: MobileNet v1

MobileNet [48] is an architecture that was built for mobiles and embedded vision application. For that, MobileNet uses a streamlined architecture which uses depth-wise separable convolution. This way, they were able to build an architecture with a smaller number of parameters, similar to SqueezeNet. This architecture is generally used for classification.

Fusing MobileNet with SSD framework was possible to obtain 33.3 FPS and a mAP of 21 % on COCO [64]. This model is the biggest "rival" to our framework since both architectures have approximately the same size.

6.3.2 SSD: Inception v2

Inception v2 [110] was an architecture that the main goal was to explore ways to use the computational cost as efficiently as possible, while still scaling up the network. For that, they factorized convolutions and used aggressive regularization techniques. This way they were able to surpass other states-of-the-art architectures.

Since Inception architecture is computationally efficient, this architecture is a good alternative to be used with SSD framework. This allowed SSD: Inception v2 to achieve 23.8 FPS with a mAP of 24% on COCO [64].

6.3.3 Faster R-CNN: Inception v2

As explained in Chapter 3, Faster R-CNN used a different path to detect object comparing to SSD and our framework, which are One-Stage Detectors. Due to this fact, it would be also interesting to compare their results with our experiments. Fusing Faster R-CNN with Inception v2 was possible to achieve 17.2 FPS, which is still considerably high, obtaining 28 % mAP on COCO [64].

6.3.4 Faster R-CNN: ResNet

ResNet [45] uses a residual learning framework, which makes easier the training process comparing to other deep networks. For that, they "*reformulate the layers as learning residual function with reference to the layer inputs, instead of learning unreferenced functions*". With this modification, they gained a considerable accuracy with the increase of depth.

To push further our comparisons and to observe better what generally happens when we increase the depth of a model, we also train two different versions of the Faster R-CNN system with this architecture. One with ResNet 50 we were able to obtain 11.2 FPS and 30 % mAP, and with ResNet 101 achieved 9.4 FPS and 32 % mAP on COCO [64].

6.3.5 Faster R-CNN: Inception ResNet v2

Recently, using a conjunction of ResNet and other more traditional architectures had improved substantially its performance. So, in [108] they associated ResNet and Inception v2, with this, they were able to obtain high accuracies. Using this fusion with Faster R-CNN framework achieves 1.6 FPS and 37 % mAP on COCO [64].

Chapter 7

Results and Discussion

After training the experiments described in subsections 6.2 and 6.3, it was performed a evaluation using Algorithm 1. This evaluation used 945 orignal images previously separated from the KITTIs dataset. With the results, it was then performed a comparison between the models' performances. At the end, it will be shown the biggest problems that the models faced.

7.1 Performance Testing

The frameworks were trained to detect two classes, cars and pedestrians, so it will be first evaluated the cars then the pedestrians and finally it will be shown the results of both detections at the same time. The evaluation will use two similar methods. First, it will be evaluated using KITTIs methodology an then the more commonly used PASCAL methodology.

To compare the results it is used a precision-recall curve and the mAP, which was considered as the optimal point where the maximum number of objects was detected with the maximum precision (mAP $\simeq$ mAR). The precision-recall curves have in solid lines represented the tweaked version (subsection 6.2), in dashed lines the other trained frameworks (subsection 6.3) and in dashed-dot line the Vanilla Version.

The following subsections will present Tables regarding the optimal points of the versions that were more interesting, however, Appendix **??** contains Tables with all the optimal points for each version. To better understand the Tables and Figures that will be presented its provided Table A.1 which contains the used model names and their references.

7.1.1 Detecting Cars

The Average precision-recall curve for detecting cars using KITTIs evaluation method can be observed in Figure 7.1. The majority of the tweaked versions (#**1.-**) performed worse than the control version #**1.1**, except for two tweak models, #**1.4** and #**1.5**, which had far better results than the others. However, the main problem in our models was the localization error, Table 7.2, being as high as 44.3% in the control version. Nonetheless, the best performance achieved an Average Precision (AP) of 46.2% (experiment #**1.5**).

Approach Name	**Reference**
Vanilla Version	#1.1
No Augmentation Data	#1.2
Batch Normalization	#1.3
Focal Loss	#1.4
Matching Strategy	#1.5
No transfer Learning	#1.6
Frozen layers	#1.7
Data Standardization	#1.8
k-means Cluster getting the anchors	#1.9
$\boldsymbol{\lambda}$ in function of IoU	#1.10
Lower Input	#1.11
SSD: MobileNet v1	#2.1
SSD: Inception v2	#2.2
Faster R-CNN: Inception v2	#3.1
Faster R-CNN: ResNet 50	#3.2
Faster R-CNN: ResNet 101	#3.3
Faster R-CNN: Inception ResNet v2	#3.4

Table 7.1: Models version reference.

Ref	**AP**	**AR**	**Localization Error**	**Background Error**	**Repetition Error**	**Optimal Confidence**
#1.1	43.4%	43.8%	41.3%	15.3%	0.0%	74%
#1.4	40.8%	41.5%	44.3%	14.9%	0.0%	73%
#1.5	46.2%	46.8%	37.8%	16.0%	0.0%	73%
#2.1	49.6%	49.7%	28.9%	21.5%	0.0%	22%
#2.2	59.6%	59.7%	19.2%	21.2%	0.0%	39%
#3.4	77.1%	77.4%	3.9%	18.9%	0.0%	98%

Table 7.2: Results obtained from detecting car in the optimal point, using KITTIs evaluation method.

Ref	AP	AR	Localization Error	Background Error	Repetition Error	Optimal Confidence
#1.1	69.5%	70.1%	13.5%	15.3%	1.7%	74%
#1.4	68.9%	70.0%	14.6%	14.9%	1.6%	73%
#1.5	70.6%	71.5%	12.3%	16.0%	1.1%	73%
#2.1	67.6%	67.8%	10.9%	21.5%	0.0%	22%
#2.2	70.9%	71.0%	7.7%	21.2%	0.1%	39%
#3.4	77.8%	78.1%	3.3%	18.9%	0.0%	98%

Table 7.3: Results obtained from detecting car in the optimal point, using PASCALs evaluation method.

The models trained afterwards had far better results than ours. Especially when using Faster R-CNN method (#**3.-**), however, they all obtained similar performances, which the highest performance was achieved by model #**3.4** with 77.1% AP, Table 7.2. SSD models (#**2.-**) had far different performances, this is due to the used architecture, achieving 49.6% AP with model #**2.1** and 59.6% AP with model #**2.2**, both performing better than our experiments.

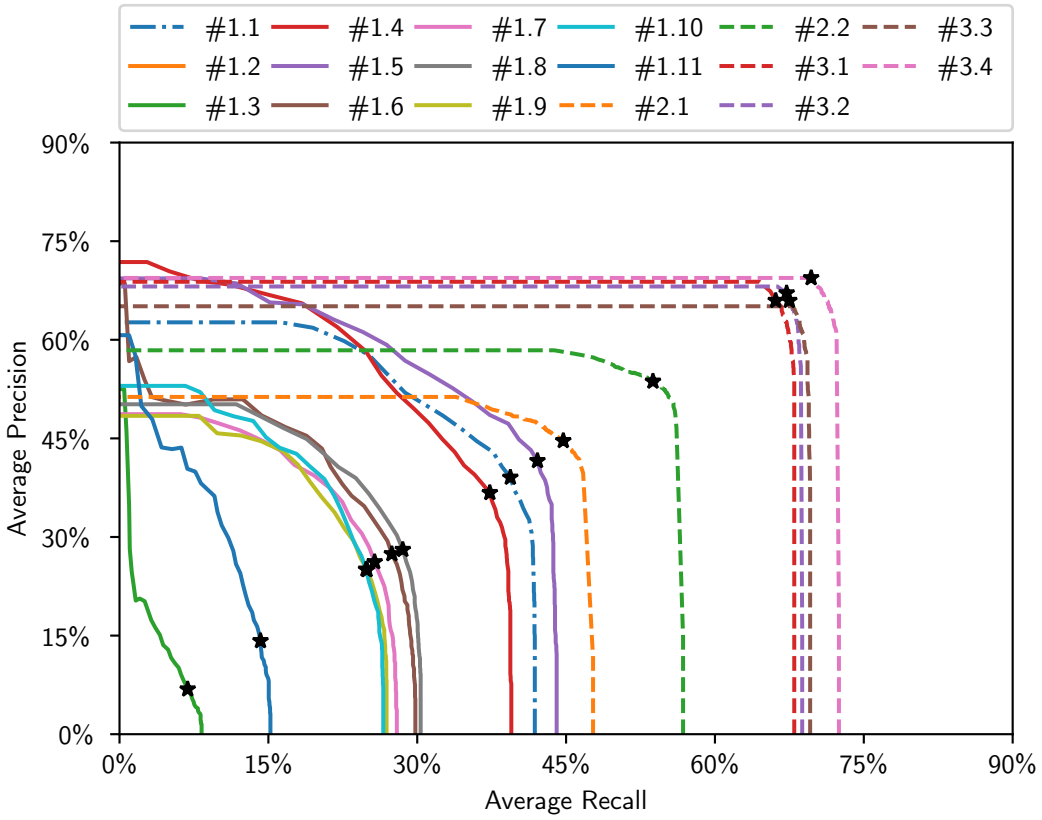


Figure 7.1: Average Precision-Recall curve for detecting Cars using KITTIs evaluation method, where * is referent to the optimal point.

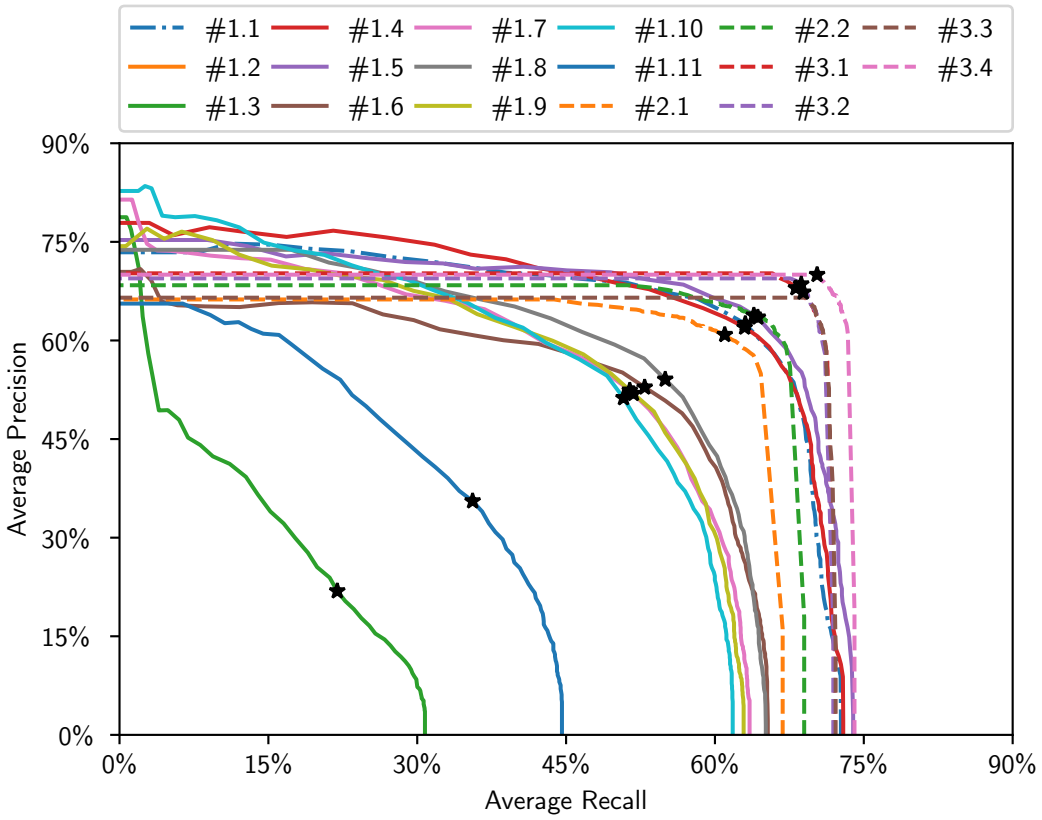


Figure 7.2: Mean Average Precision-Recall curve for detecting cars using KITTIs evaluation method, where * is referent to the optimal point.

Using PASCAL evaluation method, the results were, as expected, higher than using KITTIs method (Figure 7.2 and Table 7.3). Using this evaluation method, our top three models (#**1.1**, #**1.4** and #**1.5**) achieved higher AP than model #**2.1**, and the best experiment (#**1.5**), achieved 70.6 AP almost surpassing both SSD models (#**2.-**), in which model #**2.2** achieved 70.9%, only 0.3 % higher than ours. This is due to the decrease of the localization error, however, the repetition error increased. Yet, Faster R-CNN model achieved, again, higher performances than ours, which their best AP was 77.8% by model #**3.4**.

7.1.2 Detecting Pedestrians

KITTI and PASCAL evaluation methods used for detecting pedestrians are identical since both require an IoU of 50% or higher between the prediction and the pedestrian. Figure 7.3 and Table 7.4, contains the obtained results.

When detecting pedestrians, once more, Faster R-CNN models (#**3.-**) performed far better than any other tested detector, achieving up to 73.6% AP by model #**3.4**. However, the best result from our experiments (#**1.5**) achieved a similar performance to #**2.1**, obtaining 40.5% AP, only 0.2% less. The other experiments (#**1.-**) achieved a lower AP. The other SSD model, #**2.2**, had a far better performance than the experi-

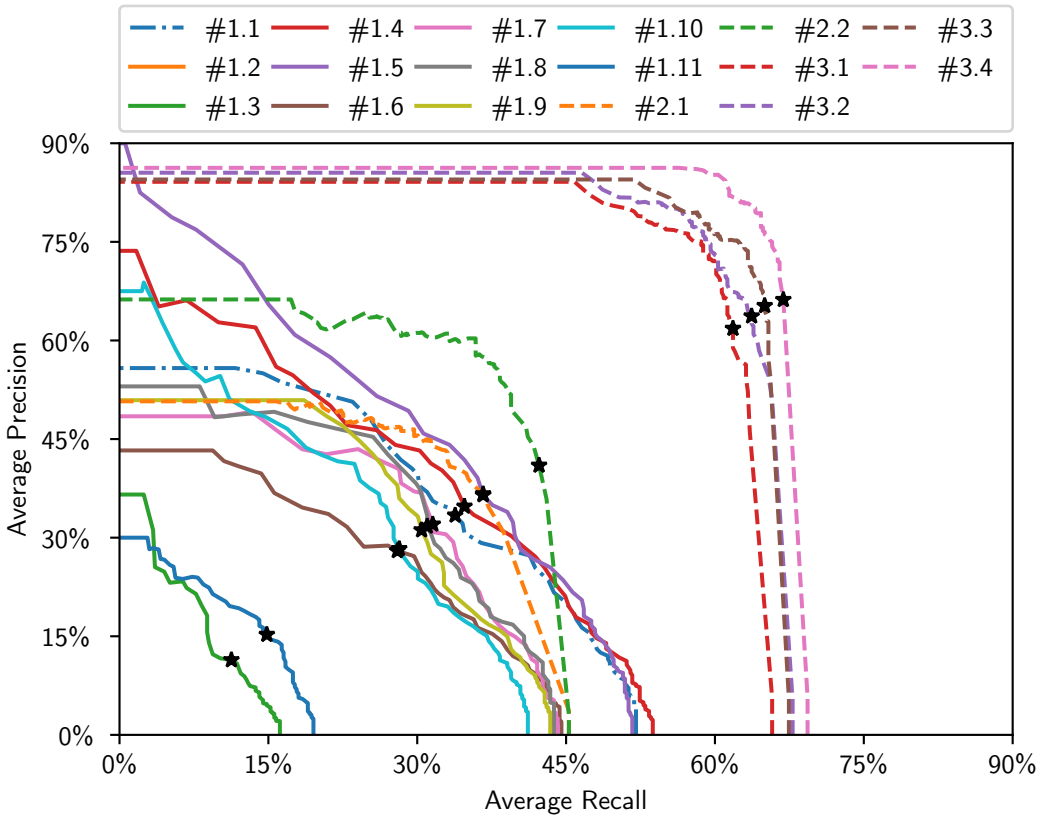


Figure 7.3: Average Precision-Recall curve for detecting pedestrians, where * is referent to the optimal point.

ments made, achieving at least more 5% AP.

7.1.3 Detecting Objects

The aim of this thesis is to detect cars and pedestrians, so this subsection is presented the joined results.

The Mean Average Precision-Recall curve for detecting cars and pedestrians using KITTIs evaluation method is shown in Figure 7.4. The experiment #**1.5** was clearly the best-tweaked model, achieving 43.2% mAP (Table 7.5), which the biggest problem was the localization error. However, the other also tested frameworks, Faster R-CNN and SSD, obtained far better performances, where #**3.4** achieved 76.6 mAP, the highest result among all, obtaining far less localization error.

The high localization error that we got might have happened because the initially weights used were previously trained to be a classifier. However, the other framework, R-CNN and SSD, used weights that were previously trained in COCOs dataset, which is an object detection dataset with 90 classes, including persons and cars. Anyway, the best experiment only performed 2% worse than #**2.1** model, which uses an architecture that is the closest to ours.

In Figure 7.5 and Table 7.6 it was used the PASCAL evaluation method, where the

Ref	AP	AR	Localization Error	Background Error	Repetition Error	Optimal Confidence
#1.1	37.1%	37.6%	33.8%	27.8%	1.2%	68%
#1.4	38.6%	38.6%	30.5%	30.1%	0.8%	65%
#1.5	40.5%	40.7%	29.3%	29.5%	0.6%	70%
#1.7	35.6%	35.1%	35.4%	25.2%	3.8%	68%
#2.1	40.7%	40.7%	23.2%	35.7%	0.4%	4%
#2.2	45.5%	47.0%	21.7%	32.2%	0.6%	3%
#3.4	84.7%	72.9%	3.4%	11.9%	0.0%	34%

Table 7.4: Results obtained from detecting pedestrians in the optimal point.

Ref	mAP	mAR	Localization Error	Background Error	Repetition Error	Optimal Confidence
#1.1	40.3%	39.7%	36.3%	23.0%	0.5%	72%
#1.4	39.8%	39.5%	37.1%	22.7%	0.4%	68%
#1.5	43.2%	43.9%	33.5%	23.1%	0.2%	71%
#2.1	45.2%	45.0%	26.5%	28.1%	0.2%	6%
#2.2	54.0%	53.8%	20.9%	24.8%	0.2%	6%
#3.4	76.6%	76.5%	5.0%	18.4%	0.0%	11%

Table 7.5: Results obtained from detecting car and pedestrians in the optimal point, using KITTIs evaluation method.

Ref	mAP	mAR	Localization Error	Background Error	Repetition Error	Optimal Confidence
#1.1	52.8%	53.9%	22.9%	23.0%	1.4%	72%
#1.4	54.1%	54.5%	23.1%	21.4%	1.3%	69%
#1.5	57.4%	55.7%	20.5%	21.3%	0.8%	72%
#2.1	54.7%	54.4%	17.6%	27.4%	0.3%	7%
#2.2	59.5%	59.9%	15.3%	24.8%	0.4%	6%
#3.4	77.1%	77.0%	4.5%	18.4%	0.0%	11%

Table 7.6: Results obtained from detecting car and pedestrians in the optimal point, using PASCALs evaluation method.

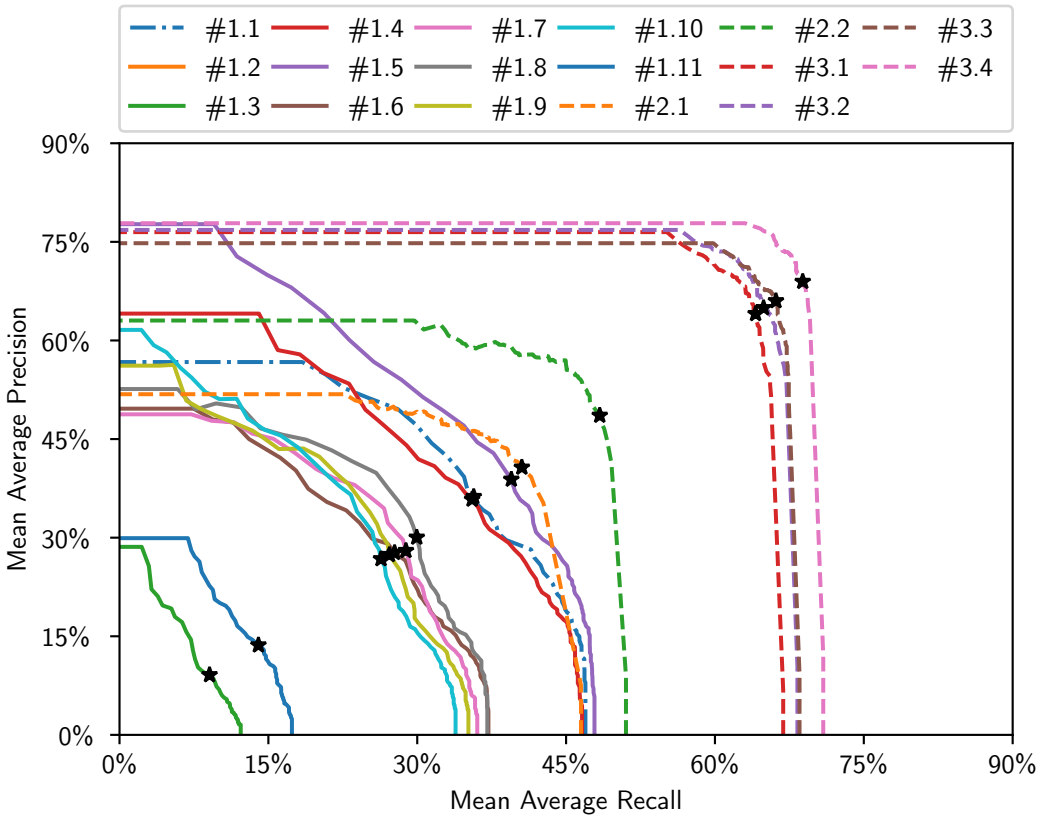


Figure 7.4: Mean Average Precision-Recall curve for detecting cars and pedestrians using KITTIs evaluation method, where * is referent to the optimal point.

best experiments (#**1.1** and #**1.5**) reached or even surpassed the mAP gotten by #**2.1**, reaching a maximum of 57.4 % mAP by #**1.5**. Even #**2.2** model, which uses a much deeper architecture, only obtained more 1.9% mAP. However, Faster R-CNN models, #**3.-**, achieved far higher precisions, surpassing up to 20% the best experiment, being the best models in terms of mAP and mAR.

7.1.4 System Performance

Object detectors implemented on self-driving cars must be able to work at high FPS while spending a low amount of GPU memory and disk space, this way not consuming all the hardware resources. So, during the experiments these parameters were tracked, as shown in Table 7.7. However, due to some incompatibilities with the GPU, which were caused by using an out-of-date CUDA [1] version, it was not possible to test Faster R-CNN performance.

During inference, our experiments achieved 46 FPS using the full image as input and 111 FPS using an image input of 375×375 pixel. However, SSD models that are known to be fast were only able to achieve between 5-6 FPS using this hardware, instead of the

[1]CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on GPUs.

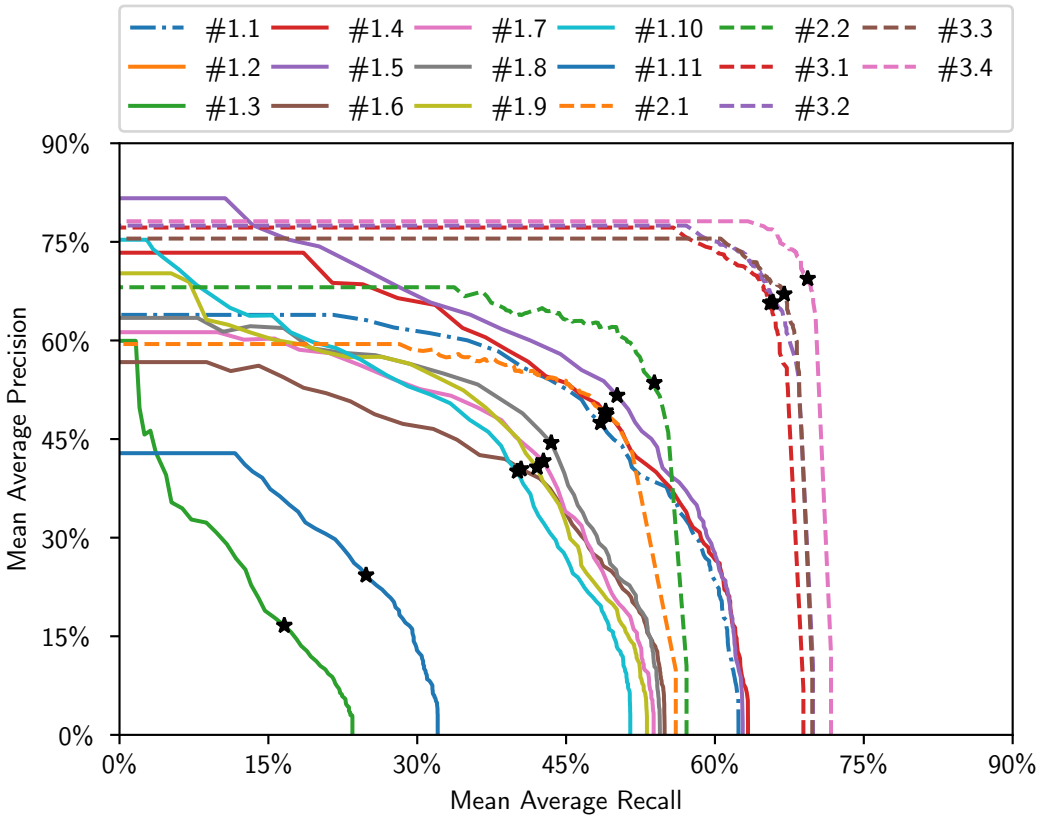


Figure 7.5: Mean Average Precision-Recall curve for detecting cars and pedestrians using PASCAL evaluation method, where * is referent to the optimal point.

23-33 FPS stated by the repository [2], which is not viable for a self-driving car. Since Faster R-CNN are slower than SSD framework, it was assumed that their frame rate is even lower, therefore, with respect to the frame rate, our framework is the only one viable for a self-driving car comparing to the other trained models.

When observing the memory needed by each model, ours are also the ones that spend less disk space, being at least 9 times smaller than any other trained framework. When analysing the GPU memory spend during inference time, we used 715 Megabyte (Mb) with our biggest models, a relatively small amount. Unfortunately, it was not possible to obtain the memory used by SSD frameworks due to the library used in its implementation.

7.1.5 Overall Performance

After analysing the previous results, we saw that the used evaluator has a huge impact on the measurement of mAP and mAR, as expected. If the minimum acceptable IoU is diminished to be a TP, then the localization error will decrease and the repetition error increases.

The majority of the experiments performed resulted in a lower performance, this will now be analysed.

[2]https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

Ref	FPS	GPU Memory (Mb)	Disk Space needed Memory (Mb)
#1.5	46	715	8.1
#1.11	111	445	8.1
#2.1	6	-	71.3
#2.2	5	-	165.7
#3.1	-	-	164.7
#3.2	-	-	410.0
#3.3	-	-	639.9
#3.4	-	-	634.3
#4	-	-	730.7

Table 7.7: Frame rate and memory needed for each tested system.

Version #1.2, in which was not performed data augmentation, did not obtain any viable results, this was expectable due to the CNNs being eager for data, proving once more, that data augmentation is extremely important to train CNNs.

Implementing batch normalization on SqueezeNet (**Version #1.3**) got a lower mAP then the control version (#**1.1**). This might have happened because it was implemented the batch normalization in the pre-trained network, and its weights did not suppose this change. Furthermore, it was also modified the task and dataset, which might have compromised the network's performance even more.

As expected, when it was not performed a transfer learning technique (**version #1.6**), the results were far worse than the control version. However, the technique Frozen Layers (**version #1.7**) trained much faster and used far less GPU memory during this phase than other experiments. This could be a good technique to be used when the GPUs memory.

In **version #1.8** it was performed a standardization to the data in order to help the learning task, but instead, the results were worse than without standardizing it. This might have happened because the used weights from SqueezeNet were not trained using this method, so instead, the learning task was more difficult.

Anchors plays a major roll in object detectors results, so we acquired our own using a k-cluster **version #1.9**), however, the performance was far worse than using the typical KITTIs anchors, which might have happened due to an extracting of the wrong features.

With the intent of thinking out-of-the-box, we tried to implement a dynamic balance variable λ in the loss function (**version #1.10**). However, it did not provide any gains, this might happened due its multiplication factor be too low. We believe that if we had used an approach similar to the Focal Loss, but applying it to the IoU, we might have been successful.

In the last experiment (**version #1.11**) which uses a smaller input data size it was obtained a lower precision as expected, however, due to having fewer parameters than the other networks, it trained faster and obtained higher FPS.

The best three versions (**versions #1.1, #1.4** and **#1.5**) surpassed all other tweaked models by at least 5% mAP. Experiment #**1.5**, due to its tweak, was able to train more predictors at the same time, allowing to achieve the best result amongst all tweaked versions. Experiment #**1.4** also obtained great results, especially in detecting

Model	FPS	mAP
MV3D [14]	2.22	79.54%
SubCNN [119]	0.5	79.2%
3DOP [13]	0.33	79.10%
Mono3D [12]	0.24	78.96%
SDP + RPN [89] [89]	2.5	78.38%
MV3D (LIDAR) [14]	3.33	78.16%
SINet VGG [50]	0.5	77.75%
Deep3DBox [73]	0.66	77.17%
SqueezeDet [116]	57.2	76.7%
#3.4	-	76.6%
SDP+CRF (ft) [120]	1.66	71.13%
Faster R-CNN [89]	0.5	71.12%
MS-CNN [9]	2.5	76.11%
Reinspect [105]	0.5	66.23%
3DVP [118]	0,03	65.38%
AOG [63] [116]	0.33	60.70%
spLBP [49]	0.66	60.59%
SubCat [76]	1.4	59.71%
#2.2	5	54.0%
YOLO 9000 [88]	33	50.25%
#2.1	6	45.3%
#1.5	46	43.2%
YOLO [87]	33	29.25%

Table 7.8: Result from various systems tested on KITTI dataset.

pedestrians due to the focal loss implementation, which focused on the hard examples and down-weighted the contribution of easy examples during training. We think that, probably, the joint of both versions may improve the object detector, resulting in an even better model.

To compare the best experiment (**version #1.5**) with other frameworks trained on KITTIs dataset, it is introduced the Table 7.8. The evaluations performed by the other models are from detecting cars, pedestrians and cyclists.

As said before, a self-driving car needs an object detector that can provide high frame rates, otherwise, it will not react in time to unexpected events. So, despite the majority of models in Table 7.8 present higher mAP than the other models, they performed at low frame rates, making them unviable to be implemented in a self-driving car. The models **#2.-** can also be excluded, since they performed at low FPS using the available hardware. Now looking at YOLO, this framework can achieve high speeds, however, the mAP is much lower than the obtained with our models, which is not beneficial.

The best results to be implemented in a self-driving car were obtain by YOLO 9000, our model (**#1.5**) and SqueezeDet, since they are real-time object detectors with a considerably high mAP. However, the achieved mAP was very different, the YOLO 9000 reached to 50.25% and SqueezeDet gotten 76.7%. We obtained 43.2% mAP, but we did not count the detected vans as a TP for the evaluation of our model, which might

Figure 7.6: Prediction of an KITTI test image, which wrongfully classified an cyclist as pedestrian.

decrease the resulted mAP.

SqueezeDet got a much higher mAP than the other models, however, the authors do not say how the evaluation was made. As other researchers [21] tried to reproduce SqueezeDet results and achieved far lower result, so we excluded this model from the comparison.

YOLO 9000 can reach a better mAP than our model, however, we can perform at high FPS, which could be important in a real-world scenario, but both are a viable solutions, depending on the trade-off that we prefer.

7.2 Predictions Visualization

This section presents some predictions that our best experiment (**version #1.5**) made, it is also shown some problems that made the mAP and mAR not being higher. The predictions analysed are made from the test dataset images.

As we can observe in Figure 7.6 and Figure 7.7, a common error that our system makes is in predicting that cyclists are pedestrians, this is probably due to the feature similarity between both classes. However, this not a desirable result and affects the mAP, increasing the background error. This also happens with vans which are wrongfully classified as cars (Figure 7.11), however, KITTIs considers this error a TP, but we do not.

Another factor that lowers the mAP was the missed or wrongfully labelled objects that some images contain, as shown in Figure 7.8 or in Figure 7.9. This will affect not only the evaluation of our model, increasing the localization and background error, but also our training, that could have been misleading.

During inference time, our model struggled to detect objects that overlapped with each other, as for example in parking lots. Since NMS discard predictors that overlap more than 50%, when two object overlap in similar or higher levels, we do not detect them, as Figure 7.10 shows, decreasing our mAR.

Figure 7.7: Prediction of an KITTI test image,which wrongfully classified an cyclist as pedestrian and detecting a hardly visible car.

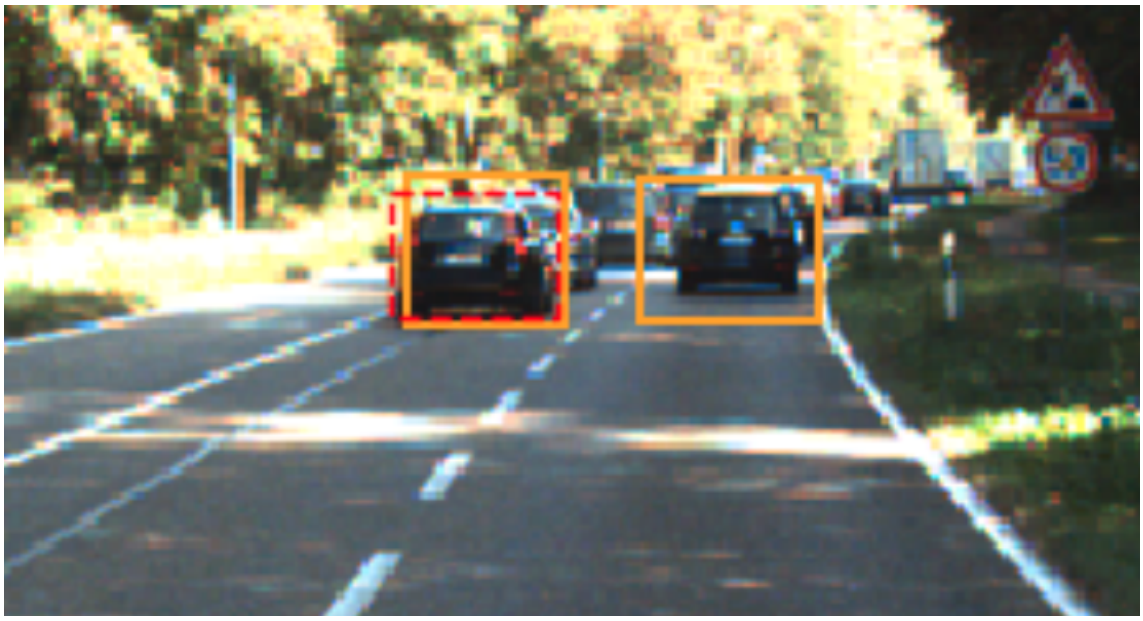


Figure 7.8: Prediction of an KITTI test image, which did not a cars GT.

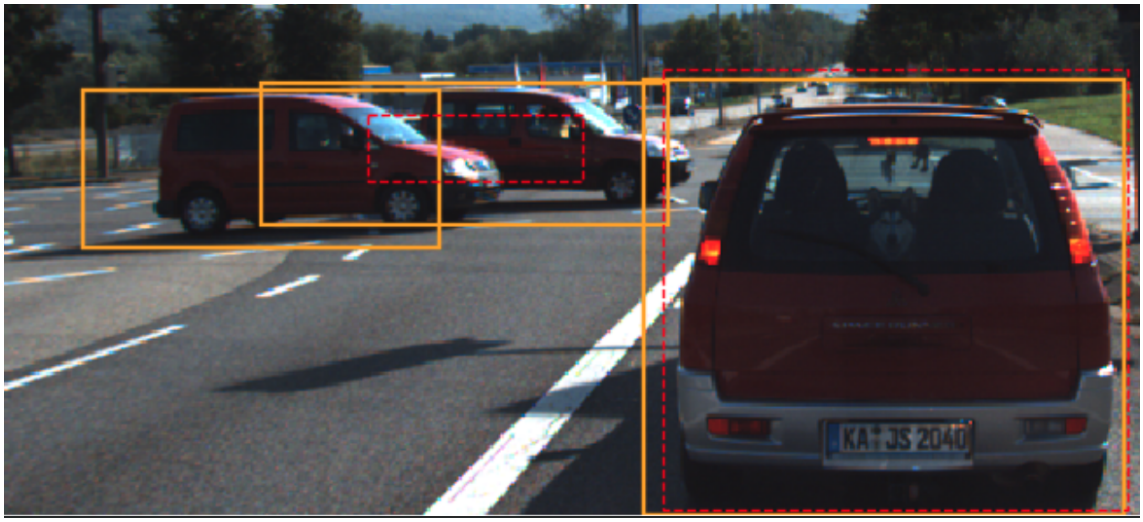


Figure 7.9: Prediction of an KITTI test image, wrong label on a Van.

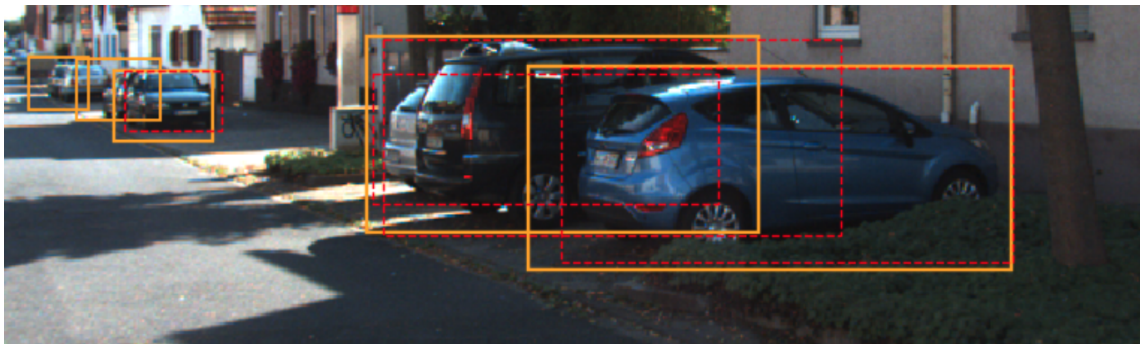


Figure 7.10: Prediction of an KITTI test image, overlapping of cars in a parking lot.

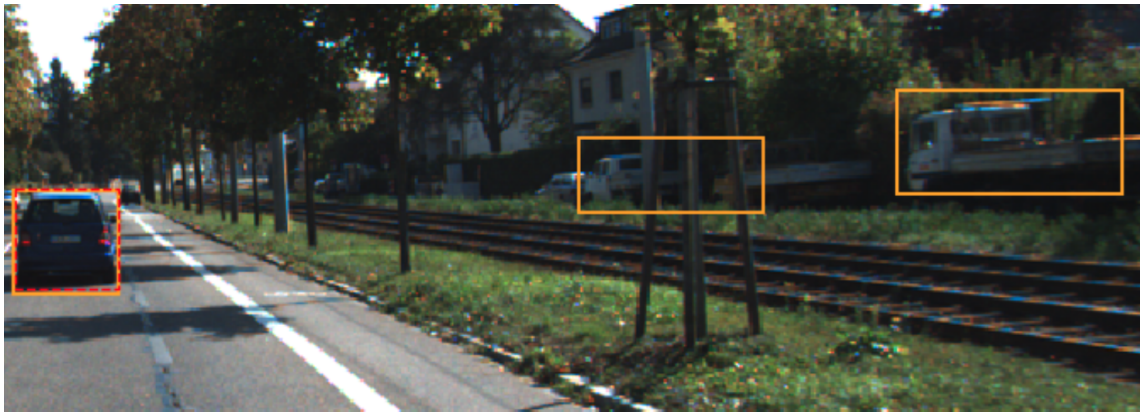


Figure 7.11: Prediction of an KITTI test image, miss classifying vans as cars.

Chapter 8

Conclusions

This chapter will contain a review of the obtained results and some concluding remarks. It is also explored some potential improvements that could be made to our system.

8.1 Theory

This thesis begun with a review of the theoretical background, explaining how ANNs work. Then it was described how CNN deal with the data and why are more effective in performing with images than classic ANNs.

Next, it was explored different ways to create an object detector, which had evolved from slower frameworks, using classic machine learning methods, as Viola and Jones [114] did it. Afterwards, R-CNN and Faster R-CNN appeared, which boosted object detection to a new level. These frameworks were able to achieve at the time impressive performances, however, the speed that they achieved was relatively low. Then appeared the One-Stage Detectors. Their main focus was to perform at high speeds, yet maintaining a considerable high accuracy.

Through time, One-Stage Detectors have evolved, becoming faster and more accurate, but generally they need the cutting edge hardware. So, we built our framework, inspired by these detectors, but trying to reduce the computational overload.

Since we built our system from scratch, one of the biggest difficulties was the information on how to do it, because the majority of the papers only explained the high-level concepts of their frameworks. However, looking through the code it was possible to recreate some parts of these detectors.

During the development of this thesis, it was learned that when building One-Stage Detectors there are two main parts: (1) the formulation of the loss function and (2) how to provide ground truth to our system.

8.2 Practice

During the creation of the framework, it was learned that the most challenging part is to implement the training phase. This is specially due to the implementation of the loss function, which is extremely difficult because it has to make all the operation using matrices to obtain full potential of the GPU. The GT is also extremely difficult due to all the transformations that are needed to made, using also matrices operations to

minimize possible bottlenecks. If it is made a mistake in the calculation during one of these steps is very difficult to debug it, due to the networks black-box effect and to the long training time. We overcame some of these problems using just one image during train and overfitting it, in order to diminish substantially the training time and made the debugging easier, but does not always work.

To training our models, we also must not forget to convert correctly our prediction to viable bounding boxes and pass them through a non-maximum suppression algorithm. Since the main goal is to obtain high speeds, these decoders and algorithms we must be implemented using matrices operation avoiding, as much as possible, bottlenecks.

When exploring already built object detectors we realized that most of them were straight out-of-the-box implementation and we would only need to provide the data with the correct labels and start training. However, these repositories generally are coded with multiple dependencies, which makes them hard to perform different experiments.

8.3 Results

The results obtained were promising since we were able to achieve high speeds (46 FPS), and still obtain a remarkable precision, 43.2% using KITTIs evaluation method. Even when comparing with other detectors as YOLO or SSD, we achieved a competitive framework. However, our models especially struggled to find the "exact" object localization, this being the biggest error that we found during our best model's evaluation.

During the experiments, the majority of the tweaks performed in the control version resulted in a decrease of precision, which in some cases was quite surprising, such as when we implemented a version with batch normalization or obtaining different anchors using a k-means cluster.

8.4 Future Work

One-Stage Detectors still have a lot of room to evolve and many experiments could be done. Even using a small model, such as we did, it is possible to obtain good results. As further work we would like to proceed in four different ways. Firstly, a study of the impact of hyper-parameters in the same model, in order to understand the impact of different elements during the training phase. Another interesting path would be modifying our loss function to decrease the localization error, building a loss that takes into account the IoU between the predictor and the ground truth. Despite our unsuccessful experiment, we believe that this would make a huge impact on the network performance. Another path would be performing multitasking learning using our framework and reusing its weights to perform a completely different task, as for example segmenting the road. Finally, we also suggest the implementation of this framework in ATLASCAR 2, as we think it would be an helpful contribute to its development.

Bibliography

[1] Amir Ahmad and Lipika Dey. A k-mean clustering algorithm for mixed numeric and categorical data. *Data & Knowledge Engineering*, 63(2):503–527, nov 2007.

[2] Timo Ahonen, Esa Rahtu, Ville Ojansivu, and Janne Heikkila. Recognition of blurred faces using Local Phase Quantization. In *2008 19th International Conference on Pattern Recognition*, pages 1–4, 2008.

[3] Imanol Bilbao and Javier Bilbao. Overfitting problem and the over-training in the era of data: Particularly for Artificial Neural Networks. In *2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*, number Icicis, pages 173–177. IEEE, dec 2017.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars. pages 1–9, apr 2016.

[6] Philip Brey. The strategic role of technology in a good society. *Technology in Society*, pages 1–7, 2016.

[7] Nikhil Buduma. *Fundamentals of Deep Learning : Designing Next-Generation Machine Intelligence Algorithms*. O'Reilly Media, 2017.

[8] Samuel Rota Bulo, Gerhard Neuhold, and Peter Kontschieder. Loss Max-Pooling for Semantic Image Segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7082–7091. IEEE, jul 2017.

[9] Zhaowei Cai, Quanfu Fan, Rogerio S. Feris, and Nuno Vasconcelos. A Unified Multi-scale Deep Convolutional Neural Network for Fast Object Detection. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9908 LNCS, pages 354–370. jul 2016.

[10] Weipeng Cao, Xizhao Wang, Zhong Ming, and Jinzhu Gao. A review on neural networks with random weights. *Neurocomputing*, 275:278–287, 2018.

[11] Joao Carreira and Cristian Sminchisescu. CPMC: Automatic Object Segmentation Using Constrained Parametric Min-Cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7):1312–1328, jul 2012.

[12] Xiaozhi Chen, Kaustav Kundu, Ziyu Zhang, Huimin Ma, Sanja Fidler, and Raquel Urtasun. Monocular 3D Object Detection for Autonomous Driving. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2147–2156, 2016.

[13] Xiaozhi Chen, Kaustav Kundu, Yukun Zhu, Huimin Ma, Sanja Fidler, and Raquel Urtasun. 3D Object Proposals using Stereo Imagery for Accurate Object Class Detection. *Advances in Neural Information Processing Systems*, pages 1–9, aug 2016.

[14] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3D Object Detection Network for Autonomous Driving. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6526–6534. IEEE, jul 2017.

[15] Chunhui Gu, Joseph J. Lim, Pablo Arbelaez, and Jitendra Malik. Recognition using regions. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, number 2, pages 1030–1037. IEEE, jun 2009.

[16] José Correia. *Unidade de Perceção Visual e de profundidade para o ATLASCAR2.* PhD thesis, University of Aveiro, 2017.

[17] George Dahl, Tara Sainath, and Geoffrey Hinton. Improving Deep Neural Netowrks for LVCSR Using Recitified Linear Units and Dropout, Department of Computer Science , University of Toronto. *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8609–8613, 2013.

[18] PDP Research Group David E. Rumelhart, James L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations.* 1986.

[19] Piotr Dollar, Zhuowen Tu, Pietro Perona, and Serge Belongie. Integral Channel Features. *Procedings of the British Machine Vision Conference 2009*, pages 91.1–91.11, 2009.

[20] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. *International Conference in Machine Learning (ICML)*, 32, oct 2014.

[21] Lucía Diego Solana Dónal Scanlan. *Deep Learning for Robust Road Object Detection.* PhD thesis, 2017.

[22] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Scalable Object Detection Using Deep Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2155–2162, 2014.

[23] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Scalable Object Detection Using Deep Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2155–2162. IEEE, jun 2014.

[24] Mark Everingham, S. M Ali Eslami, Luc Van Gool, Christopher K I Williams, John Winn, and Andrew Zisserman. The Pascal Visual Object Classes Challenge: A Retrospective. *International Journal of Computer Vision*, 111(1):98–136, 2014.

[25] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*, 88(2):303–338, jun 2010.

[26] Mark Everingham, Luc Van Gool, Christopher K.I. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010.

[27] P Felzenszwalb, David McAllester, Ross Girshick, and Deva Ramanan. Visual object detection with deformable part models. *Communications of the ACM*, 56(9):97, 2013.

[28] Pedro F Felzenszwalb, Ross B Girshick, David Mcallester, and Deva Ramanan. Object Detection with Discriminatively Trained Part Based Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1–20, sep 2009.

[29] Roger Fletcher. *Practical Methods of optimization*. Wiley, 1987.

[30] James E. Fowler. The redundant discrete wavelet transform and additive noise. *IEEE Signal Processing Letters*, 12(9):629–632, 2005.

[31] William T. Freeman and Edward H. Adelson. The Design and Use of Steerable Filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(9):891–906, 1991.

[32] Salvador García, Julián Luengo, and Francisco Herrera. *Data Preprocessing in Data Mining*, volume 72 of *Intelligent Systems Reference Library*. Springer International Publishing, Cham, 2015.

[33] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? The \textsc{KITTI} Vision Benchmark Suite. *Computer Vision and Pattern Recognition*, pages 3354–3361, 2012.

[34] Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2017.

[35] Ross Girshick. Fast R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, volume 2015 Inter, pages 1440–1448, 2015.

[36] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[37] Lee Gomes. Machine-Learning Maestro Michael Jordan on the Delusions of Big Data and Other Huge Engineering Efforts. *IEEE Spectrum*, (October):1–11, 2014.

[38] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[39] Andreas Griewank and Andrea Walther. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, jan 2008.

[40] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In José Mira and Francisco Sandoval, editors, *From Natural to Artificial Neural Computation*, pages 195–201, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[41] Bharath Hariharan, Pablo Arbeláez, Ross Girshick, and Jitendra Malik. Hypercolumns for object segmentation and fine-grained localization. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June:447–456, 2015.

[42] T Hastie, R Tibshirani, and J Friedmann. *The elements of statistical learning. Data mining, inference, and prediction*. 2001.

[43] Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5353–5360, 2015.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *Proceedings of the IEEE International Conference on Computer Vision*, 2015 International Conference on Computer Vision, ICCV 2015:1026–1034, 2015.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE, jun 2016.

[46] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *Arxiv*, pages 1–18, jul 2012.

[47] Andrew G. Howard. Some Improvements on Deep Convolutional Neural Network Based Image Classification. *arXiv*, pages 1–6, dec 2013.

[48] 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. apr 2017.

[49] Qichang Hu, Sakrapee Paisitkriangkrai, Chunhua Shen, Anton van den Hengel, and Fatih Porikli. Fast Detection of Multiple Objects in Traffic Scenes With a Common Detection Framework. *IEEE Transactions on Intelligent Transportation Systems*, 17(4):1002–1014, apr 2016.

[50] Xiaowei Hu, Xuemiao Xu, Yongjie Xiao, Hao Chen, Shengfeng He, Jing Qin, and Pheng-Ann Heng. SINet: A Scale-insensitive Convolutional Neural Network for Fast Vehicle Detection. pages 1–10, apr 2018.

[51] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–13, oct 2016.

[52] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *Iclr*, 9349:1–13, 2016.

[53] SAE international. U.S. Department of Transportation's New Policy on Automated Vehicles Adopts SAE International's Levels of Automation for Defining Driving Automation in On-Road Motor Vehicles. *SAE international*, page 1, 2016.

[54] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. feb 2015.

[55] Nathalie Japkowicz. The Class Imbalance Problem: Significance and Strategies. *Proceedings of the 2000 International Conference on Artificial Intelligence*, pages 111—-117, 2000.

[56] Jia Deng, Wei Dong, R. Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[57] S Jurecki. Driver's reaction time under emergency braking a car - research in a Driving simulator. *Polish Maintenance Society*, 14(4):295–301, 2012.

[58] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9, 2012.

[59] R. Kumari, Sheetanshu, M. K. Singh, R. Jha, and N.K. Singh. Anomaly detection in network traffic using K-mean clustering. In *2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*, pages 387–393. IEEE, mar 2016.

[60] Y LeCun and Y Bengio. Convolutional networks for images, speech, and time series. In *The handbook of brain theory and neural networks*, volume 3361, pages 255–258, 1995.

[61] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, R E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten Digit Recognition with a Back-Propagation Network. In D S Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 396–404. Morgan-Kaufmann, 1990.

[62] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[63] Bo Li, Tianfu Wu, and Song Chun Zhu. Integrating context and occlusion for car detection by hierarchical and-or model. *Lecture Notes in Computer Science*

(including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8694 LNCS(PART 6):652–667, 2014.

[64] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network. In *International Conference on Learning Representations*, pages 1–10, 2014.

[65] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In *CoRR*, volume abs/1405.0, pages 740–755. 2014.

[66] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9905 LNCS:21–37, 2016.

[67] Jiang-jing Lv, Xiao-hu Shao, Jia-shui Huang, Xiang-dong Zhou, and Xi Zhou. Data augmentation for face recognition. *Neurocomputing*, 230(July 2016):184–196, 2017.

[68] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier Nonlinearities Improve Neural Network Acoustic Models. *Proceedings of the 30 th International Conference on Machine Learning*, 28:6, 2013.

[69] Spyros Makridakis. The forthcoming Artificial Intelligence (AI) revolution: Its impact on society and firms. *Futures*, 90:46–60, jun 2017.

[70] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[71] Kishan Mehrotra, Chilukuri K Mohan, and Sanjay Ranka. Elements of artificial neural networks. *Complex adaptive systems*, pages 4–7, 9, 24–27, 39, 1997.

[72] Dmytro Mishkin, Nikolay Sergievskiy, and Jiri Matas. Systematic evaluation of convolution neural network advances on the Imagenet. *Computer Vision and Image Understanding*, 161:11–19, aug 2017.

[73] Arsalan Mousavian, Dragomir Anguelov, Jana Košecká, and John Flynn. 3D bounding box estimation using deep learning and geometry. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-January:5632–5640, 2017.

[74] Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning*, (3):807–814, 2010.

[75] J Nocedal and Sj J J Wright. *Numerical optimization*, volume 43. 1999.

[76] Eshed Ohn-Bar and Mohan Manubhai Trivedi. Learning to Detect Vehicles by Clustering Appearance Patterns. *IEEE Transactions on Intelligent Transportation Systems*, 16(5):2511–2521, oct 2015.

[77] M Oliveira and V Santos. Automatic Detection of Cars in Real Roads using Haar-like Features. *8th Portuguese Conference on Automatic Control*, (April 2015):1–6, 2008.

[78] Miguel Riem Oliveira. *Automatic Information and Safety Systems for Driving Assistance*. PhD thesis, University of Aveiro, 2013.

[79] Pariwat Ongsulee. Artificial Intelligence, Machine Learning and Deep Learning. In *Fifteenth International Conference on ICT and Knowledge Engineering*, pages 1–6, 2017.

[80] Adam Paszke, Gregory Chanan, Zeming Lin, Sam Gross, Edward Yang, Luca Antiga, and Zachary Devito. Automatic differentiation in PyTorch. *Advances in Neural Information Processing Systems 30*, (Nips):1–4, 2017.

[81] Josh Patterson: and Adam Gibson;. *Deep learning: A practictioners approach*, volume 521. 2015.

[82] Walter Murray Philip E. Gill and Margaret H. Wright. *Practical Optimization*. Academic Press, 1981.

[83] Pedro O Pinheiro, Ronan Collobert, and Piotr Dollar. Learning to Segment Object Candidates. pages 1–10, jun 2015.

[84] Kevin L. Priddy and Paul E. Keller. *Artificial Neural Networks: An Introduction*. SPIE, aug 2005.

[85] Vignesh Ramanathan, Jonathan Huang, Sami Abu-El-Haija, Alexander Gorban, Kevin Murphy, and Li Fei-Fei. Detecting Events and Key Actors in Multi-person Videos. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3043–3053. IEEE, jun 2016.

[86] Sebastian Raschka. *Python Machine Learning*. Number 1. Birmingham, 2015.

[87] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788. IEEE, jun 2016.

[88] Joseph Redmon and Ali Farhadi. YOLO9000: Better, Faster, Stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, number April, pages 6517–6525. IEEE, jul 2017.

[89] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, 2017.

[90] Shaoqing Ren, Kaiming He, Ross Girshick, Xiangyu Zhang, and Jian Sun. Object Detection Networks on Convolutional Feature Maps. pages 1–8, apr 2015.

[91] Willi Richert and Luis Pedro Coelho. *Building Machine Learning Systems with Python*. 2013.

[92] Rasmus Rothe, Matthieu Guillaumin, and Luc Van Gool. Non-maximum Suppression for Object Detection by Passing Messages Between Windows. *Asian Conference on Computer Vision (ACCV)*, pages 290–306, 2015.

[93] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, dec 2015.

[94] Stuart Russel and Peter Norvig. *Artificial Intelligence - A modern approach*, volume 74. 1995.

[95] Gerard Salton and Michael J. McGill. *Introduction to modern information retrieval.* McGraw-Hill Book Company, 1987.

[96] Dahlia Sam, Cyrilraj Velanganni, and T. Esther Evangelin. A vehicle control system using a time synchronized Hybrid VANET to reduce road accidents caused by human error. *Vehicular Communications*, 6:17–28, 2016.

[97] Sandhya Samarasinghe. Neural networks for applied sciences and engineering: from fundamentals to complex pattern recognition. page 582, 2007.

[98] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Research Journal*, 3(3):535–554, 1959.

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

[100] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):640–651, 2017.

[101] Abhinav Shrivastava, Rahul Sukthankar, Jitendra Malik, and Abhinav Gupta. Beyond Skip Connections: Top-Down Modulation for Object Detection. dec 2016.

[102] P.Y. Simard, D. Steinkraus, and J.C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, volume 1, pages 958–963. IEEE Comput. Soc, 2003.

[103] Bert Speelpenning. Compiling fast partial derivatives of functions given by algorithms. Technical report, Historical Energy Database (United States), jan 1980.

[104] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[105] Russell Stewart, Mykhaylo Andriluka, and Andrew Y. Ng. End-to-End People Detection in Crowded Scenes. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2325–2333. IEEE, jun 2016.

[106] Kah-kay Sung. *Center for Biological and Computational Learning and Example Selection for Object and Pattern Detection.* PhD thesis, Massachusetts Institute of Technology, 1996.

[107] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. 2016.

[108] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. feb 2016.

[109] Christian Szegedy, Scott Reed, Dumitru Erhan, Dragomir Anguelov, and Sergey Ioffe. Scalable, High-Quality Object Detection. 2014.

[110] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826. IEEE, jun 2016.

[111] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to data mining.* 2006.

[112] Tami Toroyan. Global status report on road safety. *World Health Orgainisation*, page 318, 2015.

[113] J R R Uijlings, K. E. A. van de Sande, T Gevers, and A W M Smeulders. Selective Search for Object Recognition. *International Journal of Computer Vision*, 104(2):154–171, sep 2013.

[114] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, 1:I–511–I–518, 2001.

[115] Min Wang, Baoyuan Liu, and Hassan Foroosh. Look-Up Table Unit Activation Function for Deep Convolutional Neural Networks. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1225–1233. IEEE, mar 2018.

[116] Bichen Wu, Forrest Iandola, Peter H Jin, and Kurt Keutzer. SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, volume 2017-July, pages 446–454, dec 2017.

[117] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep Image: Scaling up Image Recognition. 90(6):795–803, jan 2015.

[118] Yu Xiang, Wongun Choi, Yuanqing Lin, and Silvio Savarese. Data-driven 3D Voxel Patterns for object category recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June-2015:1903–1911, 2015.

[119] Yu Xiang, Wongun Choi, Yuanqing Lin, and Silvio Savarese. Subcategory-Aware convolutional neural networks for object proposals & detection. *Proceedings - 2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017*, pages 924–933, 2017.

[120] Fan Yang, Wongun Choi, and Yuanqing Lin. Exploit All the Layers: Fast and Accurate CNN Object Detector with Scale Dependent Pooling and Cascaded Rejection Classifiers. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2129–2137, 2016.

[121] Junfeng Yao, Yao Yu, and Xiaoling Xue. Sentiment prediction in scene images via convolutional neural networks. In *Proceedings - 2016 31st Youth Academic Annual Conference of Chinese Association of Automation, YAC 2016*, pages 196–200, 2017.

[122] Dong Yi, Zhen Lei, Shengcai Liao, and Stan Z. Li. Learning Face Representation from Scratch. nov 2014.

[123] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *NIPS'14 Proceedings of the 27th International Conference on Neural Information Processing Systems*, volume 2, pages 3320–3328, nov 2014.

[124] Roman Zakharenko. Self-driving cars will change cities. *Regional Science and Urban Economics*, 61(September):26–37, 2016.

[125] Matthew D. Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In *Computer Vision - ECCV 2014*, volume 8689, pages 818–833. 2014.

[126] M.D. Zeiler, M Ranzato, R Monga, M Mao, K Yang, Q.V. Le, P Nguyen, A Senior, V Vanhoucke, J Dean, and G.E. Hinton. On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521. IEEE, may 2013.

[127] Wang Zhiqiang and Liu Jun. A review of object detection based on convolutional neural network. In *2017 36th Chinese Control Conference (CCC)*, pages 11104–11109. IEEE, jul 2017.

[128] Xiaolu Zhou, Chen Xu, and Brandon Kimmons. Detecting tourism destinations using scalable geospatial analysis based on cloud computing platform. *Computers, Environment and Urban Systems*, 54:144–153, 2015.

Appendix A

Appendix

Approach Name	Reference
Vanilla Version	#1.1
No Augmentation Data	#1.2
Batch Normalization	#1.3
Focal Loss	#1.4
Matching Strategy	#1.5
No transfer Learning	#1.6
Frozen layers	#1.7
Data Standardization	#1.8
k-means Cluster getting the anchors	#1.9
$\boldsymbol{\lambda}$ in function of IoU	#1.10
Lower Input	#1.11
SSD: MobileNet v1	#2.1
SSD: Inception v2	#2.2
Faster R-CNN: Inception v2	#3.1
Faster R-CNN: ResNet 50	#3.2
Faster R-CNN: ResNet 101	#3.3
Faster R-CNN: ResNet 101 KITTI	#3.4
Faster R-CNN: Inception ResNet v2	#4

Table A.1: Models version reference.

Ref	AP	mR	Localization Error	Background Error	Repetition Error	Optimal Confidence
#1.1	43.4%	43.8%	41.3%	15.3%	0.0%	74%
#1.2	0.0%	0.0%	0.0%	0.0%	0.0%	-
#1.3	7.6%	7.6%	51.6%	40.8%	0.0%	77%
#1.4	40.8%	41.5%	44.3%	14.9%	0.0%	73%
#1.5	46.2%	46.8%	37.8%	16.0%	0.0%	73%
#1.6	30.5%	30.5%	48.8%	20.7%	0.0%	73%
#1.7	29.1%	28.6%	51.4%	19.5%	0.0%	73%
#1.8	31.2%	31.7%	52.3%	16.5%	0.0%	73%
#1.9	27.8%	27.7%	53.2%	19.0%	0.0%	73%
#1.10	27.9%	27.6%	53.8%	18.4%	0.0%	70%
#1.11	15.8%	15.8%	61.1%	23.1%	0.0%	62%
#2.1	49.6%	49.7%	28.9%	21.5%	0.0%	22%
#2.2	59.6%	59.7%	19.2%	21.2%	0.0%	39%
#3.1	73.3%	73.5%	6.1%	20.6%	0.0%	93%
#3.2	74.6%	74.7%	5.2%	20.2%	0.0%	95%
#3.3	73.3%	75.0%	5.1%	21.7%	0.0%	99%
#3.4	73.7%	76.7%	5.4%	20.9%	0.0%	99%
#4	77.1%	77.4%	3.9%	18.9%	0.0%	98%

Table A.2: Results obtained from detecting car in the optimal point, using KITTIs evaluation method.

Ref	AP	AR	Localization Error	Background Error	Repetition Error	Optimal Confidence
#1.1	69.5%	70.1%	13.5%	15.3%	1.7%	74%
#1.2	0.0%	0.0%	0.0%	0.0%	0.0%	-
#1.3	24.3%	24.4%	32.8%	40.8%	2.1%	77%
#1.4	68.9%	70.0%	14.6%	14.9%	1.6%	73%
#1.5	70.6%	71.5%	12.3%	16.0%	1.1%	73%
#1.6	58.8%	58.8%	18.1%	20.7%	2.4%	73%
#1.7	58.2%	57.1%	19.9%	19.5%	2.4%	73%
#1.8	60.1%	61.1%	20.6%	16.5%	2.8%	73%
#1.9	57.7%	57.5%	21.1%	19.0%	2.3%	73%
#1.10	57.0%	56.4%	22.8%	18.4%	1.9%	70%
#1.11	39.5%	39.5%	36.0%	23.1%	1.4%	62%
#2.1	67.6%	67.8%	10.9%	21.5%	0.0%	22%
#2.2	70.9%	71.0%	7.7%	21.2%	0.1%	39%
#3.1	75.5%	75.7%	3.8%	20.6%	0.0%	93%
#3.2	76.2%	76.3%	3.6%	20.2%	0.0%	95%
#3.3	74.8%	76.5%	3.5%	21.7%	0.0%	99%
#3.4	74.9%	78.0%	4.2%	20.9%	0.0%	99%
#4	77.8%	78.1%	3.3%	18.9%	0.0%	98%

Table A.3: Results obtained from detecting car in the optimal point, using PASCALs evaluation method.

Ref	AP	AR	Localization Error	Background Error	Repetition Error	Optimal Confidence
#1.1	37.1%	37.6%	33.8%	27.8%	1.2%	68%
#1.2	0.0%	0.0%	0.0%	0.0%	0.0%	-
#1.3	12.7%	12.5%	22.6%	64.1%	0.6%	77%
#1.4	38.6%	38.6%	30.5%	30.1%	0.8%	65%
#1.5	40.5%	40.7%	29.3%	29.5%	0.6%	70%
#1.6	31.1%	31.1%	32.8%	34.2%	1.9%	73%
#1.7	35.6%	35.1%	35.4%	25.2%	3.8%	68%
#1.8	35.3%	34.4%	33.5%	28.0%	3.2%	69%
#1.9	34.6%	33.8%	35.3%	26.5%	3.6%	66%
#1.10	31.4%	31.3%	39.4%	26.0%	3.1%	64%
#1.11	17.0%	16.5%	44.8%	37.8%	0.4%	47%
#2.1	40.7%	40.7%	23.2%	35.7%	0.4%	4%
#2.2	45.5%	47.0%	21.7%	32.2%	0.6%	3%
#3.1	68.7%	68.7%	8.8%	22.5%	0.0%	12%
#3.2	70.8%	70.8%	8.6%	20.3%	0.4%	11%
#3.3	72.5%	72.2%	11.3%	16.1%	0.0%	10%
#3.4	84.7%	72.9%	3.4%	11.9%	0.0%	34%
#4	73.6%	74.3%	6.4%	19.8%	0.2%	2%

Table A.4: Results obtained from detecting pedestrians in the optimal point.

Ref	mAP	mAR	Localization Error	Background Error	Repetition Error	Optimal Confidence
#1.1	40.3%	39.7%	36.3%	23.0%	0.5%	72%
#1.2	0.0%	0.0%	0.0%	0.0%	0.0%	-
#1.3	10.1%	10.1%	37.1%	52.5%	0.3%	77%
#1.4	39.8%	39.5%	37.1%	22.7%	0.4%	68%
#1.5	43.2%	43.9%	33.5%	23.1%	0.2%	71%
#1.6	30.8%	30.8%	40.8%	27.5%	0.9%	73%
#1.7	31.1%	32.1%	43.2%	23.7%	2.0%	70%
#1.8	33.4%	33.3%	42.5%	22.4%	1.7%	71%
#1.9	30.4%	30.2%	43.3%	24.4%	1.9%	69%
#1.10	29.7%	29.3%	45.9%	23.0%	1.4%	67%
#1.11	15.2%	15.6%	52.4%	32.1%	0.3%	53%
#2.1	45.2%	45.0%	26.5%	28.1%	0.2%	6%
#2.2	54.0%	53.8%	20.9%	24.8%	0.2%	6%
#3.1	71.2%	71.2%	7.3%	21.5%	0.0%	30%
#3.2	72.2%	72.1%	7.2%	20.5%	0.1%	23%
#3.3	73.4%	73.5%	7.7%	19.0%	0.0%	42%
#3.4	76.6%	76.5%	5.0%	18.4%	0.0%	11%

Table A.5: Results obtained from detecting car and pedestrians in the optimal point, using KITTIs evaluation method.

Ref	mAP	mAR	Localization Error	Background Error	Repetition Error	Optimal Confidence
#1.1	52.8%	53.9%	22.9%	23.0%	1.4%	72%
#1.2	0.0%	0.0%	0.0%	0.0%	0.0%	-
#1.3	18.5%	18.5%	27.7%	52.5%	1.4%	77%
#1.4	54.1%	54.5%	23.1%	21.4%	1.3%	69%
#1.5	57.4%	55.7%	20.5%	21.3%	0.8%	72%
#1.6	44.9%	45.0%	25.5%	27.5%	2.1%	73%
#1.7	46.3%	47.5%	28.1%	22.3%	3.3%	71%
#1.8	49.4%	48.3%	26.9%	20.9%	2.8%	72%
#1.9	45.2%	46.7%	29.2%	22.4%	3.2%	70%
#1.10	44.5%	44.5%	31.7%	21.3%	2.5%	68%
#1.11	27.0%	27.6%	44.5%	27.5%	1.0%	57%
#2.1	54.7%	54.4%	17.6%	27.4%	0.3%	7%
#2.2	59.5%	59.9%	15.3%	24.8%	0.4%	6%
#3.1	73.0%	72.9%	5.8%	21.2%	0.0%	31%
#3.2	73.1%	73.2%	6.2%	20.5%	0.1%	23%
#3.3	74.5%	74.4%	6.6%	19.0%	0.0%	43%
#3.4	77.1%	77.0%	4.5%	18.4%	0.0%	11%

Table A.6: Results obtained from detecting car and pedestrians in the optimal point, using PASCALs evaluation method.