BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Public transport vehicle detection based on deep learning

Supervisor
Lect. dr. Borza Diana-Laura

Author
Comănac Dragoș-Mihail

2022

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

Detectarea vehiculelor de transport public bazată pe învățare profundă

Conducător științific
Lect. dr. Borza Diana-Laura

Absolvent
Comănac Dragoș-Mihail

2022

ABSTRACT

Currently, there are billions of people that suffer from some form of visual impairment, out of which a significant part is legally blind. Also, a basic human need is mobility, but there aren't enough traditional mobility solutions for all visually impaired persons, such as assistance dogs, thus, for most legally blind people this basic human need can't be easily satisfied. A more scalable solution would be a digital one, which involves computer vision. This is possible because recent advances in hardware enable complex computations on small, mobile devices, and computer vision has emerged as a key field of artificial intelligence because it aims to replicate the human visual cortex. This kind of solution could be easily distributed to the millions of legally blind people because a mobile phone is much cheaper and more common than an assistance dog, for instance.

Therefore, the main purpose of this thesis is to provide a form of mobile assistive technology for visually impaired persons that uses an object detection system. The idea is that the object detector takes as input live images from the mobile device's camera and outputs bounding boxes that describe surrounding objects. Finally, this information is converted to sound and played using the mobile device's speakers. In this way, visually impaired persons can gather valuable information about the environment and travel through it.

Our object detector is implemented along the lines of You Only Look Once, which is a very performing object detection architecture. We implement a single convolutional neural network that takes an image and outputs the bounding boxes, together with their classes and scores. We use transfer learning with MobileNet as the body of the object detector, and we train the head from scratch, using a GPU, on a subset of Open Images V4 dataset composed of bus, car, and license plate. Also, we have developed an Android mobile application that uses this object detector in order to visualize the bounding box predictions. The key feature of the application is the accessible live object detection, in which the predictions are not visualized, but converted to sound and played using the mobile device speakers. This is the proof of concept for the assistive technology that uses computer vision, on low-budget hardware, such as a mobile phone.

In the end, we obtain, on the test set, a mean average precision of 70.03%, and for the bus class, we obtain an average precision of 90.01%, for the car class 64.04% and for the license plate 56.68%, with a speed of around 5 FPS on a mobile device. We have also trained a model on the COCO dataset that achieves around 0.4% mAP on the test dataset. The result is modest, but the object detection system was built around the Open Images V4 dataset, and we tried to obtain a model that is as small as possible in order to consume few resources.

Contents

1. Introduction

According to the World Health Organization [28], the number of people suffering from some moderate to severe form of distance vision impairment or blindness due to cataract or uncorrected refractive error is around 200 million, out of the total of 2.2 billion people worldwide estimated to have some form of visual impairment. This represents a significant segment of the population that has trouble performing daily tasks. These troubles can be alleviated using assistive technologies (AT) that can help persons with disabilities maintain or enhance their capabilities.

Computers can extract and interpret meaningful information from digital images or videos using Computer Vision (CV), which is a multidisciplinary field of artificial intelligence (AI) and deep learning (DL). It can also be regarded as a way of replicating the functions of the eye and the human visual cortex, or the area of the brain that processes visual information. So far, the functions of the eye are replicated, even exceeded, quite well by modern cameras. The functions of the visual cortex prove to be more difficult to replicate, but modern AI models can perform well on very specific tasks, sometimes rivaling human image recognition capabilities.

Traditionally, running a complex enough deep learning model required so many resources that it could only run on large and expensive equipment. But, as technology advances, as predicted by Moore's law, powerful devices become more compact and are accessible to more people. Nowadays, phones have the needed hardware, such as integrated high-resolution cameras and fast CPUs, to use AI models to extract and interpret information about the surrounding environment. This means that computer vision can be integrated into daily activities.

Given the number of people that suffer from some form of visual impairment and the fact that computers can substitute visual functionalities, computer vision has the potential to play the main part of assistive technology for visually impaired persons (VIP), such that it helps the user to better understand the surrounding environment when performing different kinds of tasks. It is worth noting that this kind of technology can also prevent accidents. Also, a bonus is that anyone can have a phone, which is relatively cheaper and easier to use, compared to other specialized devices that might not even be available in all countries.

1.1 Objectives

One of the fundamental human needs is mobility and one important way of achieving it is public transport. This way of traveling is especially important to the VIP, since they are not allowed to drive. Therefore, the main ways a VIP can travel is by public transport, ridesharing, or taxi, but they experience many difficulties on their journeys, often experiencing social exclusion because they are limited in their choices of public transport [26].

Given that most mobility solutions don't provide adequate accessibility facilities, our aim is to develop **a mobile assistive technology solution** for VIP that provides spatial information by using a **real-time object detection model** inspired by You Only Look Once [29]. More specifically, the user can be guided toward the bus or car by the auditory information provided by the mobile application. All this information is extracted from the bounding boxes predicted by the object detection model. Also, information about the license plates can be provided to help the user to identify the vehicle.

Concretely, our objectives are to study existent object detection models, and to implement one that can be run in a reasonable time on a mobile device. We also aim to extract information about the license plate through an external optical character recognition API.

In the end, our main achievements are that we implemented and trained the object detection system, and that we composed from scratch the pipeline that takes live images and outputs the bounding boxes in the form of audio, only on the mobile device. On the test set, we obtain, a mean average precision of 70.03%, and for the bus class, we obtain an average precision of 90.01%, for the car class 64.04% and for the vehicle registration plate 56.68%, with a speed of around 5 FPS on a mobile device.

1.2 Thesis structure

In what follows, we first describe the current approaches in terms of public transport accessibility and object detection in Chapter 2. Afterwards, we detail the theoretical foundations relevant to our approach in Chapter 3. Then, in Chapter 4 we present the design and implementation details of our approach by detailing our object detector and our mobile application. Finally, we discuss the performance of our object detector and how it is compared to other object detection systems in Chapter 5. The user manual for our mobile application is detailed in Appendix A and further details about our results are represented in Appendix B.

2. Existing methods for public transport accessibility and object detection

In what follows we will emphasize what are the current options the VIPs have when it comes to public transportation. Also, we will review some of the most performing object detection systems that are currently used in various domains.

2.1 Accessible public transportation applications for VIP

In this section, we will explore two approaches of improving the accessibility of public transportation. Firstly, we will review a classical method based on radio signals, then more of a lightweight computer vision based method.

2.1.1 Radio frequency approach

The current approach to making busses more accessible to the VIP is a solution based on Radio Frequency Identification (RFID) [7]. Basically, it uses wireless radio frequency transmissions to transfer data between two devices. The system is composed of four main parts: the device held by the VIP which is compatible with a braille keyboard, the bus station controller, a device on the bus, and a database. The VIP searches the bus number and destination using his device. Then the bus controller sends signals to all busses in the radio frequency area, then passes back to the user the information about incoming busses, through a vocal message.

One disadvantage of this method is the complex infrastructure required. Basically, the bus, station, and the user are dependent upon each other. This makes it hard to implement it on a large scale and it does not guide the user toward the bus. Also, it requires expensive equipment for the user such as a braille keyboard, which is around 1000 dollars, an RFID reader which is around 100 dollars, and a personal digital assistant, which is around 20 dollars. The hardware needed for stations and busses is cheaper by a lot (around 1 dollar per bus or station), but it wouldn't be feasible because of the integration costs.

2.1.2 Computer vision approach

Another approach would be to make the user independent of any infrastructure. This should give the user the freedom to access any bus, in any city, regardless if there are any accessibility features available or not. This can be achieved by using a mobile version of a computer vision model that tells the user where the bus is located by using real-time object detection. This way there is no need for extra equipment on busses or stations.

Using this approach, RenewSenses, an Israeli company that develops assistive technologies, together with Omdena, created Travis [32], which is an Android add-on that is simply connected to the smartphone and uses its computing power to execute several computer vision tasks, mainly object detection.

The basic idea is that a piece of extra hardware equipment that contains a camera is attached to the smartphone which in turn provides to the user multi-sensory information about the surrounding environment through the phone vibrations and a vision to sound algorithm.

This system can also detect bus lines and warns the user about nearby obstacles.

We aim to develop a system along these lines but using only the built-in phone camera which feeds information to an object detector.

2.2 Object detectors

Typically, object detection is solved using two approaches: two-stage object detectors and one-stage object detectors.

Initially, the task of object detection was decomposed into multiple tasks, that together made a pipeline, which is hard to train. For example, region-based object detectors such as R-CNN [10] and its faster variants first generate bounding boxes through selective search, then a convolutional network extracts features that are classified by a support vector machine. All these steps slow down performance. This approach is called two-stage object detection.

Single-shot object detectors achieve real-time speed with decent accuracy because their detection pipeline consists only of one neural network that processes the image and directly outputs the predictions. This approach used to have low accuracy, but recent advances have made the single-shot detectors rival the two-stage detectors in terms of accuracy, without losing speed. Thus, we will focus especially on the single-shot detector class of object detection models.

2.2.1 You only look once

You only look once (YOLO) [29] is a single-shot object detection system that achieves real-time performance. Instead of a long and complex detection pipeline, in YOLO, the object detection problem is treated as a regression. There is only one neural network that predicts the bounding boxes and class probabilities from an image. This way, the network can benefit from using end-to-end learning, and the inference time is greatly reduced, thus achieving real-time performance.

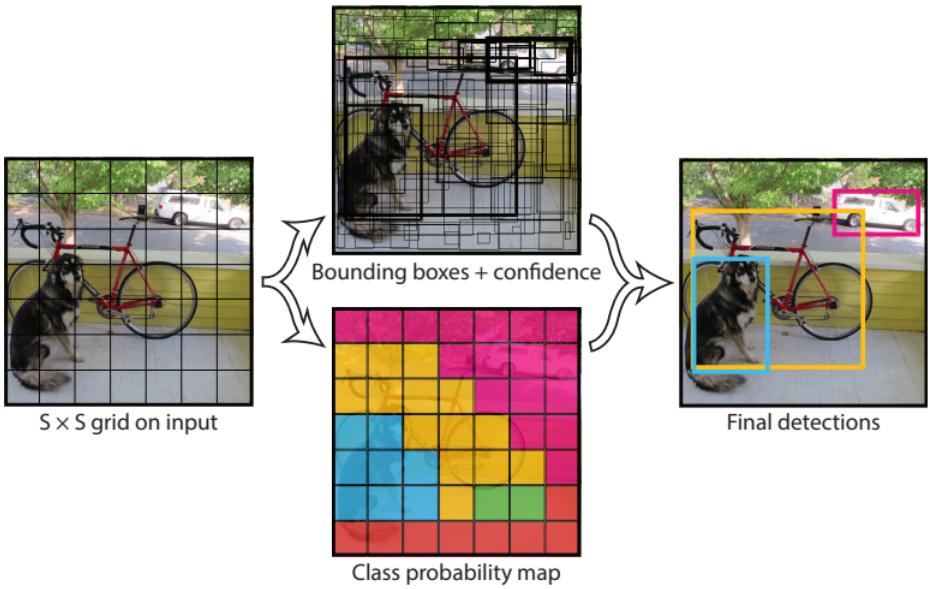


Figure 2.1: Detection pipeline of the first version of YOLO [29]

As it is depicted in 2.1, the image is split into an SxS grid, each cell being responsible for the detection of one object. For each cell, there are B bounding boxes predicted, and for each box, the position of the center, relative to the grid cell is predicted, the width and height are predicted relative to the image and a confidence score is predicted indicating if there is an object in that box. Also, C class probabilities, conditioned on the cell containing an object, are predicted for each cell, regardless of the number of boxes B. Therefore, the network outputs a tensor of shape $S \times S \times (B * 5 + C)$. In the second version of YOLO [30], the notion of anchors is introduced. This changes the output to be $S \times S \times B \times (5 + C))$, where B is the number of anchors. This allows the model to detect multiple objects in the same grid cell, with multiple shapes and sizes. This is the approach we have followed, and it is explained in more detail in a later section.

The network proposed by the authors is inspired by the GoogLeNet model [40]. It has 24 convolutional layers followed by 2 fully connected layers, although a faster version contains only 9 convolutional layers and fewer filters.

Leaky rectified linear activation function is on all layers, except the last one,

where a linear activation function is used. Sum squared error is used for the loss function, but the authors note that it is not ideal because localization errors and classification errors are treated the same. Also, the cells that do not contain any images will have confidence scores close to zero and can overcome the gradients from the cells that do contain objects, causing the training to diverge. To prevent this, two parameters are introduced. λ_{coord} and λ_{noobj} increase or decrease the coordinate prediction loss and confidence prediction loss respectively.

When training, for each cell, only one bounding box is chosen, the one with the highest input over union (IOU) with the ground truth. This way recall is improved because the bounding box predictors perform better on certain sizes, aspect ratios, or classes.

At test time, YOLO achieves great speeds because it only needs a single pass through the network. Non-maximal suppression is used to choose between different bounding boxes.

Because each cell predicts only one object, multiple clustered small objects are harder to detect, thus the main source of error is localization error.

One important metric in measuring the performance of object detection systems is mean average precision (mAP). Precision measures the percentage of correct predictions for a given class. A prediction is considered true positive if the IOU is greater than a set threshold. Recall is another metric that measures the percentage of found positives. Both precision and recall depend on the given threshold for true positives. This means that by plotting different values for precision and recall for different thresholds we get a curve. The area under the curve is called average precision. And the mean over the area under the precision-recall curve for all classes is the mean average precision.

According to [29], in terms of performance on the Pascal VOC 2007 dataset [8], the first version of YOLO achieves 63.4% mAP and 45 frames per second (FPS), and the faster version has 52.7% mAP and 155 FPS. For comparison, Faster R-CNN [31] achieves 73.2% mAP but 7 FPS, which is accurate but very slow, and the Deformable parts model [34] achieves 100 FPS but it has 16% mAP, or a slower variant achieves 26.1% mAP at 30 FPS. Therefore YOLO strikes a good balance between speed and accuracy.

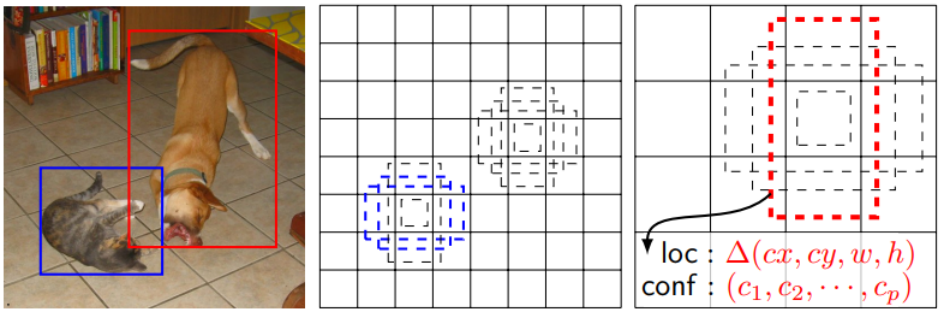
Newer versions of YOLO bring some incremental improvements. Also, the Microsoft COCO dataset [42] is another relevant dataset for comparing object detection systems. On this dataset, YOLOv4 [1] achieves 43.9% mAP at 31 FPS, or a smaller variant achieves 38 FPS, with 41.2% mAP.

2.2.2 Single Shot MultiBox Detector

Single shot MultiBox Detector (SSD) [24] is another example of an object detection system that achieves real-time performance, encapsulating all operations in a single deep neural network. Like YOLO, this helps SSD to outperform previous approaches that use multiple stages in detection such as R-CNN.

The network is composed of two parts. The first one consists of what is called the base network, which is a truncated version of an image classifier, where the classification layers are removed, that is used to extract features. On top of the base network, several structures specific to object detection are added.

The key features of SSD are the multi-scale feature maps, convolutional predictors, and default boxes.



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 2.2: Visualization of the feature maps used in SSD [24]

To the base network, there are appended several convolutional layers that decrease progressively in size the feature map from the base network. This way predictions are made for each newly added layer, therefore, the predictions are made at various scales, as it is depicted in 2.2.

Each cell in each feature map has associated K default bounding boxes, whose positions are relative to their cell. Then for each box several kernels of size $3 \times 3 \times P$, where $M \times N$ is the size of the feature map and P is the number of channels, are used to predict C class probabilities and 4 offsets to the respective box. This means that each box uses $(C+4) \cdot K$ filters, therefore the size of the predictions is $(C+4) \times K \times M \times N$.

During training, the outputs need to be assigned to their corresponding ground truths. Then the loss function and backpropagation are applied end-to-end. Furthermore, the set of default boxes and scales is chosen, and hard negative mining and data augmentation strategies are used.

For matching the outputs, each ground truth box is associated with the default box with the highest Jaccard overlap. The novel approach here is that the default boxes are also matched with any ground truth box with a Jaccard overlap higher

than a threshold. This allows predictions with high scores for multiple overlapping default boxes.

The loss is a weighted sum between confidence loss and localization loss.

Another crucial part is choosing scales and aspect ratios for the default boxes. Each feature map has a specific scale, distributed evenly between 2 values. The aspect ratios are chosen from a predefined set. The width and height are computed using the scale and the aspect ratio. The center is chosen based on the feature map cell size.

The matching steps produce more negatives than positives. This introduces an imbalance, and to fix this, the negatives are filtered by their confidence loss so that a ratio of 3 to 1 is kept between the negatives and positives.

Also, data augmentation is used. During training for each image either a patch is randomly sampled, a path is sampled so that the minimum Jaccard overlap with the objects is higher than a threshold, or the original image is used. After this, the image is resized and flipped with a probability of 0.5, and some photometric distortions are applied.

Regarding performance on the Pascal VOC 2007 dataset, SSD achieves 74.3% mAP and 59 FPS, surpassing the first version of YOLO in terms of speed and accuracy balance.

On the Microsoft COCO and in the context in which YOLOv4 was tested, described in [1], SSD achieves 43 FPS with 25.1% mAP. This shows that YOLO surpasses SSD in terms of prediction quality, with minimal time costs.

3. Theoretical foundations

In this chapter, we will go through the basics of supervised learning, and we will particularly analyze the convolutional neural network (CNN) way of solving this problem. More specifically we will focus on the CNN layers relevant to our problem and the fundamentals of training a CNN. Also, we will review some important model architectures and their importance.

3.1 Supervised learning

Let x and y be multi-dimensional variables and f be a mapping that takes as input x and outputs y.

$$f(x) = y \tag{3.1}$$

As we can see in 3.1, f is essentially a function defined as $f : X- > Y$ where $x \in X$ and $y \in Y$. Usually, x and f are given and y is deduced, but the idea behind supervised learning is to find this f, or mapping, between some inputs and outputs that are known beforehand in order to predict for some inputs, as correctly as possible, their outputs that are not known beforehand. We denote this prediction as $\hat{y} \in Y$.

The input-output pair is also known as labeled data, or more commonly training data, and the more pairs are in a set of training data, the better the mapping function will be. This is useful when the exact mapping is not known, or it is too computationally complex.

As a parallel, unsupervised learning is similar, but the labels are missing altogether. These two types of algorithms belong to the class of machine learning in which machines or algorithms can learn from existing experience or data. This field is part of the much larger domain of AI which is like an umbrella term for a broad variety of algorithms that replicate natural intelligence.

Our problem is mainly solved through supervised learning, therefore we will delve deeper into this matter.

Considering the way we have defined supervised learning in 3.1, two kinds of problems are fit to be solved with this method: classification and regression. The

difference between the two is given by the label y. If y is a discrete variable, then the problem is a classification one. These finite possibilities are also called classes, hence the name classification. For example, in order to classify digits, ten classes are needed, one for each digit. In this case $y \in \{0, 1, 2, ..., 9\}$.

In the case of regression, y is a continuous variable, meaning it can take an infinite number of values in a certain interval I, where $I \subseteq \mathbb{R}$. For example, in the case of real estate prices $y \in [0, \infty)$.

In all these cases, the input can be virtually anything from a set of features to images or sounds.

Since supervised learning is all about finding the output of a previously unknown input, using a set of known input-output pairs, the problem of overfitting arises. This basically occurs when the supervised learning algorithm only performs well on the known input-outputs, i.e., training data, but it performs poorly on the unseen input-output pairs, i.e., test data. This means that it learns features about the training data, but it can't generalize to new data. This problem often occurs due to noisy data, or bad quality data in general.

Also, underfitting is similar, but in this case, the algorithm doesn't learn the training data.

In order to know when an algorithm is overfitting, the balance between the train and test error must be analyzed, relative to the human error. This balance is also known as bias-variance trade-off.

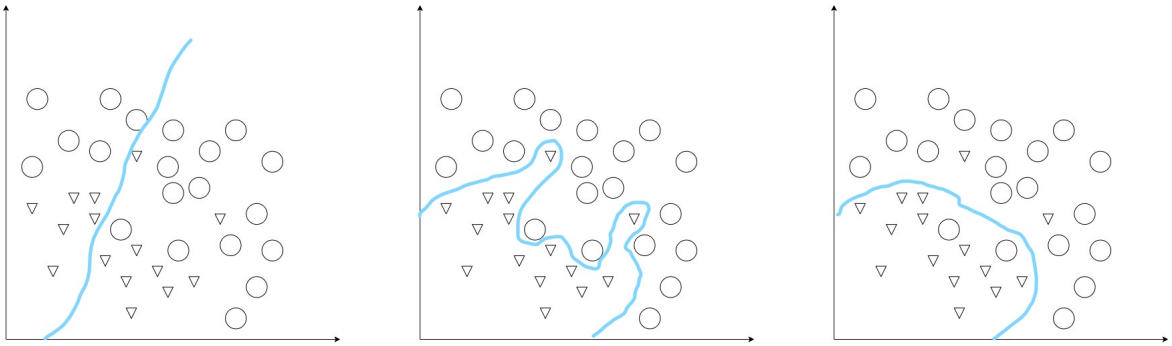
	High variance	High bias	High bias High variance	Low bias Low variance
Train data error	6%	19%	16%	5.5%
Test data error	20%	20%	30%	6%

Table 3.1: Bias-variance trade-off concrete example when the human error is $\sim 5\%$

A high gap between the test and train error, while the training error is close to the human error is called high variance and this is an indicator of overfitting. This can be seen in the first column of 3.1. Meanwhile, a high difference between the human error and the training error represents a high bias and it indicates that the model is underfitting and there are problems during training. This can be seen in the second column of 3.1.

Ideally, an algorithm should have low bias and low variance as it is in the fourth column in 3.1, but it may happen that both bias and variance are high, like in the third column of 3.1, which is the bad extreme, and it represents very poor performance.

Further on, we present the bias-variance trade-off in a graphical way on a simple binary classification problem. In 3.1 the circles and the triangles are the instances



(a) High bias (b) High variance (c) Balanced

Figure 3.1: Bias-variance visualization in the case of binary classification

of the two classes, represented in a simple 2D plane. The blue boundary represents the function from 3.1 that should classify or separate the two classes. As it is depicted in 3.1a, in the case of high bias, the algorithm would underfit and not extract anything meaningful from the structure of the train data. However, when the algorithm overfits and extracts too well the structure, something like 3.1b happens. Otherwise, when the data is good and the algorithm is able to learn it, the function from 3.1 would look something like this 3.1c, and because the shape of the function is smoother, the function is more robust to new data, and it performs better.

3.2 Artificial neural networks

The problem of supervised can be solved in various ways such as decision trees, linear regression, support vector machines, or neural networks. We will further explore the latter because it is more relevant to our problem.

The class of neural network algorithms itself is divided into several types of neural networks such as artificial neural networks, generative adversarial networks, or convolutional neural networks.

In general, the idea behind neural networks is to try to replicate the structure of the human brain. In the case of artificial neural networks, there are simple cells that receive one or multiple inputs and have usually one output that is the result of some computation over the input, as is depicted in 3.2.

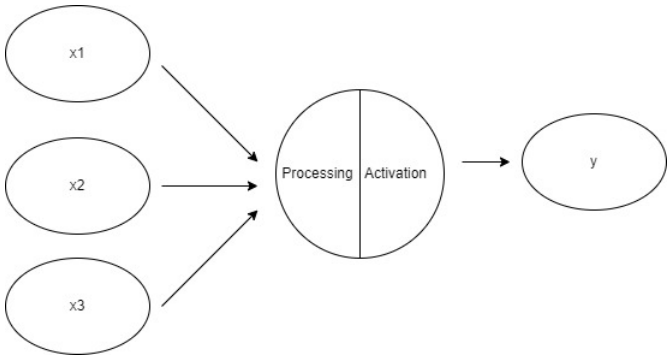


Figure 3.2: Perceptron

This is called a perceptron and it is an attempt to replicate one neuron from the brain, although a neuron is capable of more things and of more complex computations. 3.2 is a graphical representation of the function from 3.1 and we can see its components, where the input x corresponds to x1, x2, x3, and y is the output in both representations.

Usually, the processing part is computed as a polynomial where the edges are the coefficients of each x and there is another value added called bias, or the free term of the polynomial. The order of the polynomial is equal to the number of edges, or weights as they are called more commonly. The activation part will be explained in a latter section.

In order to build an artificial neural network, multiple perceptrons are stacked in a layer, which is also called a dense layer, then multiple layers are stacked as well. Usually, there is an input layer and an output layer and in the middle, there can be several hidden layers.

It is easy to see that the size of the network grows quickly as the size of the input grows. For example, if the network has only one hidden layer with three cells and we need to process an RGB image of size 416x416 we would first need to flatten the image. This results in an input vector of size $416 * 416 * 3 = 519168$. Then there would be $3 * 519168 + 3 = 1557507$ or around 1.5 million parameters in total. We multiply by three because each cell is connected to each input, and we add three because each cell has a bias. This is a large number, and in most cases, a network like this is too small to learn anything from images, therefore more layers would be required, and hence the number of parameters would grow exponentially. This type of neural network is not feasible for analyzing images.

3.3 Convolutional neural networks

The type of neural network suitable for processing images is the CNN because it can automatically extract features from images.

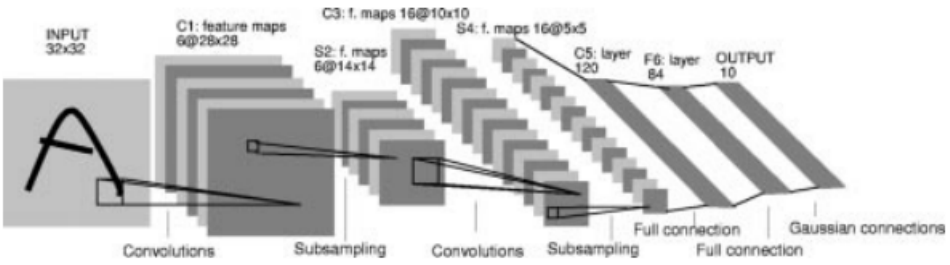


Figure 3.3: The first convolutional neural network developed by Yann LeCun used for digit recognition [22]

The first CNN was developed in 1998 by Yann LeCun [22] and it is depicted in 3.3, but at that time the hardware wasn't sufficiently advanced in order to highlight the potential of this type of network, and only in 2012 they began to be widely used when the AlexNet CNN [20] achieved the best results up to that time on ImageNet [4].

CNNs are similar to artificial neural networks in the sense that they respect the supervised learning formula 3.1, but they use convolutions. Basically, instead of the perceptron, convolutional filters are stacked together in what is called a convolutional layer.

3.3.1 Convolutional layer

The fundamental notion of CNNs is the convolution, which is not the same as the convolution from mathematics.

In computer vision, the convolution operation consists of applying a filter or kernel over an image. We are interested in the two-dimensional convolution, but the principles can be extended to three dimensions.

More formally, the input is a three-dimensional volume, where the first two dimensions are called spatial dimensions and the last dimension is called channel dimension or depth.

The parameters or the weights as they are more commonly called, of the operation are a filter or a kernel that has the same number of channels as the input volume but usually, the spatial dimension is much smaller, ranging from 3×3 to 11×11. These weights are the equivalent of the coefficients from the polynomial that is computed by the perceptron.

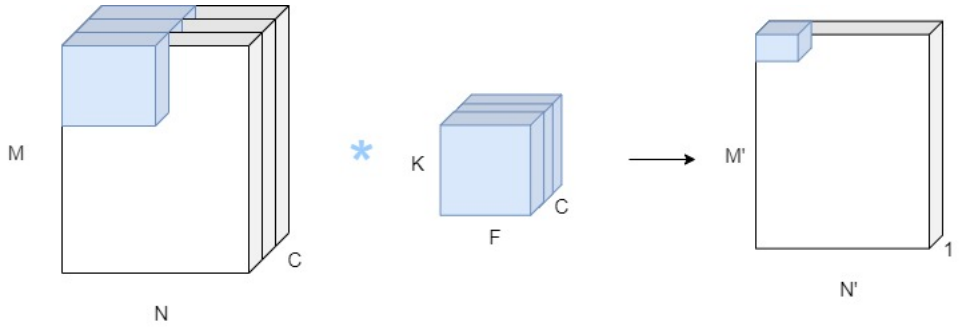


Figure 3.4: Visualization of the convolution operator. M is the height and N is the width of the input volume. K is the height and F is the width of the filter. C is the number of channels of both the input volume and the filter. M′ is the height and N′ is the width of the output volume which has 1 channel.

The actual computation of the convolution is highlighted in 3.4. The blue filter is overlapped at each position of the input volume and the result is the sum of the elements of the pointwise multiplication of the overlapped area. The result is put in the output feature map at the position in which the overlapping was done. This is the position in which the center of the filter is aligned in the input volume.

Following this rule, and the naming convention described in 3.4, the height and width of the output volume are:

$$M' = M - K + 1$$

$$N' = N - F + 1$$

The convolution operation also supports padding and stride. Padding represents the extra values that are appended to the start and end in both spatial dimensions in order to be able to overlap the filter closer to the margin of the input volume. A common value for padding is zero. The stride represents the size of the step the filter takes when it traverses the input volume, both horizontally and vertically. Usually, the stride is one.

The updated formulas for computing the width and height of the output volume are:

$$M' = \frac{M - K + 2 * P}{S} + 1$$

$$N' = \frac{N - F + 2 * P}{S} + 1$$

Where P is the size of the padding and S is the size of the stride.

In the same way that a dense layer stacked perceptrons, the convolutional layer can stack filters to obtain an output volume with multiple channels. If the output

volume needs to have some number of channels, the same number of filters must be created.

The advantages of this method are that the spatial information is well kept, and it is robust to the exact position, but the number of parameters is greatly reduced as well because it does not depend on the input volume, but it depends on the filter. This also means that a convolutional layer can be applied to any volume and the size of the output will be adjusted depending on the input. As a parallel, a dense layer is more rigid because it requires a fixed input size.

Also, as the authors of VGG [23] note, multiple stacked filters of small size have a receptive field equal to the receptive field of a larger filter, but the number of parameters is smaller. For example, a 7×7 filter has $49 * C$ parameters, whereas three 3×3 filters have $3 * 9 * C = 27 * C$ parameters, where C is the number of channels, and the smaller stacked filters have the same receptive field as the single large one. Therefore, it is better to stack multiple small filters, rather than using a small number of large filters.

3.3.2 Depthwise separable convolutions

When the spatial or channel dimension is high, even the convolution operation struggles in terms of performance. Spatial convolution was first introduced in [36] and their purpose is to compute the convolution in two steps in order to reduce the number of parameters. This usually negatively impacts the accuracy by a small margin, but it gives a considerable speed boost since there are fewer parameters.

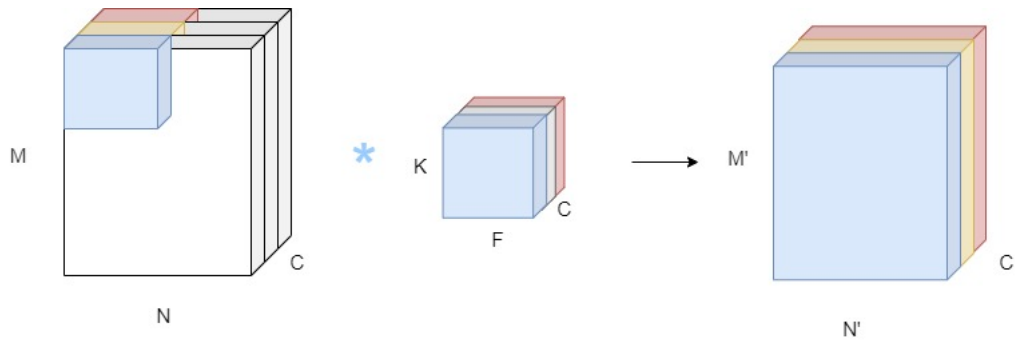


Figure 3.5: Visualization of the depthwise convolution operator. Same naming convention as in 3.4

There are two main steps in computing the depthwise separable convolution. Firstly, a depthwise convolution is applied, which is similar to a normal convolution, but each filter is applied to one channel only and the number of filters is equal to the depth of the input, i.e. the number of channels, as it is represented in 3.5. The cost of this operation is $K * F * C * M' * N'$.

The second step is the pointwise convolution, which is a normal convolution but with 1x1 filters. This kind of filters are used to control the channel dimension. And this is indeed the case. In order to obtain the desired number of channels, a pointwise convolution is applied to the output of the depthwise convolution. The cost of this operation is $C * C' * M' * N'$.

Using the notations from 3.4, the cost of a convolution in terms of the number of operations is around $K * F * C * C' * M' * N'$, where C' is the number of output channels, whereas a depthwise separable convolution has a cost of $K * F * C * M' * N' + C * C' * M' * N'$. The formulas might be similar, but the key takeaway is the plus in the middle. The computations are reduced by the following factor:

$$\frac{K * F * C * M' * N' + C * C' * M' * N'}{K * F * C * C' * M' * N'} =$$
$$= \frac{K * F * C * M' * N'}{K * F * C * C' * M' * N'} + \frac{C * C' * M' * N'}{K * F * C * C' * M' * N'}$$
$$= \frac{1}{C'} + \frac{1}{K * F}$$

3.3.3 Transposed convolution

The classical convolution can reduce the spatial dimension, or at most keep the same spatial dimension between the input and output, but sometimes it is needed to increase the spatial dimension, and this is what a transposed convolution does.

The transposed convolution is a special case of up-sampling because it is learnable.

In this case, the filter is multiplied pointwise with the input at each position and the resulting volume is accumulated in the output as it is shown in 3.6.

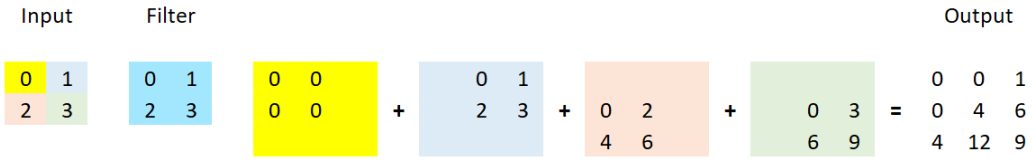


Figure 3.6: Transposed convolution computation example

This is especially useful when some operation needs to be computed on the output of two layers that output volumes with different spatial dimensions.

3.4 Training a neural network

So far, we have defined the building blocks that can be used to build a neural network. The next step is to train the neural network and we will explore the fun-

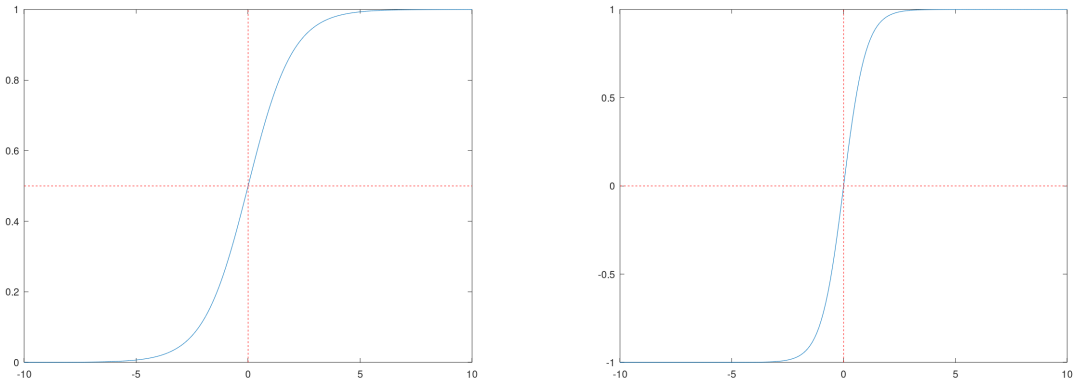
damentals of training such as activation functions, optimization, initialization, and regularization mechanisms.

3.4.1 Activation functions

Usually, real-world data is so complex that in order to create boundaries between classes, for example, it is not sufficient to have linear classifiers. This means that the data is not linearly separable, even in higher dimensions.

In an earlier section, we have explained how the perceptron basically computes a polynomial, and a convolution does something similar. This means that multiple stacked cells only give another polynomial. And for some data, it would be really difficult for the neural network to learn it this way.

This problem is solved with activation functions. As it is shown in 3.2 the activation function is applied to the processed result. This way nonlinearities are introduced and the function from 3.1 is not a polynomial, but a complex function that is able to learn the structure of the data.



(a) $\sigma(x) = \frac{1}{1+e^{-x}}$ (b) $tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

Figure 3.7: Sigmoid function on the left and tanh function on the right

Sigmoid 3.7a and tanh 3.7b functions used to be the choice when it came to activation functions, but due to the fact that on the margins the derivatives are very small, they are not that used anymore as an activation function in a hidden layer. Usually, the sigmoid is used in the output layer because its output is between zero and one, and this is useful for binary classification or proportions.

Another downside of the sigmoid is that it can't be used in the case of classification with multiple classes. For this problem, the softmax function is used 3.2.

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, i = 1..n \tag{3.2}$$

The idea is that a vector of values is mapped to another vector of values in which the sum of the elements is one because they are probabilities, and the proportions are kept relative to the first vector, i.e. the largest value in the first vector will also be the largest in the second and so on.

Nowadays, the ReLU function became the standard for activation functions inside hidden layers. The reason is that it is non-linear because if the value is less than 0, the result is 0, otherwise the result is the input value. This way the derivatives don't become too small or too large on the margins, but it is one if x is larger than 0, and 0 otherwise.

An improvement over ReLU is LeakyReLU 3.8, which has another parameter, alpha, that is the value of the derivative when x is smaller than 0. If alpha is 0, then LeakyReLU is the same as ReLU.

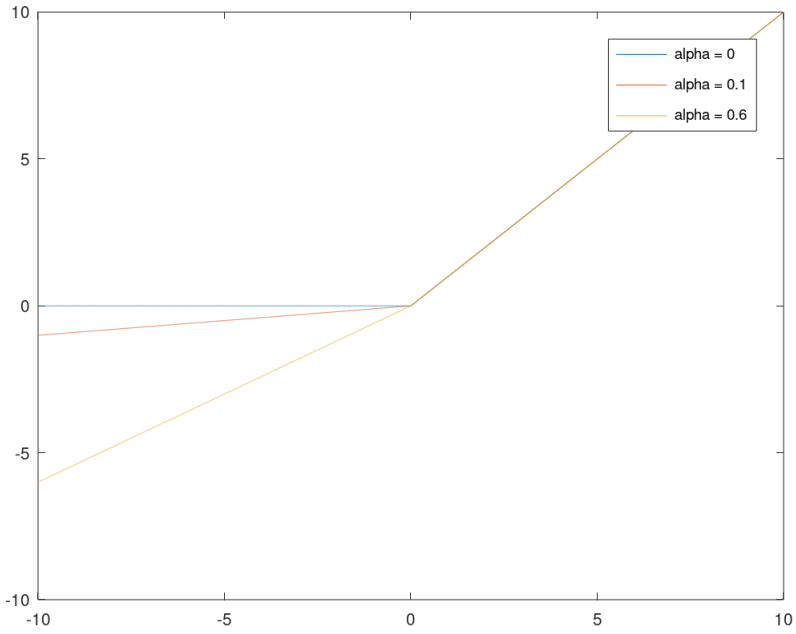


Figure 3.8: $LeakyReLU(x) = \begin{cases} x & 0 \leq x \\ alpha * x & otherwise \end{cases}$

3.4.2 Optimization

The next, and most important step is to find the values for the weights that best fit the data. This is usually done with gradient descent and backward propagation of errors, or backpropagation for short.

The gradient descent method goes back to 1847 when Cauchy proposed it [3] and a real-life intuition would be that it is similar to trying to reach the lowest point in a mountainous area, starting from some random location like the top of the mountain.

In more mathematical terms, the gradient is the equivalent of the derivative, but in higher dimensions. For example, if the derivative is positive, then the function is increasing, if the derivative is negative it is decreasing, and if it is zero, then that point is a maximum or a minimum. Therefore, the derivative gives information about the direction towards a minimum or a maximum.

Something similar happens in gradient descent too. The gradient of 3.1 gives the direction at some point and based on that direction, some adjustments are made to the weights, or the parameters of the function in order to reach a minimum, preferably a global minimum, hence the descent part.

To do this parameter update, backpropagation is used, which consists first of a forward pass, and a backward pass. The forward pass is simply passing the input through the network and recording the output values of each part of the network. This is like applying the function from 3.1. Then, in the backward pass, based on some loss function that gives the error, or the difference between the predicted output and actual output, and the computed values in the forward pass, the weights are updated from the end of the network, back to the start of the network.

In the classical version, the parameter update is done after seeing all inputs. If the parameter update is done after seeing a single input, then the method is called stochastic gradient descent, otherwise, if it is done after a batch of examples, then it is called mini-batch gradient descent.

The classical formula for gradient descent is:

$$W = W - \alpha * dW$$

Where W are the weights, dW are the gradients and α represents the learning rate, which is a parameter, usually between 0 and 1, that controls the size of the step taken.

Over time several improvements over gradient descent have been done. For example, gradient descent with momentum was introduced in [39] and has the following formula, which is an exponential weighted average:

$$v_{dW} = \beta * v_{dW} + (1 - \beta) * dW$$

$$W = W - \alpha * v_{dW}$$

If we visualize the process of finding the weights in order to find a minimum as a ball rolling on a surface, then the weights are the ball, β is like the friction, v_{dw} is like the velocity and dW is like the acceleration.

In some variants, the term with $1 - \beta$ is removed completely.

The next improvement is AdaGrad, first introduced in [6], and the idea is that it keeps a cache, and has the following formula:

$$cache+ = dW^2$$

$$W = W - \alpha * \frac{dW}{\sqrt{cache} + 1e^{-7}}$$

This formula balances the step on shallow directions.

An improvement over AdaGrad was given in [14], by Geoffrey Hinton in a Coursera lecture, called RMSProp. The formula is the following:

$$cache = \beta * cache + (1 - \beta) * dW^2$$

$$W = W - \alpha * \frac{dW}{\sqrt{cache} + 1e^{-7}}$$

Basically, exponential weighted average is applied on the cache.

Another improvement on RMSProp is Adaptive Moment Estimation, or Adam [18]. This method adds momentum to RMSProp:

$$m = \beta_1 * m + (1 - \beta_1) * dW$$

$$v = \beta_2 * v + (1 - \beta_2) * dW^2$$

$$W = W - \alpha * \frac{m}{\sqrt{v} + 1e^{-7}}$$

The m comes from momentum and v comes from RMSProp. The effect is that the noise in the gradients is reduced. This optimizer is the usual choice nowadays because it is more robust to different training scenarios, but when more control is needed some may choose simple gradient descent.

3.4.3 Initialization and regularization mechanisms

Another, not so intuitive, important part of training is the initialization of the weights. One option would be to initialize the weights with zeros, but this won't work because the output will be the same for every input. A more viable option is to initialize the weights with random numbers, but they need to be small to not cause vanishing and exploding gradients.

One common option is Xavier initialization [11], in which the weights are random numbers with zero mean and $\frac{1}{n}$ variance, where n is the number of weights. The authors found out that by initializing in this way, the weights keep the same variance during training, thus leading to a more stable backpropagation.

Another way is the He initialization [12]. It is similar to Xavier initialization, but in this case, the variance is $\frac{2}{n}$ and the authors point out that this kind of initialization is suited for the class of ReLU activation functions.

Initialization helps in the problem of underfitting, but overfitting can be reduced by regularization. Overfitting often appears because the function 3.1 is too complex and fits exactly the training data. The idea of regularization is to reduce this complexity, by making the function simpler, in our case, making the neural network simpler.

Regularization can be achieved by adding to the loss a value named regularization loss. There are two types of this regularization.

In L1 regularization, the sum of the absolute values of the weights are added, and this causes the weights to be sparser.

In L2 regularization, the sum of the square values of the weights is added, and this causes the weights to be spread out.

In both cases, a parameter λ is multiplied by the value of the regularization loss in order to control its value.

Another popular method of regularization is dropout [38]. Based on a given probability, the cells of the neural network output zero. The effect is that for each training epoch, the neural network has different shapes because it's like random branches are dropped, and the network is simpler.

This is the behavior during training, but during inference, dropout is not applied.

Batch normalization was introduced in [16] and can have a regularization effect. During training, each channel is normalized by the mean and variance of the whole batch, channel-wise, as it is depicted in 3.9. During inference, most likely there isn't a batch, therefore the channels are normalized by a mean and variance computed empirically during training.

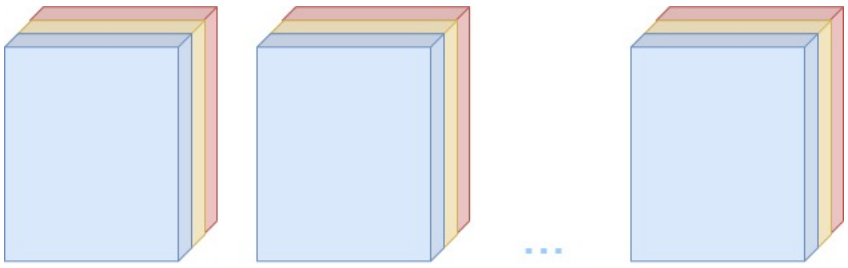


Figure 3.9: During batch normalization each channel is normalized by the mean and variance of the whole batch, but channel wise. In this example, each blue channel would be normalized by the mean and variance of all blue channels. The same process happens for the other channels.

So far, we have discussed regularization methods that reduce the complexity of the neural network, but a very crucial regularization method is data augmentation. This is a data-centric regularization method, as it doesn't alter the structure of the neural network, but it alters the input itself. The key point is that by altering the input, new data is generated, and the network will adapt to a more variate set of inputs.

For example, common data augmentation techniques for images include photometric distortions like changing the brightness or saturation, geometric distortions like flipping or rotating, or other methods like cutout [5], which simply blacks out a piece of an image. This way, the network should be able to recognize some objects, even if the whole object is not visible. Other methods combine several training images into a single one. For example, mosaic augmentation, introduced in [1], combines four images into one, each having different aspect ratios. This helps the network learn to identify smaller objects.

3.5 Neural network architectures

In this section, we will analyze some neural network architectures relevant to our problem. It is important to understand established neural networks because they introduce several structural innovations and, usually, there are pretrained weights that can be used in other problems. This is called transfer learning, because the knowledge gained during learning one dataset, could be transferred to some other dataset.

For example, when there aren't many examples in a dataset, transfer learning is the usual choice because the earlier layers in the pretrained network learned the basic features needed for the problem and can be reused in the new problem. Also, for these established architectures there is usually a set of weights that have been trained on very large datasets, therefore it is not worth it, or feasible in terms of time or data quality most of the time, to train it from scratch.

Practically, what is done in order to use transfer learning, is to take a pretrained model on a similar dataset to the dataset of the problem at hand, and replace the last layer, or layers depending on how similar the datasets are, with new layers that are trained from scratch.

3.5.1 Residual neural networks

In general, adding more layers should not hurt performance, it should at least improve it, but until residual connections were introduced in [13], it was very hard to optimize deep networks. The reason was that with a lot of layers, the gradients become very small, thus it was hard for the earlier layers to be updated and therefore there wouldn't be much knowledge gained. The innovation is represented in 3.10, where the input is simply added to the output of the following operation, forming what is called a residual block, where the addition is the residual connection.

This allowed the authors to obtain state of the art results with networks that have tens of layers by stacking residual blocks. The reason why this is effective is that

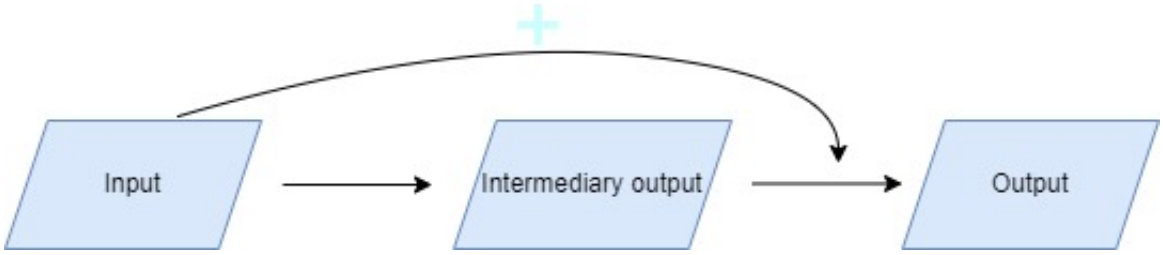


Figure 3.10: Residual block

even if on the main path the gradient flow is bad, on the residual path, there will still be gradients that allow the weights to update and exit a local minimum or a plateau. Another reason is that if the network doesn't really need the residual connection, it still can learn the identity function, therefore extra complexity is avoided, and the network does not overfit.

3.5.2 U-Net

U-Nets [33] are characterized by their U shape, hence the name. As we can see in 3.11, there are two paths, a downsampling path, and an upsampling path and there are connections between the two.

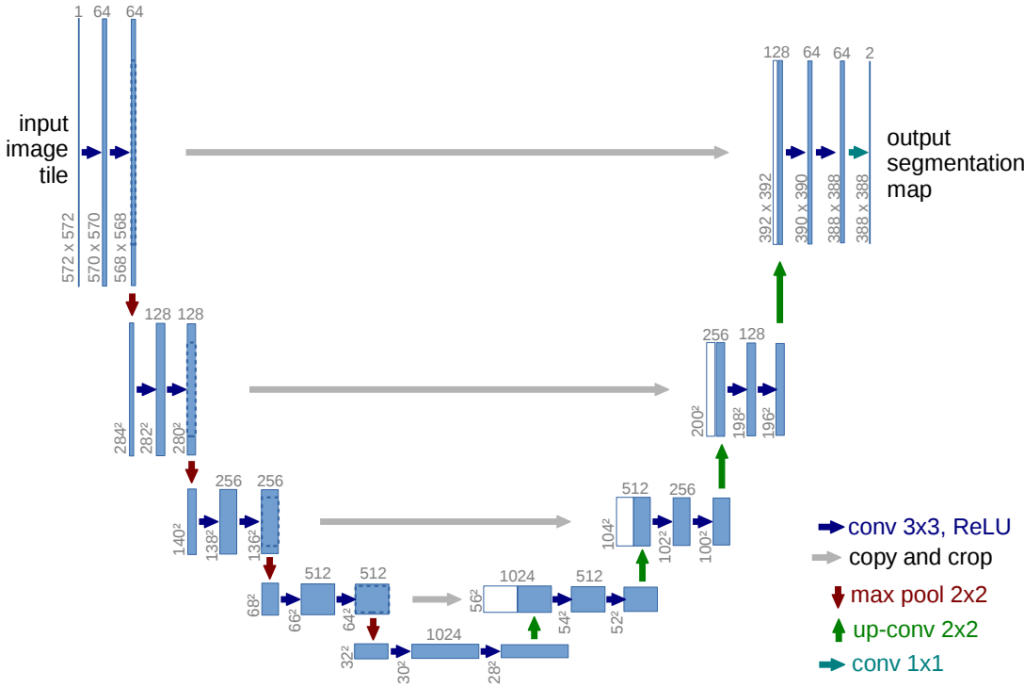


Figure 3.11: U-Net architecture introduced in [33]

This kind of architecture is especially useful when the output is itself a feature map, and in this way, it is easy to build up the spatial dimension.

The downsampling path represents the usual flow of the network in which the input features are refined step by step. In the upsampling path some mechanism of increasing the spatial dimension is used, like transposed convolutions, in order to build up the feature map. At the same time, feature maps from the downsampling layers are concatenated to the feature maps from the upsampling layers. This is like allowing the network to make predictions at multiple resolutions.

Another advantage is that the network is fully convolutional, and this means that the spatial dimension of the input can vary, and the dimension of the output map will change accordingly.

3.5.3 MobileNets

One way to run a neural network on a phone or an embedded system is to use a server that communicates with the remote device through an API, by receiving the input and doing the computation, and responding with the output. This approach is not ideal. It would be difficult to have a real-time system due to latency, and the remote device depends on the internet connection. Also, neural networks require a complex computation, and probably there would be several requests that would need to be processed in parallel, therefore a strong server would be required, which tends to be expensive.

Recently, due to the various developments in hardware, the use of neural networks on embedded or mobile systems began to gain traction. The idea of a server is removed completely, and the actual computation is done on the device. This method also has its disadvantages. The power consumption would be high, therefore very large networks are still not feasible on a mobile device. Therefore, a compromise between speed and performance is achieved.

Usually, when accuracy or performance in some metric is key and the time is not that important, the first method is used. When speed is of the essence and it doesn't matter if the network becomes a little bit weaker, then the second approach is suitable.

MobileNets were introduced in [15] and their purpose is to be an efficient type of neural network to be run on a mobile device, as the name suggests. Their efficiency is owned to the use of depthwise separable convolutions, which as we have discussed in an earlier section, are a lightweight variant of the classical convolution.

Also, they are called MobileNets because the authors trained several networks, of different sizes, in order to accommodate the needs of various mobile devices dynamically. For example, a parameter α, or width multiplier as the authors call it, controls the percentage of filters that are used. If $\alpha = 100\%$ then all filters are used, but if $\alpha = 50\%$ then the neural network has half the filters of the first one. In this

way, a powerful mobile phone would use the full network, and a weaker one would choose a weaker variant, depending on its hardware.

Some improvements are brought in MobileNetV2 [35] such as linear bottlenecks and inverted residuals blocks. Inverted residuals are similar to a normal residual block, but the difference is that an inverted residual block increases the channel dimension in the middle volume, whereas the normal residual shrinks it. Also, the authors observe that non-linear activation functions such as ReLU can destroy information, therefore they have introduced linear bottlenecks, which simply means that they don't use an activation function after the final operation in a residual block.

4. Design and implementation

In this chapter, we will focus on the implementation details of our solution. The discussion is split into two main sections, one for the object detector and one for the mobile application, which uses the object detector.

4.1 Object detector

4.1.1 Dataset

For training the object detection system we use the Open Images Dataset V4 [21], available at [19]. In total, the dataset contains 9.2 million images, including 14.6 million bounding boxes across 600 classes on 1.74 million images. We use only a subset of classes: bus, car, and license plate or vehicle registration plate as it is called in the original dataset.

The tool used to download the bus, car, and license plate classes and the corresponding bounding boxes is OIDv4 ToolKit [41]. The dataset is split into three parts: train, validation, and test, and the exact number of images and bounding boxes in each set is detailed in 4.1.

	Number of images	Number of bounding boxes			
		Total	Bus	Car	License plate
Train	30939	80291	11927	60513	7851
Validation	4967	9985	92	9381	512
Test	14894	30660	353	28737	1570

Table 4.1: Bounding box statistics related to the subset of Open Images

The dataset, as it is in its original form, is unbalanced. As we can see in 4.1, the car class has around 5-6 times the number of bounding boxes the other classes have. This can be problematic because the object detector will tend to predict mostly cars.

We try to address this issue by using a technique called undersampling. The idea is that we try to balance the numbers by removing instances of the dominant class. Therefore, we prune the dataset, and the resulting number of bounding boxes is detailed in 4.2.

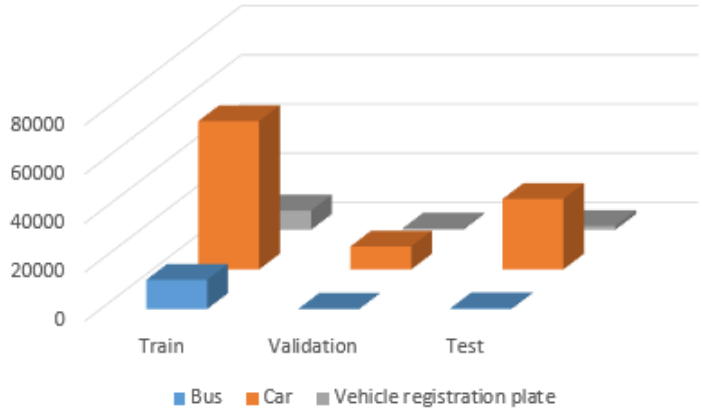


Figure 4.1: Dataset bounding boxes distribution

	Number of images	Number of bounding boxes			
		Total	Bus	Car	License plate
Train	19773	47938	11927	28159	7852
Validation	4967	9985	92	9381	512
Test	669	1483	353	764	366

Table 4.2: Bounding box statistics related to the undersampled subset of Open Images

Also, as we can see in 4.2, the classes are more balanced. There are only three times more car bounding boxes than other classes.

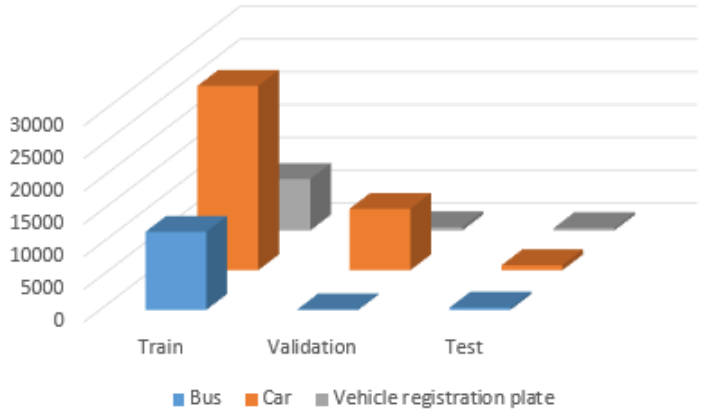


Figure 4.2: Undersampled dataset bounding boxes distribution

We further improve the dataset by enhancing the license plate class using an existing highly performant license plate detector. Therefore, we use an object detector based on YOLO [17] in order to add new bounding boxes if they don't already exist. As we can see in 4.3, the number of boxes for the license plate is larger, and in 4.3 we can see the detailed distribution.

Another detail is that the maximum number of bounding boxes an image can have is 111.

	Number of images	Number of bounding boxes			
		Total	Bus	Car	License plate
Train	19773	50110	11927	28159	10024
Validation	4967	10824	92	9381	1351
Test	669	1580	353	764	463

Table 4.3: Bounding box statistics related to the undersampled and enhanced subset of Open Images

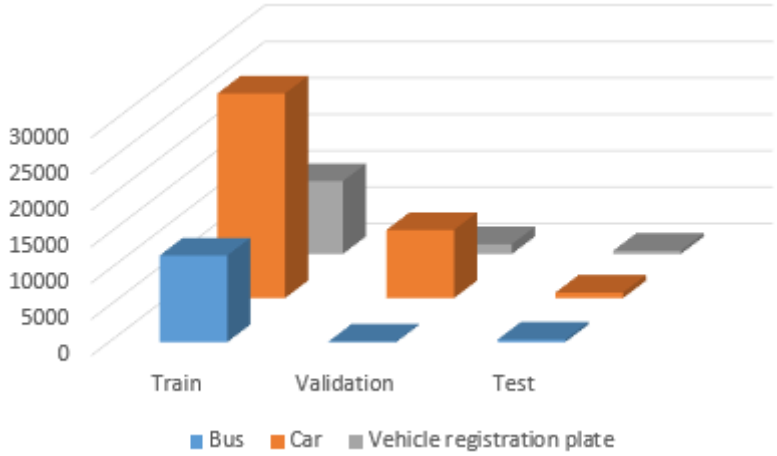


Figure 4.3: Enhanced dataset bounding boxes distribution

Each image is resized so that its dimension is 416×416 and the bounding boxes are modified accordingly. This is done with the help of Albumentations Python library.

The resizing helps the object detection task, as explained in [30] because this way we can split the image into a grid of 13×13 cells of size 32×32 so that there is a cell in the center that can detect the larger objects that are centered in the middle, rather than have 4 cells in the middle that try to detect the same object.

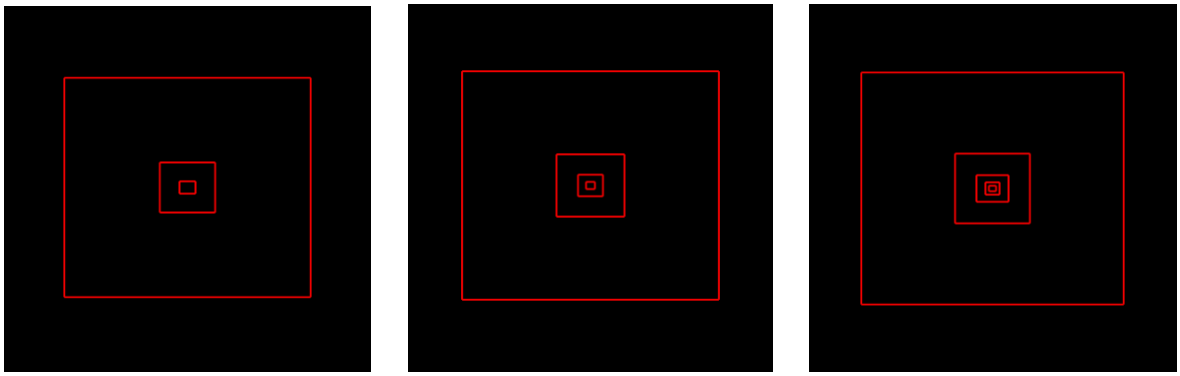
Each image is associated with multiple bounding boxes and each cell is responsible for detecting multiple bounding boxes through the use of anchors as explained in [30]. We use three anchors per cell so that each cell can detect various shapes and sizes. The anchor boxes are chosen using K-Means over the dataset.

For computing the distance between the centroid and bounding box the following formula is used, as in [30]:

$$distance(centroid, box) = 1 - IOU(centroid, box)$$

Where IOU represents intersection over union and is calculated as the area of the intersection of the two boxes over the area of the union of the two boxes.

We have tried to use three 4.4a, four 4.4b and five 4.4c centroids, or clusters, in



(a) Three anchors (b) Four anchors (c) Five anchors

Figure 4.4: Anchors for the bus, car, and license plate classes using three different values for the number of centroids

order to generate the anchors, but as it can be seen in the figures, in all cases the outer boxes are the same and in the case of four and five anchors, only smaller and smaller anchors are added which are not different enough from each other, therefore we choose to use the variant with three anchors.

Each image needs to be associated with a ground truth. We represent the ground truth as a $C \times C \times B \times (5 + N)$ tensor. For each cell in the $C \times C$ grid, we associate B anchor boxes. Each anchor box has the following parameters: b_x and b_y represent the center of the box, b_w and b_h represent the width and height, c is the probability that the anchor predicts an object and $c_1, c_2, c_3, ...c_n$ represent the class probabilities. In our case, C is 13 and both B and N are 3. Therefore, the output is a tensor of shape $13 \times 13 \times 3 \times 8$.

The formulas that are used to compute the center, width, and height from the output of the model are described in 4.1.

$$\begin{aligned} b_x &= \sigma(t_x) + c_x \\ b_y &= \sigma(t_y) + c_y \\ b_w &= p_w \cdot e^{t_w} \\ b_h &= p_h \cdot e^{t_h} \end{aligned} \tag{4.1}$$

Where t_x, t_y, t_w, t_h represent the raw predictions to which the sigmoid function is applied, c_x, c_y represent the coordinates of the upper left corner of the cell that predicts the box, and p_w, p_h represent the width and height of the anchor that predicts the box. Over the output for the class probabilities, the softmax function is applied.

4.1.2 Model

The object detection model is inspired by YOLOv2 [30]. The neural network is fully convolutional and is composed of three parts: feature extractor, neck, and head. For the feature extractor, we use MobileNetv2 [35] because of its flexibility and the fact that it uses depthwise convolutions, inverted residual blocks, and linear bottlenecks, which all help with performance, thus consuming less power, which is crucial for mobile solutions. Also, we use pretrained weights on ImageNet [4] to benefit from transfer learning.

For the rest of the model, we use convolution blocks 4.5 which are composed of a convolution layer that uses HeNormal initialization, batch normalization, and optionally LeakyReLU activation function. In general, we choose an alpha of 0.1 for LeakyReLU.

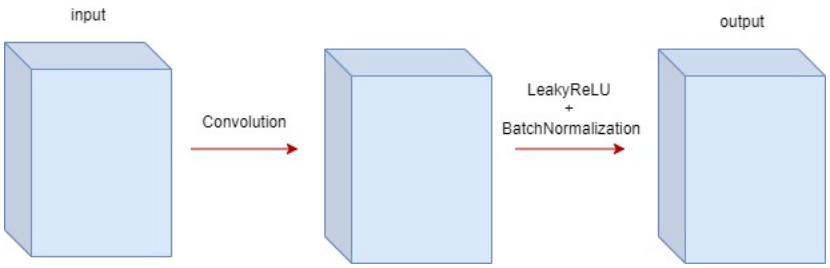


Figure 4.5: Convolutional block

The neck is inspired by U-Net [33]. The aim is to add features from earlier layers to the result through skip layers and upsample blocks 4.6 that use transposed convolutions, followed by LeakyReLu and batch normalization. This helps the network to ”see” the image at multiple resolutions as explained in the fine-grained features section in [30].

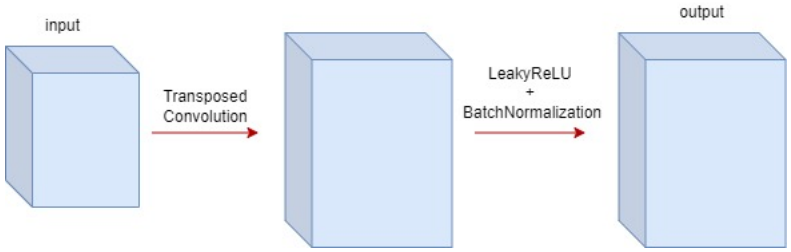


Figure 4.6: Upsample block

On top of the upsamlple blocks, a dropout layer with a probability of 0.3 is used for regularization in order to reduce overfitting. Another reason for adding this layer at this specific position is that the upsample blocks have the most trainable parameters, compared to other areas of the model. After the dropout layer, a convo-

lutional block and two inverted residual blocks 4.7 are added, which help in refining the feature maps.

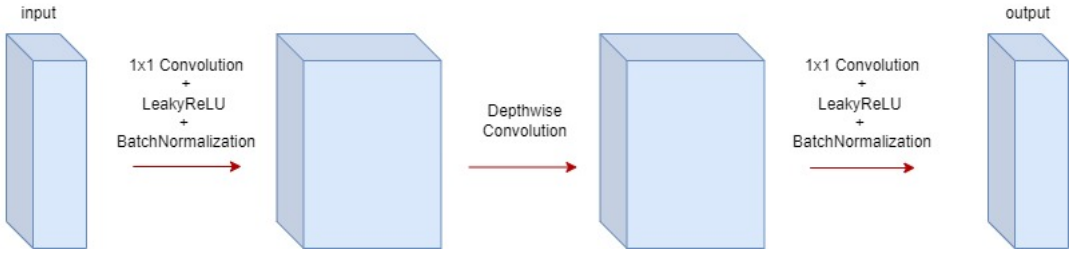


Figure 4.7: Inverted residual block

The head is composed of a convolution layer that has 24 filters in our case, so the final output is $13 \times 13 \times 3 \times 8$ after a reshape layer. This is because we use three anchors and three classes. This can vary if other datasets are used.

Also, an additional input that represents all the true bounding boxes is directly added to the output. This is an implementation trick that only helps in the computation of the loss because, even though each image is associated with a specific anchor, it is not restricted to be predicted only by that anchor. If the IOU threshold between the predicted box from another anchor and one of the true bounding boxes is high enough, that prediction is considered correct. During normal inference, a dummy array is passed for this input.

In total, our model has around 2 million parameters, out of which around 1.3 million are from MobileNet.

4.1.3 Loss

During training, we optimize a composed loss function described in 4.2. The implementation of the loss is an adaptation of [27].

$$L = L_{loc} + L_{obj} + L_{class} \tag{4.2}$$

In general, a variable a_{ij} represents the ground truth and $\hat{a_{ij}}$ represents prediction for the i'th cell and j'th anchor.

The first component is basically a sum-squared error, handling the localization loss, and is described in detail in 4.3.

$$L_{loc} = \frac{\lambda_{coord}}{N_{L^{obj}}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{ij}^{obj} [(x_{ij} - \hat{x}_{ij})^2 + (y_{ij} - \hat{y}_{ij})^2 + (\sqrt{w_{ij}} - \sqrt{\hat{w}_{ij}})^2 + (\sqrt{h_{ij}} - \sqrt{\hat{h}_{ij}})^2] \tag{4.3}$$

Where $L_{ij}^{obj} = \begin{cases} 1 & C_{ij} = 1 \\ 0 & otherwise \end{cases}$ is an indicator function in which C_{ij} means that there is an actual object in the i'th cell and j'th anchor. $N_{L^{obj}}$ just represents the number of actual object in the image and it is given by the following formula: $\sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{ij}^{obj}$. The center of the bounding box is denoted using the x for the horizontal position and y for the vertical position. The width is denoted with w and the height with h. The square root of the width and height is used because, otherwise, the error in small and large bounding boxes is treated the same.

The second component 4.4 is related to the objectness of a bounding box, which represents the probability that an object is present in that bounding box.

$$L_{obj} = \frac{\lambda_{obj}}{N^{conf}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{ij}^{obj} (IOU_{prediction_{i,j}}^{ground\ truth_{i,j}} - \hat{C}_{ij})^2 + \frac{\lambda_{noobj}}{N^{conf}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{ij}^{noobj} (0 - \hat{C}_{ij})^2 \quad (4.4)$$

Where $L_{ij}^{noobj} = \begin{cases} 1 & max_{i'j'} IOU_{prediction_{i,j}}^{ground\ truth_{i',j'}} < IOUthreshold\ and\ C_{ij} = 0 \\ 0 & otherwise \end{cases}$ is an indicator function which is one only if a predicted bounding box that does not appear in the ground truth in the respective cell and anchor, has a low enough IOU overlap with any ground truth bounding box. Basically, if the prediction has a high enough overlap with any bounding box, but it does not appear in the ground truth, then it is considered correct and it's not penalized, otherwise, it is not considered an object and it must increase the loss. Here we use the extra output from the previous section. $N^{conf} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{ij}^{obj} + L_{ij}^{noobj}(1 - L_{ij}^{obj})$ counts the number of bounding boxes from the ground truth, but also the boxes that predict objects where there shouldn't be any. Therefore, the first part penalizes the errors in the confidence scores for objects that should be predicted, and the second part penalizes the boxes that predict an object that should not be there.

The last component represents the loss from the class probabilities, and when multiple classes are involved, usually cross-entropy loss is used. This is the case too in 4.5.

$$L_{class} = -\frac{\lambda_{class}}{N_{L^{obj}}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} L_{ij}^{obj} \sum_{c \in classes} p_{ij}^{c} log(\hat{p}_{ij}^{c}) \quad (4.5)$$

Most grid cells do not contain any boxes. Therefore, in order to balance the confidence scores, λ_{coord} and λ_{noobj} are added in order to increase the loss from bounding box predictions and decrease the loss from confidence predictions. Also, λ_{obj} and λ_{class} are added to control the loss from the objectness and class losses. In 4.4, we detail the values for these hyperparameters.

λ_{coord}	5
λ_{obj}	2
λ_{noobj}	0.5
λ_{class}	3
IOU threshold	0.6

Table 4.4: Loss hyperparameters

4.1.4 Data augmentation

We apply various photometric data augmentation techniques such as random hue 4.8b, random brightness 4.8c, random contrast 4.8d and random saturation 4.8e. The data augmentation techniques in 4.8 are applied over 4.8a.

(a) Original (b) Hue (c) Brightness (d) Contrast (e) Saturation

Figure 4.8: Photometric data augmentation techniques

In 4.9 we present other data augmentation techniques that we have used. In 4.9a we represent the Cutout technique, introduced in [5]. The idea is to make the pixels from a random patch in the image black. This way the network should adapt to recognize objects even if they are partially visible. We also follow the recommendation from 4.9a, which states that the patch doesn't have to fit fully in the image. This means that if the center of the patch falls near one of the edges of the image, only the part that overlaps with the images is blacked out.

The other technique is called Mosaic 4.9b and it was first introduced in [1]. Here, four images are concatenated in order to form a single image. The effect is that

(a) Cutout (b) Mosaic

Figure 4.9: Other data augmentation techniques

the number of bounding boxes in a single image is increased, and the size of the bounding boxes becomes smaller, thus the network should better adapt to small bounding boxes.

In practice, for each image, we apply a random data augmentation technique from 4.8, and we apply all these data augmentation techniques only during training, and all of them, except Mosaic, are vectorized in order to be computed on the GPU.

4.1.5 Training

For training, we use a pretrained version of MobileNetV2 [35] on ImageNet [4], then we use Adam optimizer with epsilon 10^{-8} and decay 0 as in [27] and the rest default. We use a cosine annealing scheduler for the learning rate, described in [25], which follows the formula in 4.6.

$$LR_{epoch} = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min}) \cdot (1 + cos(\frac{epoch}{T} \cdot \pi)) \tag{4.6}$$

The main idea is that the learning rate will follow a cosine wave between the minimum learning rate η_{min}, and the maximum learning rate η_{max}. The epoch from 4.6 represents the current epoch, and T represents the restart epoch, meaning that every T epochs, the learning rate will reach either the minimum η_{min}, or the maximum η_{max}, or every T epochs, the monotony is changed. We can better see the phenomenon in 4.10, where various schedules for the learning rate are plotted over a total of 50 epochs. We can see that when T is equal to the total number of epochs, the learning rate decreases monotonously from the maximum to the minimum. If T is larger than the number of epochs, then the learning rate doesn't even reach the minimum, and if the learning rate is smaller than the number of epochs, the cosine wave can be seen, and every T epochs, the slope changes.

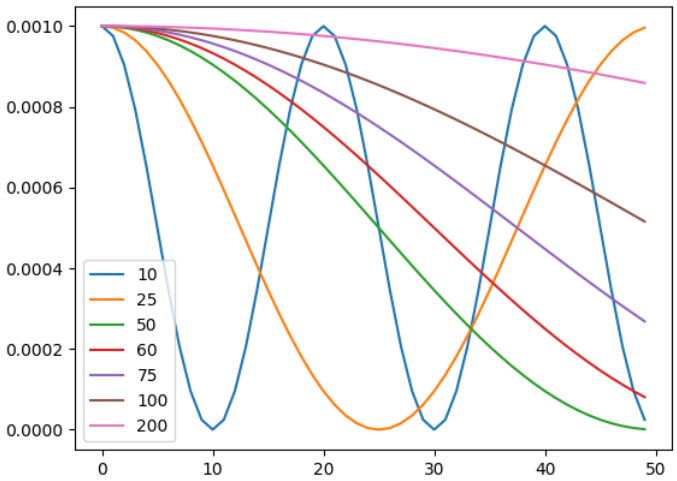


Figure 4.10: Cosine annealing scheduler

Regarding the loss, we usually expect a graph similar to 4.11, in which the horizontal axis represents the epoch, and the vertical axis the loss. Ideally, the validation loss is always around the training loss. This is a sign that the model does not overfit.

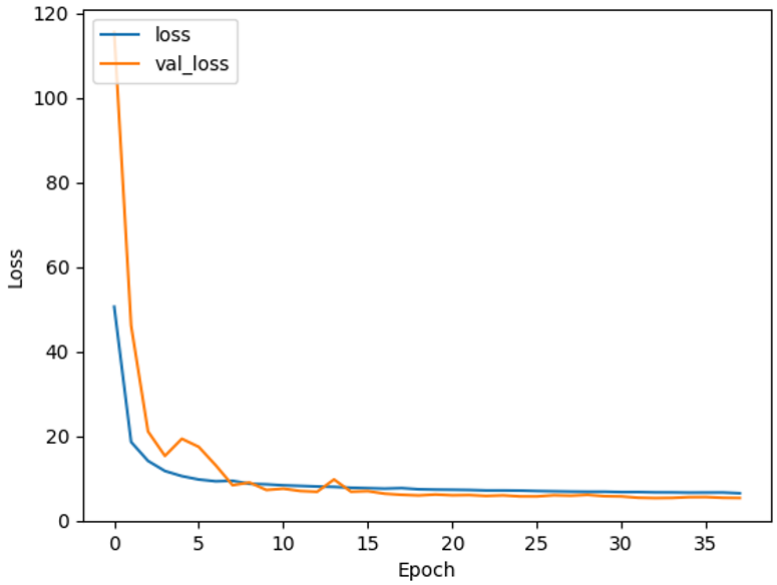


Figure 4.11: Training (blue) and validation (orange) loss for model v28

We train the model for 50 epochs, meaning that the network sees 50 times the whole train set and validation set. The model is saved every time the loss on the validation set improves. We also use early stop with the patience of five epochs and a delta of $1e^{-4}$. This means that if the model does not improve after some epochs, by the given delta, the training stops because we don't want to overtrain, in order to both save time and reduce overfitting also.

At the beginning of the epoch, the learning rate is set, following the formula in 4.6, and at the end, the images are shuffled in order for the network to see the images in a different order, each epoch. Also, during training, before any processing, random photometric data augmentation is used, followed by cutout and mosaic as explained in a previous section.

During a normal training session, the weights of the pretrained model are frozen, meaning that they do not update. We do this in order to not break the knowledge stored in the weights. But, during a fine tuning session, which occurs after a normal training session, we set a very small learning rate and unfreeze the pretrained model. By doing this, the previously frozen weights are updated to better fit our dataset. The learning rate needs to be very small in order to not break the knoweldge but to slowly enhance it. Usually, a fine tuning epoch takes much longer, because the pretrained model has the largest share in parameters of the total number of parameters.

4.1.6 Inference

The inference represents a pipeline of processing an input and getting the predictions for that input. In our case, the input is an image and there are three stages of processing that an image goes through in order to get the output, represented by the bounding boxes.

Firstly, there is preprocessing, which consists of normalizing the images in the range [-1, 1]. This is required by the MobileNetV2 backbone model. Also, here the data augmentation occurs, before the normalization.

The second step is passing the image through the actual neural network as we have explained in previous sections.

Postprocessing is the final step. Firstly, the bounding boxes are extracted from the resulting tensor of size $C \times C \times B \times (5 + C)$. Basically, for each cell in the $C \times C$ grid, we extract the $B \times (C + 5)$ raw bounding boxes. Then the raw values are converted using 4.1 to obtain the actual values. These boxes are filtered based on their score.

So far, most of the time, there are a lot of overlapping boxes that predict the same object. This is solved using Non-maximum Suppression (NMS) which prunes away extra bounding boxes. This is done by ordering the boxes by their scores, descending. Then each box is kept only if they have a low enough IOU with any previously kept bounding box with the same label. In this way, if there are a lot of boxes with the same label in some area, only the one with the highest score is kept. The authors of the original YOLO note in [29] that this step is not as important as it is for other models such as R-CNN [10], but it still improves the bounding box quality.

This stage introduces two new hyperparameters. One that controls the minimum score threshold of the bounding boxes, and one that controls the maximum IOU a bounding box can have with other boxes with the same label for NMS.

4.1.7 Implementation

Our object detection system is implemented in Python 3.7 and uses Tensorflow 2.3, including Keras. Other notable libraries include NumPy for various mathematical operations, Matplotlib for plotting, and Albumentations for some operations on bounding boxes. In terms of hardware, the training is performed on a NVIDIA GeForce GTX 1050 Ti GPU with 4021 MB of VRAM, the images are stored on a Samsung SSD 970 PRO and the CPU is an Intel i5-7300 with four cores, each having a frequency of 2.50 GHz.

4.2 Mobile application

In this section, we will describe the mobile part of our solution, which includes visualization capabilities of object detection systems in general and an assistive technology module demo. We follow the software engineering development activities described in the book by Bruegge and Dutoit [2].

4.2.1 Requirements elicitation

The purpose of requirements elicitation is to clearly describe the functionalities or features of the software application. Each functionality has a set of requirements that it must meet.

After this activity, the system is described in terms of use cases and actors. As is explained in [2], actors are defined as the external entities that interact with the system, and the use cases are sequences of actions between an actor and the system that describe a functionality. A use case can be split into several scenarios in order to describe multiple possible flows of the functionality.

Our application aims to provide the means to visualize the output of object detectors, both for photos and live images. This means it shows over the image, static or live, the predicted bounding boxes, the classes, and the scores. Also, it provides an assistive module demo, in which the live predictions are not shown on the image but are converted to sound such that a VIP can know about the surrounding objects by hearing the predictions. Finally, on static images and in the assistive module, using optical character recognition (OCR), information about the text present in the image is provided.

Firstly, we define the functional requirements by describing the use cases and their scenarios in natural language.

Configure the object detector parameters

The user is able to configure various parameters of the object detector. Several object detectors are available beforehand to choose from and they are identified by some name. Also, the minimum confidence score for a box can be set, which is a natural number from 1 to 100 and it represents the percentage, the maximum IOU percentage for NMS that two boxes can have, which is represented in the same way as the minimum confidence score, and the maximum number of bounding boxes that can be predicted can also be set, which can be a natural number between 1 and 111. Also, the configuration is persisted on the mobile device.

Perform object detection on static images

The user can either take a photo or choose an image from the phone's gallery. The object detection is performed on the given image and the image is shown together with the bounding boxes, the class label, and the scores.

Perform OCR on static images

The user can see the result of performing OCR on the selected image. Unlike in the case of object detection, OCR is not performed automatically but is required explicitly by the user.

Perform object detection on live images

This feature has its own screen, which is filled with live input from the camera. The live images will be fed to the object detector and the bounding boxes will be shown on the screen whenever they are available.

Perform accessible object detection on live images

The setup is the same as in the case of simple live object detection, but the live images and the predictions are not shown on the screen. Only information about the detected objects, more specifically how many objects there are of some type and the label, is converted to sound and played using the phone speakers. Also, OCR is applied in a live manner and whenever some text is detected, it is also converted to sound and played using the phone speakers.

All functionalities are represented using a use case diagram 4.12.

There are two actors in 4.12. One is the actual human user, and the other is an external OCR web service.

The main non-functional requirement is related to the speed of the object detector. Ideally, in order to resemble real-time object detection, the system must detect images at a rate of 60 FPS, but given our problem, only a few FPS are needed. As low as 5 FPS is enough for our problem.

This is the case mainly because in the accessible mode there are no images to display, therefore the 60 FPS that are required by the human eye to perceive real-time are not needed. While the sound is playing, the object detector must be fast enough to make the predictions, hence the need for at least a few FPS.

Also, while some sound is playing, the detector can make multiple sets of predictions. In order to improve usability, these sets of predictions are merged using NMS. Otherwise, too many sounds would be played.

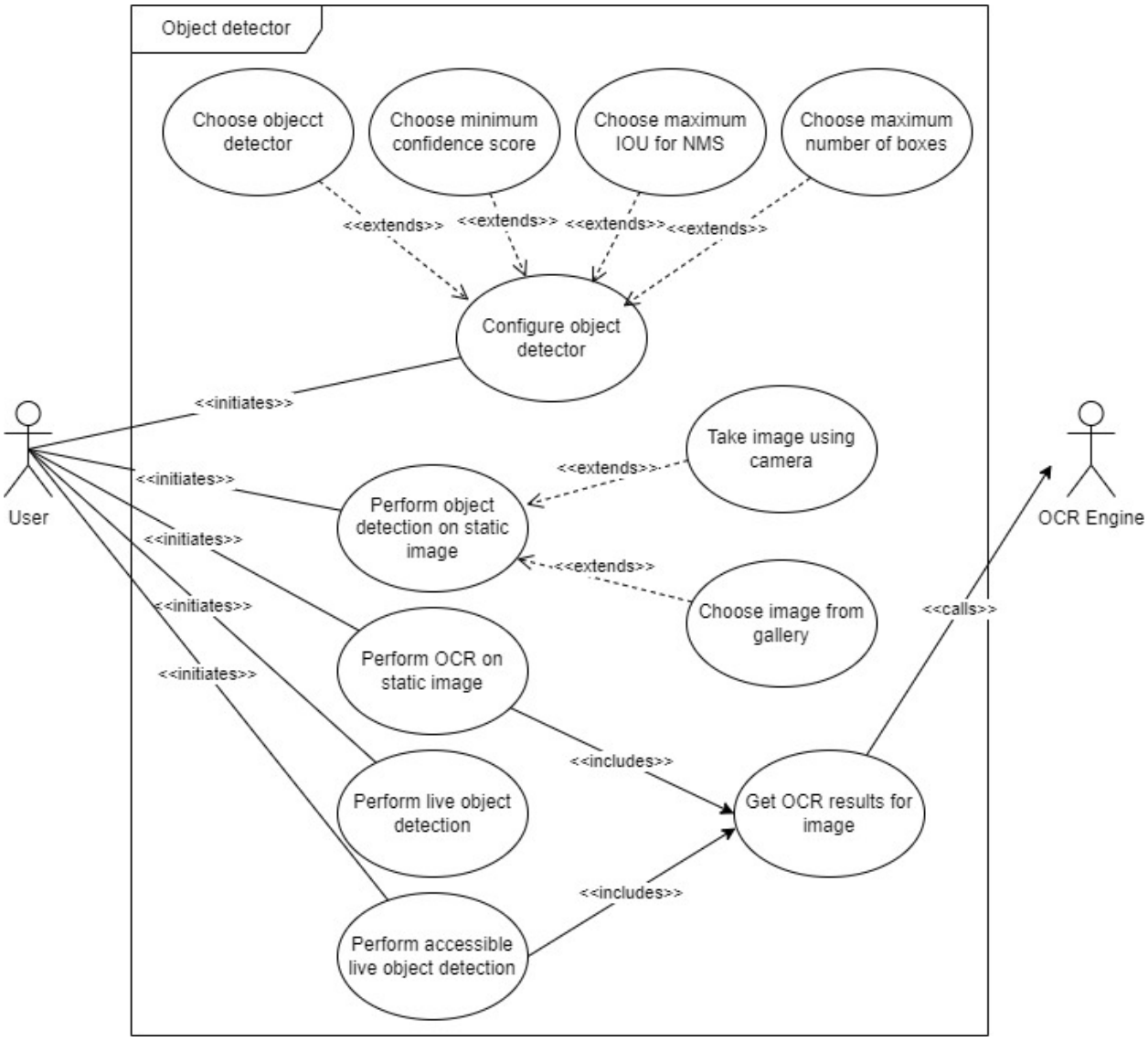


Figure 4.12: Use case diagram

4.2.2 Analysis

During the requirements elicitation, we have provided in natural language an overview of the application functionalities. Due to the ambiguity of natural language, the aim of the analysis activity is to make the requirements clearer by detailing the flows and interactions between the actors and the system for each use case, in order to limit subjective interpretations.

Configure the object detector parameters

The user can choose in any order which attributes wants to change, and not all parameters need to be changed. Also, as we have represented in 4.13, firstly the new values are chosen, then an update button is triggered in order to persist locally the new configuration.

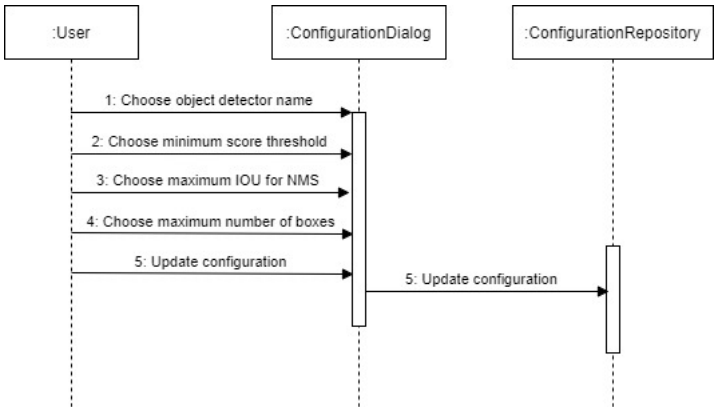


Figure 4.13: Sequence diagram for setting configuration configuration

This use case is contained in a dialog window and all the available settings the user can select are valid.

Perform object detection on static images

This use case has two scenarios, as in 4.14. In the first scenario, the user presses a button that allows him to take a photo with the built-in back camera of the mobile device. In the other scenario, the photo is chosen from the phone's gallery.

Afterward, in both cases, the image is forwarded to the object detector, which returns the predicted bounding boxes. While the predictions are made, all buttons on the screen are disabled, the original image is shown on the screen, and a progress bar will be shown.

Once the predictions are computed, the progress bar disappears, the buttons become enabled, and the bounding boxes are drawn over the image.

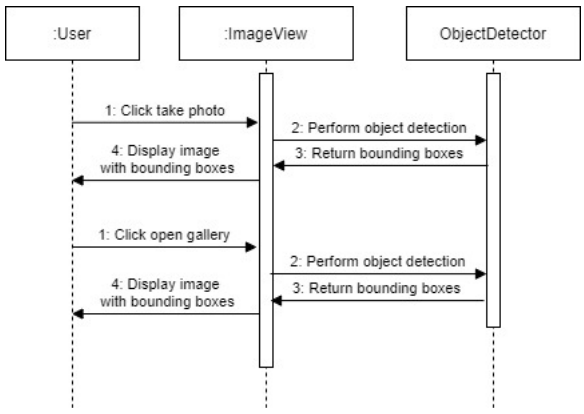


Figure 4.14: Sequence diagram for object detection on static images

Perform OCR on static images

This case also has two scenarios, as in 4.15. One scenario describes what happens if the API call to the OCR engine is successful, and the other if there is an error. If

multiple OCR engines are available to use, then each should be called until a success is met.

The user can trigger the OCR by pressing a button only if an image was already loaded by using the previous use case. After the OCR begins, all buttons are disabled, and a progress bar is shown.

Depending on the result, an error message or the detected text is displayed, the buttons become enabled and the progress bar disappears.

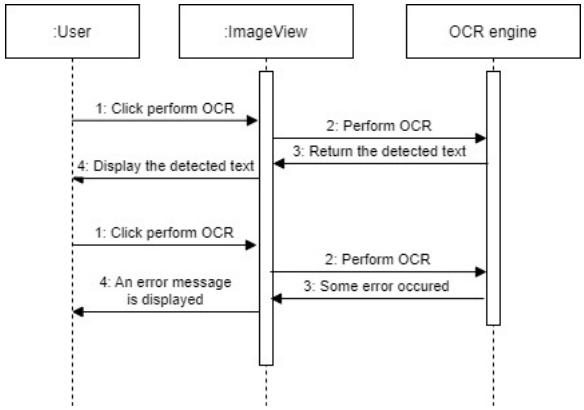


Figure 4.15: Sequence diagram for OCR

Perform object detection on live images

For this use case, the user only needs to access the corresponding screen, as in 4.16, which is filled with the live input from the back camera of the mobile device. Then, automatically, the images are fed to the object detector, and when the predictions are available, they are drawn on the screen.

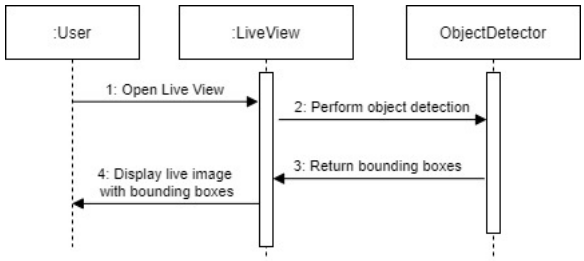


Figure 4.16: Sequence diagram for live object detection

Perform accessible object detection on live images

Similar to the previous use case, the live input from the back camera of the mobile device is fed to the object detector, but the difference is that there is no visual output, only auditory output. As is described in 4.17, the predictions are converted to text, then sound, and finally, they are played on the mobile device's speakers.

If there are any predictions made during the playing of the sound, then they are merged using NMS and when the previous sound has finished playing, the new boxes are converted to sound, and the process repeats.

Also, in parallel, calls to the OCR engine are made and whenever text is detected, it is converted to sound and played immediately.

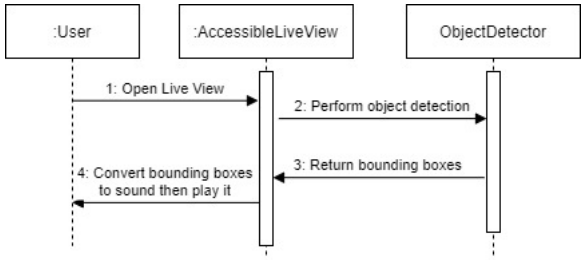


Figure 4.17: Sequence diagram for accessible live object detection

4.2.3 System design

The goal of system design is to refine the analysis model and define various subsystems that together compose the system. Basically, so far, we have defined what the system should do, but starting with this activity, we will define how the functionalities are implemented. Also, we need to define the software and hardware strategy.

In terms of hardware, the system will run on mobile devices, and in terms of software, we use Kotlin programming language on Android. For running the object detector, we are using TFLite. Therefore, our system only runs on Android mobile devices.

We describe our system decomposition using a component diagram 4.18. There are three types of subsystems. The Database and OCR API are supporting systems, which handle the persistence and the OCR.

The Configuration subsystem uses the Database subsystem in order to persist changes to the configuration and it provides a service called Configuration Repository. Also, the Object detection subsystem uses the Configuration system, besides handling the object detection logic which consists of predicting bounding boxes for a given image. The OCR subsystem handles the logic of performing OCR on an image and uses the OCR API subsystem. These two subsystems provide the Detector and OCR services.

The subsystems that the user interacts with directly are the Image subsystem, which handles object detection and OCR on static images. Also, here the user can access the configuration dialog in which the object detector's parameters can be set. The Live subsystem handles both simple and accessible live object detection. These higher-level subsystems use the services provided by the lower-level subsystems.

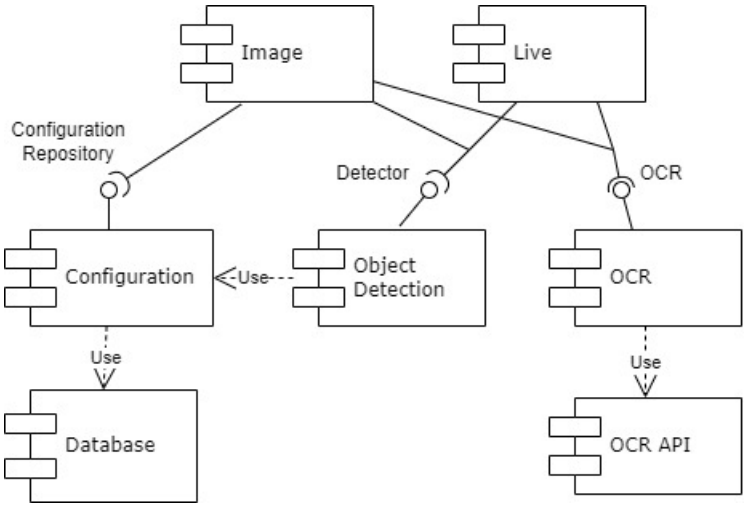


Figure 4.18: Component diagram

4.2.4 Object design

So far, we have given a high-level overview of the system. The aim of the object design activity is to expand the system design model in order to bridge the gap between the conceptual model and the code implementation. Therefore, we will give detailed explanations of the classes that compose each subsystem by specifying the fields, methods, and the relations between them.

Database

In order to persist locally the configuration, we use the Room persistence library from Android. This is an abstraction over SQLite and facilities the interaction with the database.

OCR API

In order to perform OCR on images, we use a remote free API provided by [37]. Several endpoints are provided through which two separate OCR engines can be accessed. We are interested more in the POST endpoint, which requires an image encoded in base 64, the engine we wish to use, which can be either 1 or 2, and a valid API key. For the other parameters that can be set, we use the defaults that are explained in [37].

The advantages of this API are that it is free, fast, and easy to use, but the disadvantages are that sometimes the OCR engines are down on purpose to reduce the heavy traffic and the number of images that can be sent is limited. There is also a paid option, for which the engines are always up, and more images can be processed.

Configuration

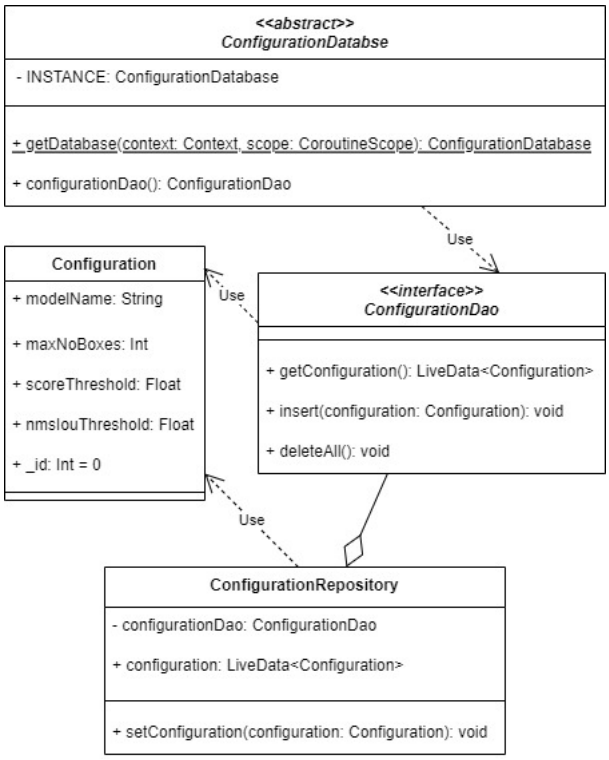


Figure 4.19: Configuration subsystem class diagram

This subsystem handles the persistence of the configuration. The ConfigurationDatabase class 4.19 is an abstract class which extends the RoomDatabase class. This is essentially a singleton through which the ConfigurationDao can be accessed. This is an interface implemented behind the scenes by the Room persistence library and it exposes several methods that are generated based on SQL commands written explicitly. Also, the ConfigurationRepository handles the access to the ConfigurationDao and provides a LiveData object which behaves like an observable object. Basically, when the configuration is modified in the database, this object will notify any subscribed observers.

Usually in Java, for example, the Configuration class would have the fields private and there would have been some getters and setters, but in order to achieve this in Kotlin the fields are represented as in 4.19. Particularly, this is a data class.

Object detection

The Detector class from 4.20 encapsulates the object detection logic. Because detection it's a computationally complex process, only one image should be processed at any given time, therefore this class is a singleton.

Other important methods include NMS and drawing on a canvas or on an image. The merge functionality just combines two lists of DetectionResults and applies

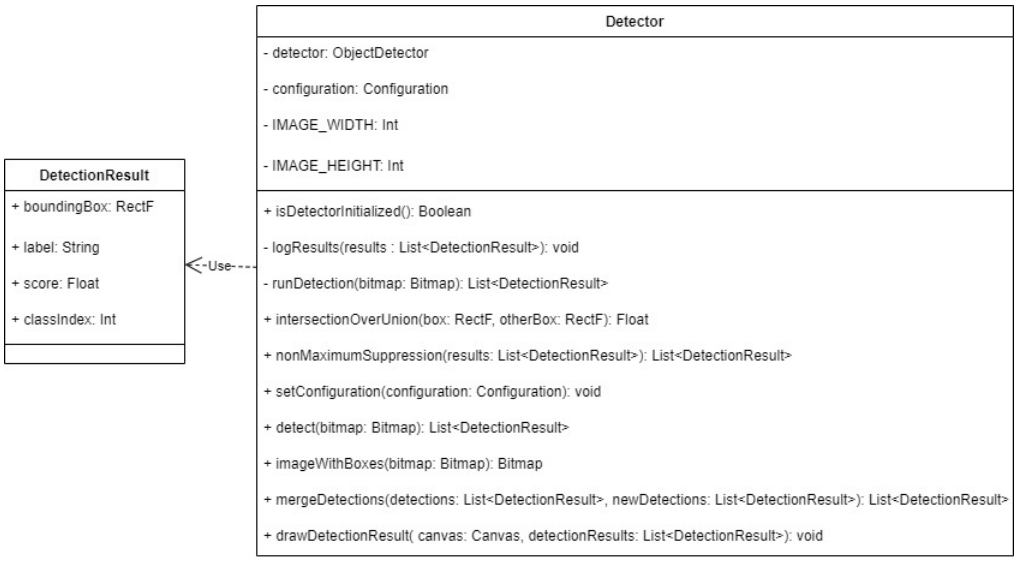


Figure 4.20: Object detection subsystem class diagram

NMS on the result.

Also, the specific behavior at runtime is determined by the configuration field.

OCR

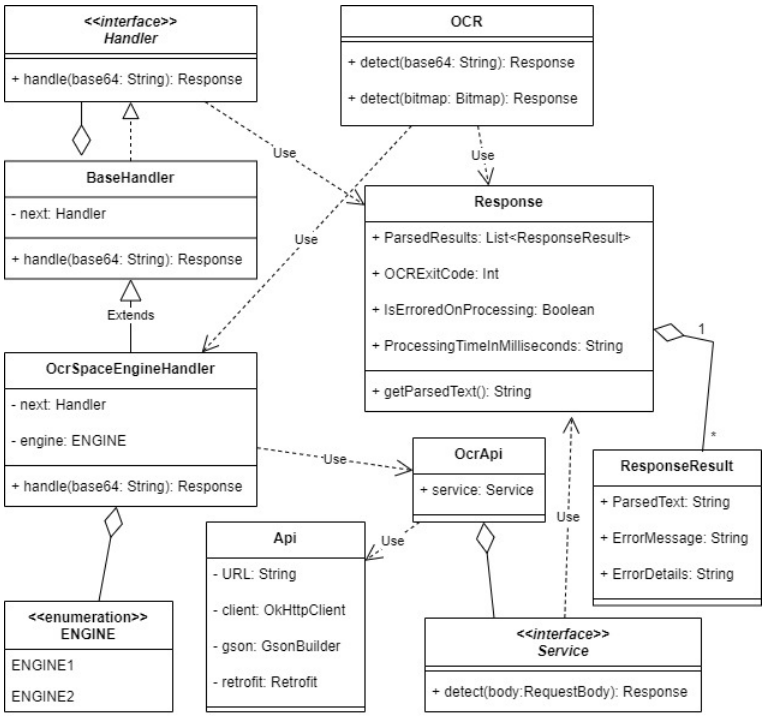


Figure 4.21: OCR subsystem class diagram

The classes needed for calling the OCR API [37] are detailed in 4.21. As we have explained earlier, there are two engines available, and in the future, there is the possibility to add another OCR API, and if the call to the first engine does not work,

then the second one should be called, and if this one does not work, then an error is given, otherwise, if there is another OCR API available, then that one is called and so on.

We implement this kind of behavior by using the chain of responsibility design pattern. To do this, we need a Handler interface and a BaseHandler which has a Handler and is itself a Handler. The handle operation just calls the handle operation of the next Handler.

The class that calls the API is the OcrSpaceEngineHandler. This way, if the need arises in the future for another OCR API, only another Handler class which inherits from BaseHandler needs to be added, plus the classes that handle the low-level API call.

In our case, the Response class only has a subset of the available fields described in [37].

Image

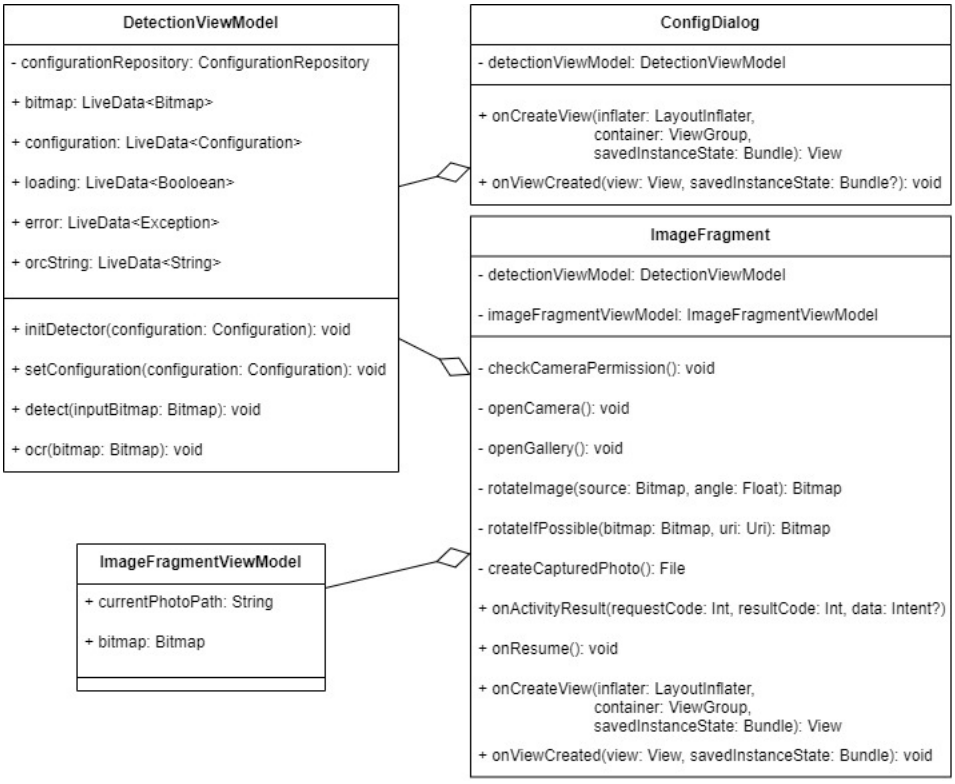


Figure 4.22: Image subsystem class diagram

In 4.22 we describe the Android specific classes needed to implement the screen which serves the functionalities for setting the configuration and performing object detection and OCR on static images.

When implementing Android applications, the Model-View-ModelView architectural design pattern is used most of the time. Meaning that we have the Im-

ageFragment and ConfigDialog, which inherit the Fragment and DialogFragment respectively Android classes which hold the logic of the interface. As the name suggests, the ConfigDialog is a dialog window that allows the user to set the configuration and it can be opened through a button on the ImageFragment.

The DetectionViewModel and ImageFragmentViewModel both extend the AndroidViewModel class, and their purpose is to hold the state of the fragments across the activity lifecycle.

Live

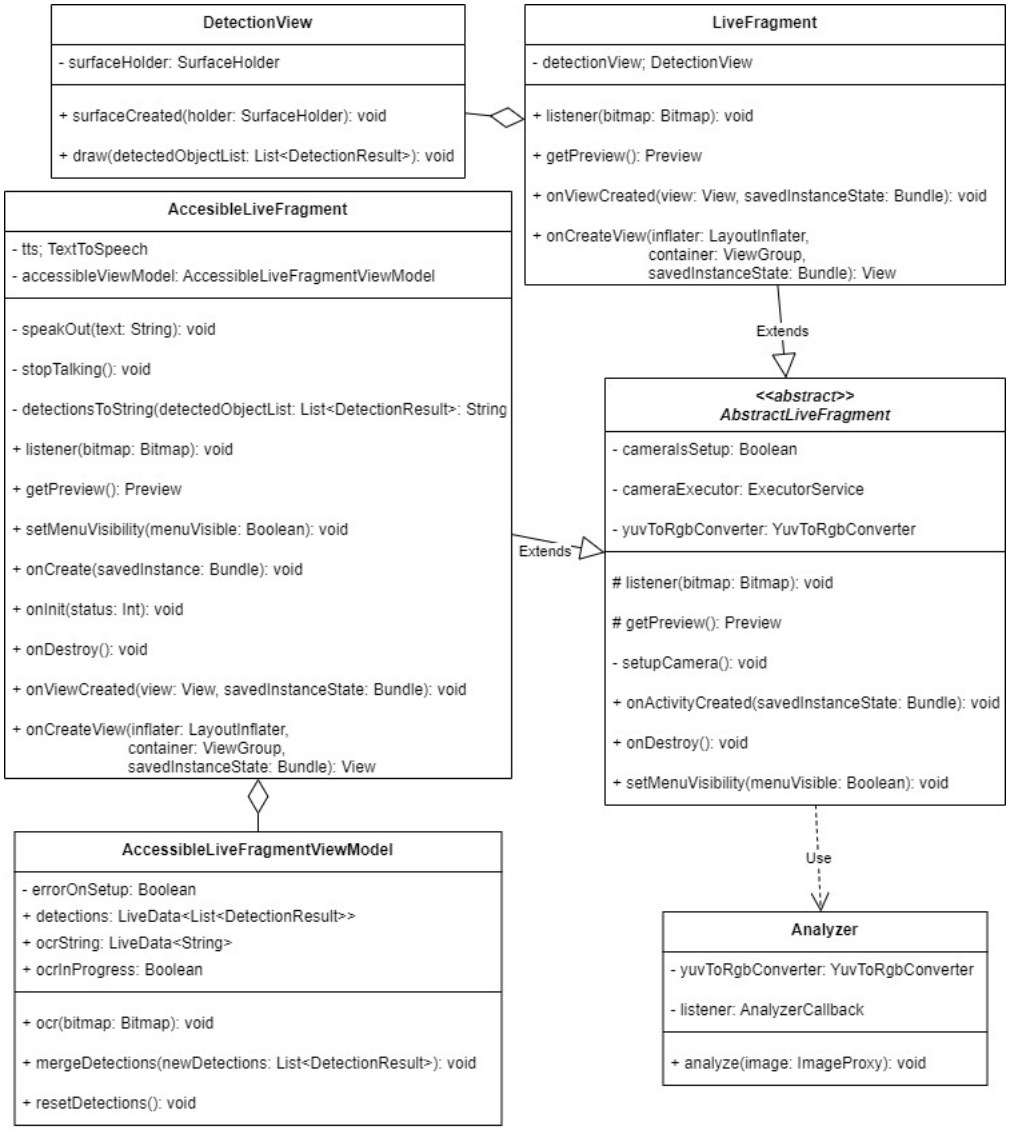


Figure 4.23: Live subsystem class diagram

Since the screens for simple live object detection and accessible live object detection have the same main logic, only the display is different. We have chosen to put the common behavior in the AbstractLiveFragment 4.23, which, together with the

Analyzer, takes the stream of images from the camera, and passes them one by one to the listener method, which is abstract. Both the LiveFragment and AccessibleLiveFragment have the same meaning as the previous fragments and implement the listener method. In LiveFragment, the listener method only calls the Detector class and draws the resulting bounding boxes on the screen. In AccessibleLiveFragment there is no need for drawing, but the results are merged with the already existing results and OCR is performed on the image.

Also, the AccessibleLiveFragmentViewModel has the same meaning as the other view models and the idea is that it notifies the AccessibleLiveFragment to play the text identified by the OCR whenever it is available, or the text resulted from object detection. The merge operation simply merges the detection results using the function from the Detector class, while there are sounds playing.

The DetectionView is simply a view on which the bounding boxes are drawn.

4.2.5 Implementation

During this activity, the actual source code is developed, following the guidelines defined in the previous activities. We are also going to discuss some details about the implementation.

TensorflowLite

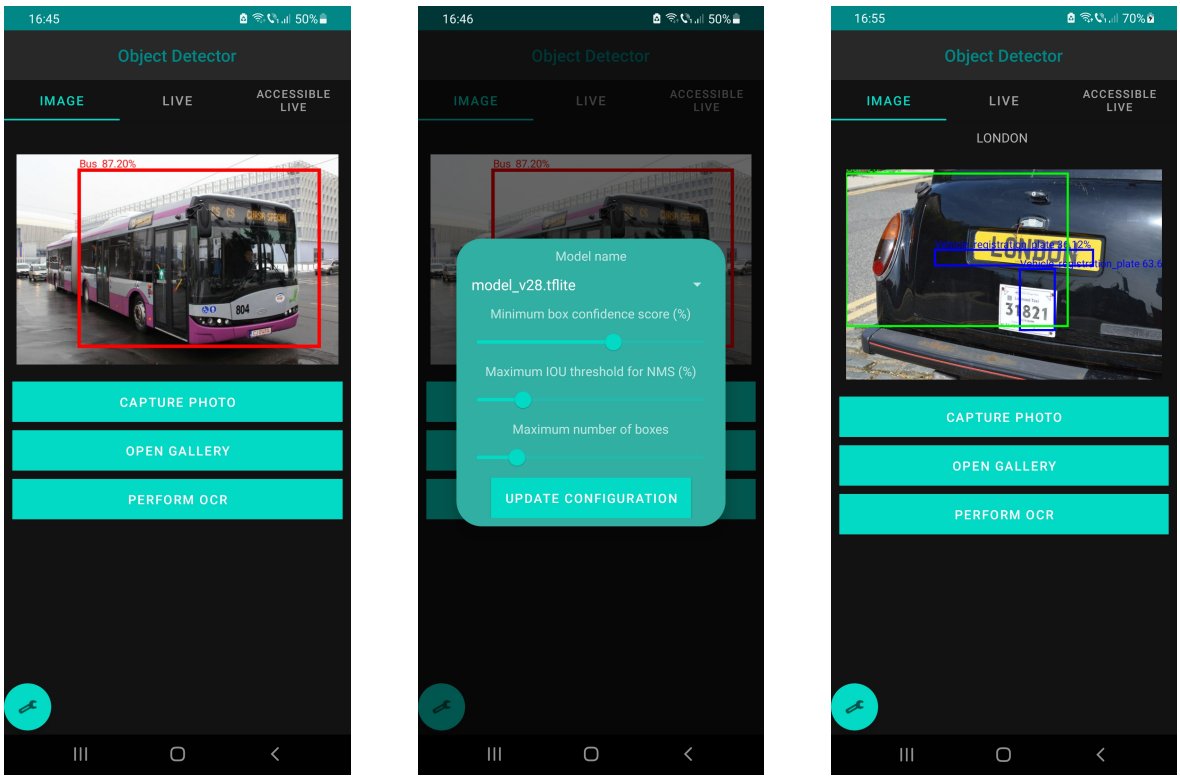
Firstly, the object detection system is run using the TensorflowLite Task Library, more specifically the object detector that it's already implemented. In order to use this class, the model must be stored in a .tflite file and it must meet some compatibility requirements. The input must be of size $1 \times height \times width \times 3$ because currently batch inference is not supported and the input must be an RGB image, hence the last dimension is 3. Also, if the input is of type float, then metadata about the normalization of the input must be present in the file, i.e. the mean and standard deviation to which the input should be normalized.

The output needs to be composed of four components. The first must be the locations tensor which needs to be of type float32 and of size $1\times$ *number of results* $\times 4$ because the locations are given by the top left and bottom-right coordinates of the bounding box. The second output must be of type float32 and of size $1\times$ *number of results* and it represents the class index for each prediction. Additionally, a label map is specified in the model file. The third tensor represents the scores tensor and it has the same type and size as the previous one and it stores the probability for each prediction. The last output represents the number of predictions, and it is represented as a tensor of size 1 and of type float32.

The model is run on a CPU and the number of threads is equal to the available number of CPU cores.

User interface

As we have specified in earlier sections, there are three screens in total. We implement this using an Android pager. This way the user can swipe between the screens.



(a) After loading an image (b) Configuration dialog (c) After performing OCR

Figure 4.24: Screen with static image functionalities

The first screen of the application covers the configuration dialog 4.24b, the object detection 4.24a and OCR 4.24c on static images.

The second screen is entirely dedicated to live object detection 4.25.

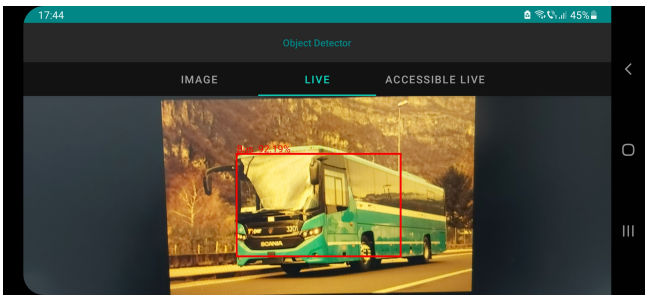
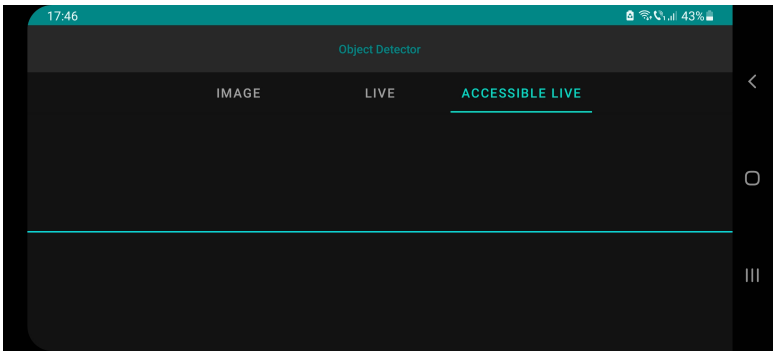
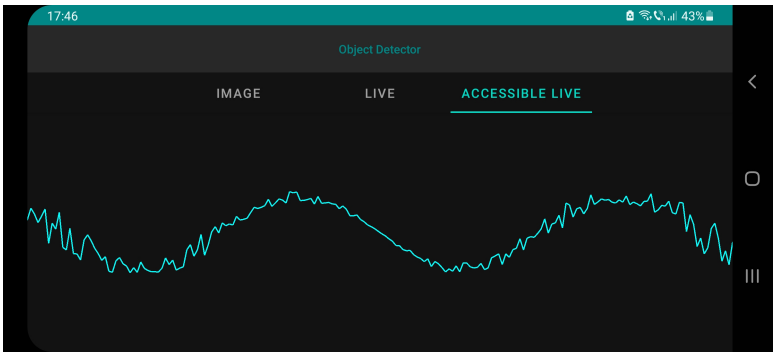


Figure 4.25: Live object detection screen



(a) No sound is playing



(b) Sound is playing

Figure 4.26: Accessible live object detection

The third screen handles the accessible live object detection 4.26 and it is composed of a state in which no sound is playing 4.26a, and a state in which some sound is playing 4.26b.

Version control

Regarding version control, we are using git and GitHub and both the object detector, and the mobile application are available at https://github.com/ComanacDragos/PublicTransportDetector.

4.2.6 Testing

The scope of this activity is to check if the implemented system behaves correctly. There are several ways to test an application, but we have chosen mainly manual testing because the main functionalities are based on the processing in some way of an image and simply displaying the results. Therefore, the object detector itself is tested manually by computing the mean average precision and by using it on a test set. Also, data augmentation techniques are tested manually because they involve strictly images.

On the Android side, the reasoning is similar because we only display the output of an object detector in various ways. The difference is that here we have imple-

mented NMS from scratch and, because it is an important algorithm in the detection pipeline, we have chosen to create some unit tests using a white-box approach because the source code is available.

Firstly, we are going to test the IOU algorithm because the NMS algorithm depends on it. We choose two extreme cases, one in which the boxes are fully overlapped and the other when the boxes do not overlap at all. Then we choose two cases in which they partially overlap on the horizontal axis and on the vertical axis.

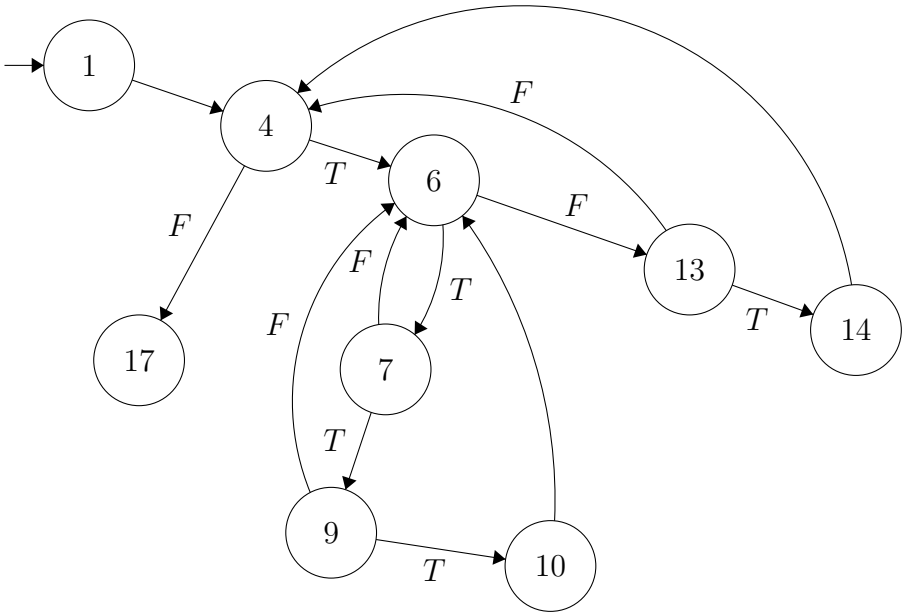


Figure 4.27: Control flow graph of NMS

The NMS algorithm is a bit more complex because it contains several loops and conditions. Therefore, we represent the possible paths using the control flow graph 4.27 of the corresponding source code 4.1, where each node represents the line in the source code and the outer edges represent the next possible line that should be executed. In the case of if statements, the outer edges are annotated with T and F and they indicate the path when the condition is true, respectively when it's false. In a similar way, the for statements are annotated, but in this case, T means that there are more elements to be processed, and F means that the for statement is done.

Some lines have been skipped in the control flow graph for simplicity because there is no choice but to go forward.

The cyclomatic complexity indicates the number of independent paths that can be taken in 4.27. It can be computed by counting the number of predicates and then adding one. In this case, the cyclomatic complexity is $6 = 5 + 1$.

The test cases that we designed cover the individual paths in the control flow graph and are implemented in Kotlin using JUnit4 and following the Arrange, Act and Assert pattern.

```
fun nonMaximumSuppression(results: List<DetectionResult>): List<DetectionResult>{
    val newResults = LinkedList<DetectionResult>()
    results.sortedByDescending { it.score }
        .forEach{
        var maxIou = -1f
        for(result in newResults){
            if(result.label == it.label){
                val currentIou = intersectionOverUnion(it.boundingBox, result.boundingBox)
                if(maxIou < currentIou)
                    maxIou = currentIou
            }
        }
        if(maxIou < configuration.nmsIouThreshold/100)
            newResults.add(it)
    }
    return newResults
}
```

Listing 4.1: NMS method

5. Experimental results

In this chapter, we will define the performance metrics that we use to evaluate our model. Then we will discuss how we have tuned the hyperparameters such that we achieve the best results. Afterward, we present these results and finally we compare our model with other existing methods.

5.1 Perfomance evaluation

To measure an object detection system performance usually frames per second (FPS) and mean average precision (mAP) are used. FPS simply means the number of images, or frames, an algorithm can process in a single second. In order to explain mAP, we first need to define precision and recall.

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

Precision 5.1 is a measurement in percentages and it indicates how reliable are the algorithm's predictions. In other words, if the algorithm predicts a certain class, precision tells how likely is that the respective prediction is correct. This metric is computed for each class, and it is the ratio between the number of true positives (TP) and the number of all positives, including the false positives (FP). By positive, we mean the class for which the precision is computed, therefore a TP represents a prediction that is correct, and a FP represents a prediction for the respective class when the class is not present.

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

Recall 5.2 is another similar metric that measures the percentage of found positives, also known as the true positive rate. It is also computed for each class, and it tells how well the algorithm can find all instances of a given class. The formula is similar to the one for precision, but this time it is the ratio between TP and the sum between false negatives (FN) and TP. A FN occurs when the algorithm doesn't predict a class when it should.

Depending on the problem, what is a true positive can be defined in different

ways, usually depending on some threshold. In the case of bounding boxes, a true positive occurs when the predicted bounding box has a high enough IOU with the ground truth box.

To compute mAP, first, the predictions are sorted descending by their confidence score. Then, the predictions are parsed one by one, and at each step, the precision and recall are computed, taking into consideration only the parsed prediction up until that point. For the recall, we consider all positives, including those that were not parsed. If we would plot these values, with the recall on the horizontal axis and the precision on the vertical axis, we would get what is called the precision-recall curve. Average precision is defined as the area under the precision-recall curve and mAP is defined as the mean of the average precisions for each class. We use the official implementation from [9] in order to compute the average precision, given the precision and recall, for a given image.

5.2 Hyperparameter tuning

Hyperparameters are different from normal parameters in the sense that they are not trainable, and they are more related to the configuration of the system. For example, in 5.3, x is a parameter and a is a hyperparameter.

$$f(x) = a \cdot x \tag{5.3}$$

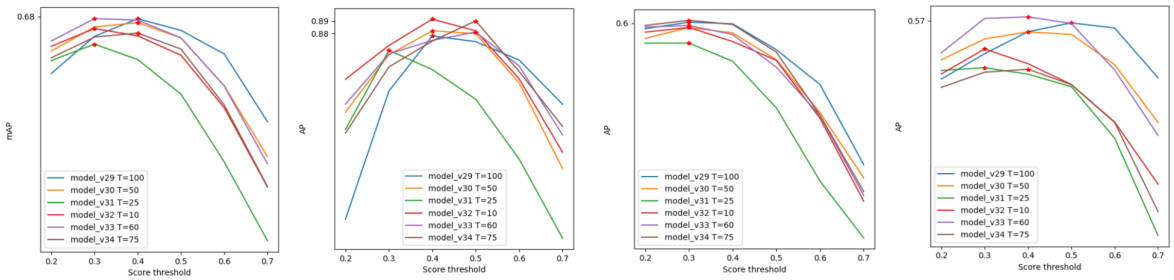
Hyperparameters are important because the performance of an algorithm can be improved by simply tuning the hyperparameters. This is not a trivial task, especially when there are a lot of hyperparameters and the training time is large, which happens most of the time. We select some hyperparameters and explore how they impact the mAP. Also, in order to better comprehend the differences, we compute the mAP at different score thresholds, which results in a curve that we use to compare different models that were trained in different hyperparameter setups. In general, we use a true positive threshold of 50% and a NMS threshold of 30%.

We start by analyzing the scheduler. In the formula from 4.6, we use for the maximum learning rate a value of 10^{-3} and for the minimum learning rate a value of 10^{-6}. We compare how different values for the restart epoch parameter T influence the mAP. We detail in 5.1 the value for T used in training each model.

Model	v29	v30	v31	v32	**v33**	v34
T	100	50	25	10	**60**	75
mAP	67.7%	67.17%	64.23%	66.36%	**67.72%**	65.74%

Table 5.1: Restart epoch values

In 5.1 we can see the AP and mAP curves for the values in 5.1. All three classes are represented in 5.1b, 5.1c, and 5.1d. The mAP is represented in 5.1a.



(a) mAP (b) Bus (c) Car (d) License plate

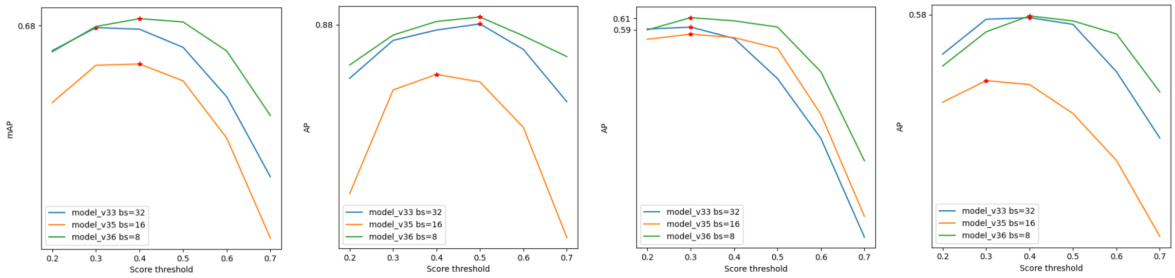
Figure 5.1: AP and mAP variation for different values of the restart epoch

We can tell from 5.1 that a value of 60 yields good results across all classes. These models were trained for 50 epochs, and we can see that in general, values under 50 for the restart epoch give slightly worse results, meaning that increasing and decreasing the learning rate doesn't help that much. Therefore, further on, we use 60 as the restart epoch.

Model	v33	v35	**v36**
Batch Size	32	16	**8**
mAP	67.72%	62.88%	**68.91%**

Table 5.2: Batch size values

We also see how the batch size influences the performance. In 5.2 we detail the batch size value used in training each model.



(a) mAP (b) Bus (c) Car (d) License plate

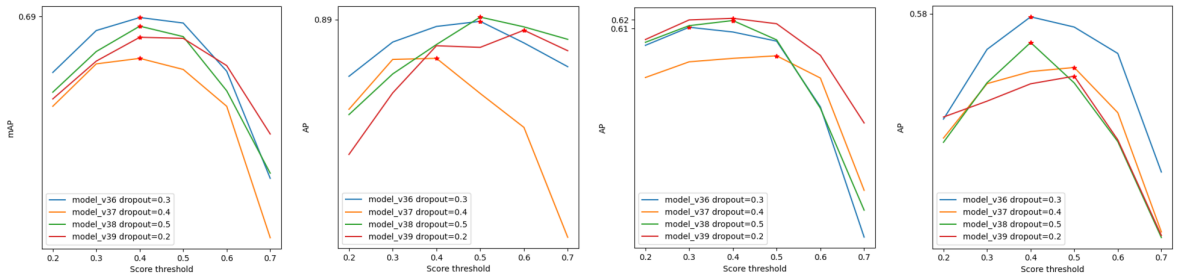
Figure 5.2: AP and mAP variation for different values of the batch size

As we can see in 5.2, a batch size of 8 yields the best results in all cases, therefore, further on, we use this value.

Model	**v36**	v37	v38	v39
Dropout	**30%**	40%	50%	20%
mAP	**68.81%**	65.64%	68.22%	67.33%

Table 5.3: Dropout values

The dropout probability is important because of its regularization effect. We study various values presented in 5.3.



(a) mAP (b) Bus (c) Car (d) License plate

Figure 5.3: AP and mAP variation for different values of the dropout probability

We can see in 5.3 that for each case, a different dropout value has the best results. We go further on with a value of 30% because it is the best in the average case, and it is also good in the individual classes.

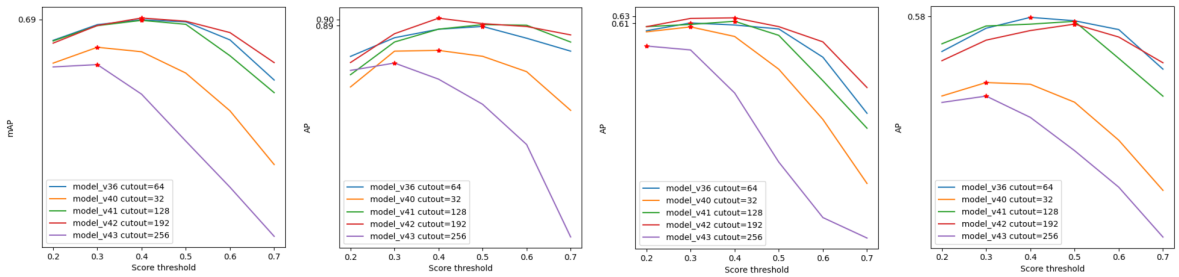
So far, we have tuned parameters related to training or to the model itself. Finally, we will see how the data augmentation hyperparameters affect the performance.

Model	v39	v40	v41	**v42**	v43
Cutout	64	32	128	**192**	256
mAP	67.33%	63.03%	68.79%	**69.29%**	59.33%

Table 5.4: Cutout values

For the cutout data augmentation, we consider as a hyperparameter the length of the side of the cutout square. We can see in 5.4 the different values that we have chosen to see the influence of the size of the cutout patch.

In 5.4 we can see that the lowest and the highest values yield the worst results. Also, a patch of 192x192 gives good results across all classes therefore, further on we use this value.



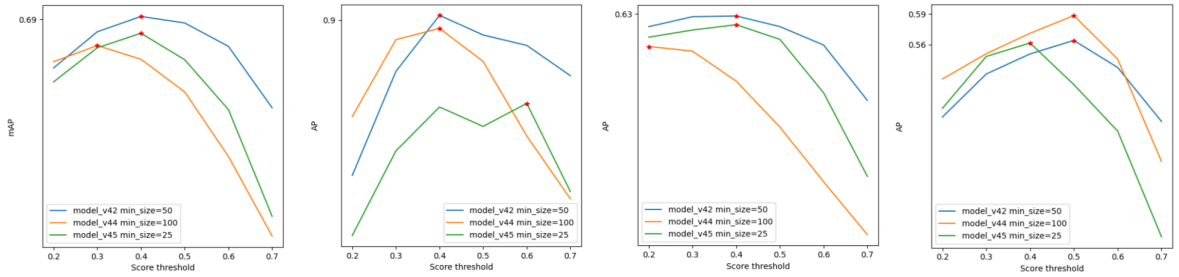
(a) mAP (b) Bus (c) Car (d) License plate

Figure 5.4: AP and mAP variation for different values of the side length for the cutout patch

Model	**v42**	v44	v45
Size	**50**	100	25
mAP	**69.29%**	66.25%	67.54%

Table 5.5: Mosaic minimum size values

For the mosaic data augmentation, we study the influence of the probability that it is applied and the minimum size that one of the four images can take. For example, when the minimum size is 50, then each image will have a size of at least 50x50. In 5.5 we describe the values for the minimum size.



(a) mAP (b) Bus (c) Car (d) License plate

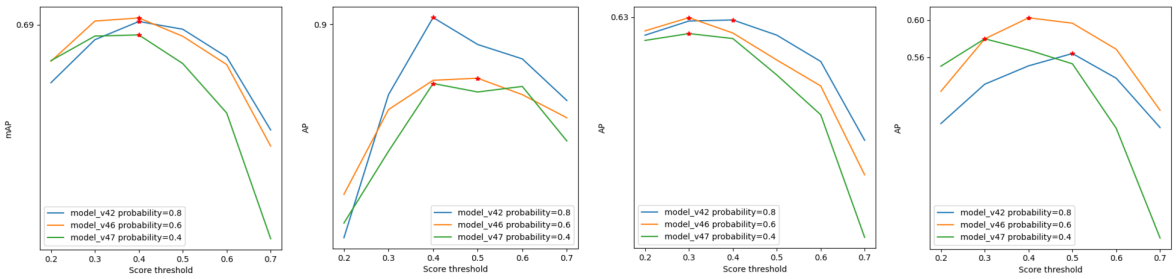
Figure 5.5: AP and mAP variation for different values of the mosaic minimum size

In 5.5 we can see that a value of 50 for the minimum size is best in general. Even though it does not benefit the license plate class that much, we choose this value because it's best for the bus class, and it has the best mAP curve.

Model	**v42**	v46	v47
Probability	80%	60%	40%
mAP	69.29%	**69.6%**	68.11%
Bus AP	**90.23%**	88.16%	87.98%

Table 5.6: Mosaic probability values

In 5.6 we have the values that we have used for the probability that mosaic data augmentation is applied during training.



(a) mAP (b) Bus (c) Car (d) License plate

Figure 5.6: AP and mAP variation for different values of the mosaic probability

In 5.6 we can see that a low mosaic probability such as 40% results in low mAP. In the average case, the models with 60% and 80% probability are close in terms of mAP. For the license plate, the model with 60% probability is a little better, but because for the bus, the mAP is much larger with a probability of 80% we consider this value to be the best. Also, it performs better for the car class too.

Random Hue	delta	0.5
Random Saturation	lower	5
	upper	10
Random Brightness	delta	0.3
Random contrast	lower	1
	upper	2

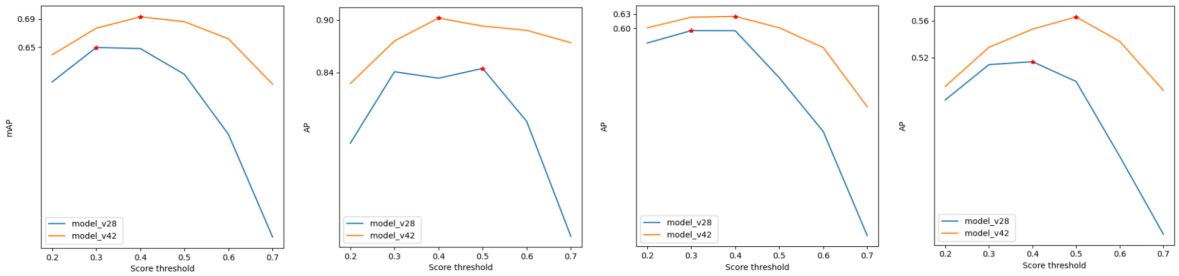
Table 5.7: Photometric data augmentation hyperparameters

In 5.7 we present the hyperparameters used in the data augmentation techniques presented in 4.8.

5.3 Results

In this section we present our final results on the subset of Open Images Dataset V4 [21], comprising three classes: bus, car, and vehicle registration plate.

Firstly, in 5.7 we present the results on two models, one before hyperparameter tuning (model_v28 with blue) and the one after hyperparameter tuning (model_v42 with orange). It is clear that significant improvements have been achieved only by finding optimal hyperparameters values.



(a) mAP (b) Bus (c) Car (d) License plate

Figure 5.7: AP and mAP for the final model (orange) and the model before hyperparameter tuning (blue)

Class	Car	Bus	License plate	Average
Before tuning	59.52%	84.07%	51.25%	64.95%
After tuning	62.54%	90.23%	55.09%	69.29%
Improvement	3.02% ↑	6.16% ↑	3.84% ↑	4.34% ↑

Table 5.8: Improvement after hyperparameter tuning in percents for each class in AP and mAP for the average case

We detail the exact improvements in percents in 5.8. To compute the exact improvement, we consider the maximum value from the mAP curve in 5.7.

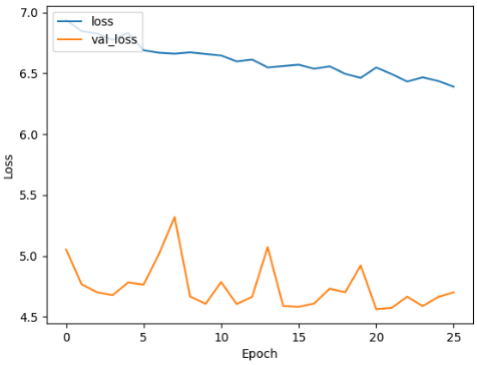
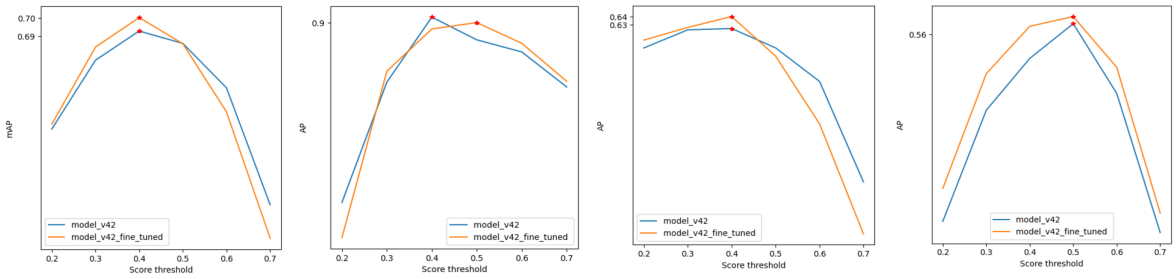


Figure 5.8: Training (blue) and validation (orange) loss for fine tuning model v42

After tuning the hyperparameters, we perform fine tuning, meaning that we unfreeze the backbone of the model and further train with a very small learning rate. In 5.8 we can see the loss curves. For fine tuning, we have used $\eta_{max} = 10^{-5}$ and $\eta_{min} = 10^{-8}$.

In 5.9 we can see that we achieved small improvements only by fine tuning and in 5.9 we detail the exact values obtained and the improvements. For the bus class there is a slight decrease, but in general the improvements are positive.

In terms of speed, our solution achieves approximately 5 FPS on a Samsung A70 mobile phone.



(a) mAP (b) Bus (c) Car (d) License plate

Figure 5.9: AP and mAP for the fine tuned model and the model before fine tuning

Class	Car	Bus	License plate	Average
Before fine tuning	62.54%	90.23%	55.09%	69.29%
After fine tuning	64.04%	90.01%	56.68%	70.03%
Improvement	1.5% ↑	0.22% ↓	1.59% ↑	0.74% ↑

Table 5.9: Improvement after fine tuning in percents for each class in AP and mAP for the average case

5.4 Comparison with other methods

State of the art object detection models such as YOLOv4 [1] achieve real-time performance and overall good detection accuracy. We use the COCO dataset [42] in order to compare our method with other existent methods. This dataset has several splits such as validation or test-dev. We test our method by performing object detection on each of these dataset splits and uploading the results to the evaluation server. We follow the guidelines regarding the data format described on the official website, where details about the evaluation server are also provided.

On this dataset, the YOLO state of the art detector achieves 43.9% mAP, SSD achieves 25.1% mAP, and the currently best mAP on the detection competition leaderboard is 63%. Our method achieves a modest 0.4% mAP on the test-dev bounding box evaluation server, indicating that there is still room for improvement. This performance might be due to the fact that our method was developed considering only three classes, and the COCO dataset has 80 classes. We also tried to develop a very small model because the resources are limited on a mobile device. For example, our model has around 14 megabytes, whereas better object detectors have around 200-300 megabytes or more, which is 10 to 20 times larger. Also, the hyperparameters are not tuned for this specific dataset.

Using our own measurement, which is based on the official PASCAL VOC mAP implementation [8], we achieve 28% mAP on the validation set. We did this computation on the validation set because the ground truth annotations are not publicly available for the test set.

6. Conclusions and future work

In conclusion, our solution aims to ease the use of public transport by visually impaired persons. The first step in doing this is creating an object detection system that can recognize busses, cars, or vehicle registration plates. This part is implemented using a custom version of YOLOv2 [30] and we obtain, on the test set, a mean average precision of 70.03%, and for the bus class, we obtain an average precision of 90.01%, for the car class 64.04% and for the vehicle registration plate 56.68%, with a speed of around 5 FPS on a mobile device. We have also trained a model on the COCO dataset that achieves around 0.4% mAP on the test dataset. The second part is represented by the mobile application, which serves both as an object detection system visualizer and as a proof of concept for assistive technology for the VIPs that uses object detection. This is illustrated by the accessible live object detection, in which the predictions are not visualized, but converted to sound and played using the mobile device speakers.

These results need more work until they reach state of the art, which is real-time speeds of 30-60 FPS and better performance in terms of accuracy. There are several parts that can be improved, and we leave them as future work. Firstly, the dataset could be enhanced with images with bad lights or weather, or night images. Recent advances have shown that data-centric AI yields better results than model-centric AI, therefore the dataset could use more attention, in the sense that bad ground truth annotations should be found and fixed. Another key point in training a neural network is the loss function. Here, focal loss is an option worth exploring. Other techniques presented in the other YOLO papers such as training with images of different sizes, could prove useful. Also, our solution could benefit from a larger dataset, and this would require access to multiple powerful GPUs.

On the Android application side, the conversion from bounding box to sound could be improved. For example, we don't use the positions of the bounding boxes, therefore valuable information is not used. The solution would be to develop a sound convention, that could convert the bounding boxes to a compact and easy-to-understand form.

Bibliography

[1] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection. *ArXiv*, abs/2004.10934, 2020.

[2] Bernd. Bruegge and Allen H. Dutoit. *Object-oriented software engineering : Using UML, patterns, and Java*. Prentice Hall, 2010.

[3] M. Augustine Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes rendus de l'Académie des Sciences*, pages 536–538, 1847.

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

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

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[7] Lamya El Alamy, Sara Lhaddad, Soukaina Maalal, Yasmine Taybi, and Yassine Salih-Alj. Bus identification system for visually impaired person. In *2012 Sixth International Conference on Next Generation Mobile Applications, Services and Technologies*, pages 13–17, 2012.

[8] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

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

[10] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

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

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[13] 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, 2016.

[14] Geoffrey Hinton. Divide the gradient by a running average of its recent magnitude. *Coursera*, lecture 6, 2012.

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

[16] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[17] Saeed Khazaee, Ali Tourani, Sajjad Soroori, Asadollah Shahbahrami, and Ching Y. Suen. A Real-Time License Plate Detection Method Using a Deep Learning Approach. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12068 LNCS, pages 425–438, 2020.

[18] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

[19] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Shahab Kamali, Matteo Malloci, Jordi Pont-Tuset, Andreas Veit, Serge Belongie, Victor

Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html*, 2017.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[21] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, and et al. The Open Images Dataset V4. *International Journal of Computer Vision*, 128(7):1956–1981, Mar 2020.

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[23] Shuying Liu and Weihong Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734, 2015.

[24] 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*, page 21–37, 2016.

[25] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *International Conference on Learning Representations*, 2017.

[26] Wai-Ying Low, Mengqiu Cao, Jonas De Vos, and Robin Hickman. The journey experience of visually impaired people on public transport in london. *Transport Policy*, 97:137–148, 2020.

[27] Anh Huynh Ngoc. YOLOv2 implementation. Accessed: 25.03.2022, https://github.com/experiencor/keras-yolo2, 2019.

[28] Geneva: World Health Organization. World report on vision. *World Health Organization Publications*, page 77, 2019.

[29] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

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

[31] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39:1137–1149, 2015.

[32] RenewSenses. Travis, the ultimate personal assistant in a discreet add-on to you smartphone. Accessed: 27.03.2022, https://renewsenses.com/, 2021.

[33] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *MICCAI 2015. Lecture Notes in Computer Science*, 2015.

[34] Mohammad Amin Sadeghi and David Forsyth. 30hz object detection with dpm v5. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 65–79, Cham, 2014. Springer International Publishing.

[35] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[36] Laurent Sifre. *Rigid-Motion Scattering For Image Classification*. PhD thesis, Ecole Polytechnique, Centre de Mathématiques Appliquées, 91120 Palaiseau, France, October 2014.

[37] A9T9 software GmbH. Ocr space: Free ocr api. Accessed: 13.04.2022, https://ocr.space/.

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

[39] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[40] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[41] Angelo Vittorio. Toolkit to download and visualize single or multiple classes from the huge Open Images v4 dataset. Accessed: 25.03.2022, https://github.com/EscVM/OIDv4_ToolKit, 2018.

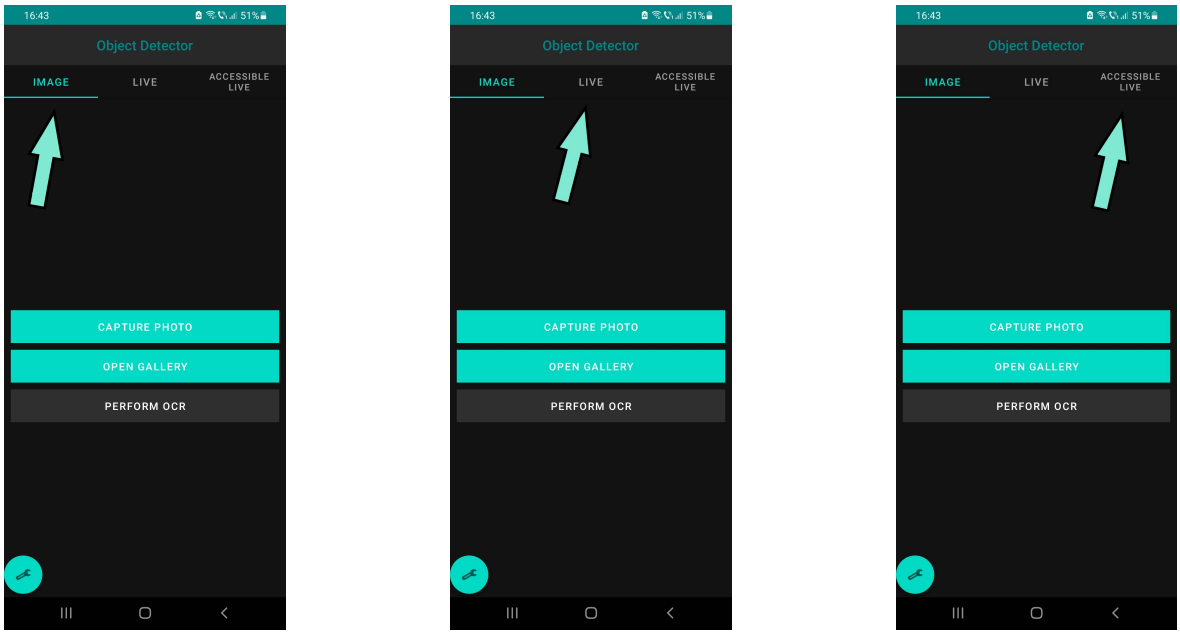
[42] Tsung-Yi LinMichael MaireSerge BelongieJames HaysPietro PeronaDeva RamananPiotr DollárC. Lawrence Zitnick. Microsoft coco: Common objects in context. *European conference on computer vision (ECCV)*, pages 740–755, 2014.

Appendices

A. User manual

In what follows, we will present a guide that a common user can follow in order to use our application. To do this, we will present for each functionality how it can be accessed and how it can be used.

First of all, the application is composed of three screens, one for static images related functionalities, one for live, and one for accessible live respectively. The first screen that can be seen when starting the application is A.1.



(a) Image screen button (b) Live screen button (c) Accessible live screen button

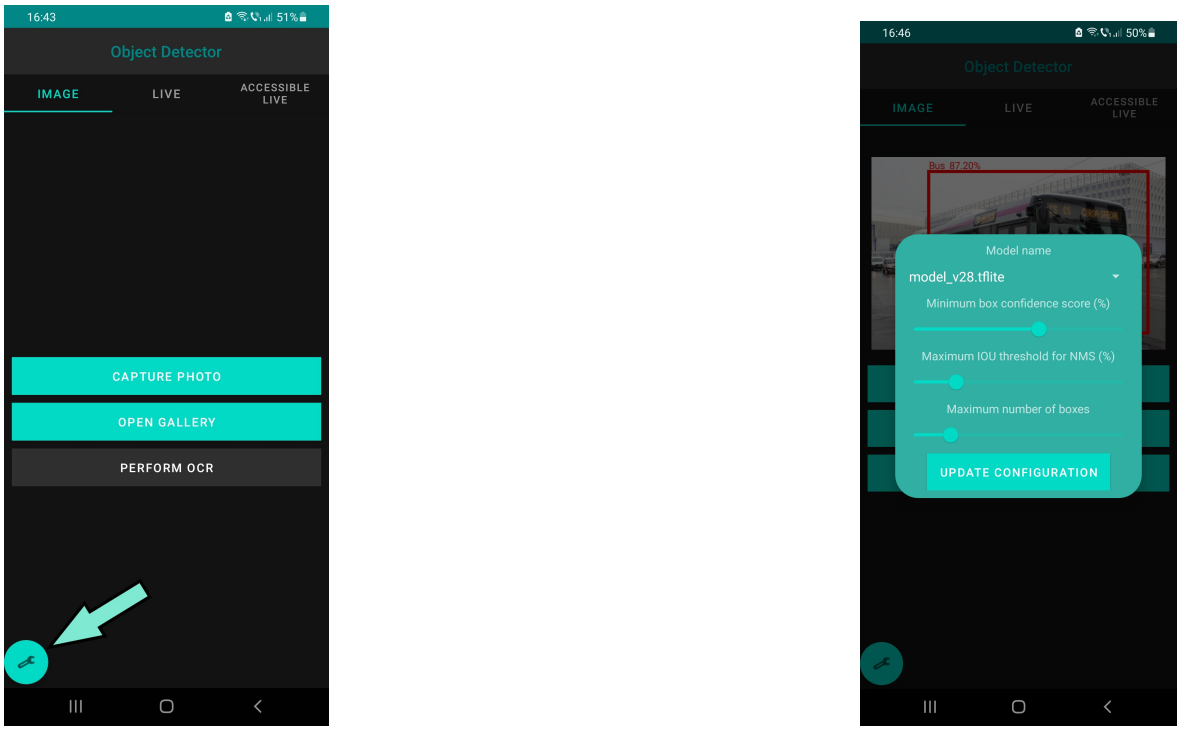
Figure A.1: Starting screen

Each screen can be accessed by swiping to the left or to the right until the desired screen is reached. Also, for each screen, there is a button that can be pressed in order to reach the respective screen, as in A.1a, A.1b and A.1c.

Configure the object detector parameters

In order to configure various parameters of the object detector, the configuration dialog must be opened. This can be achieved by pressing the corresponding button,

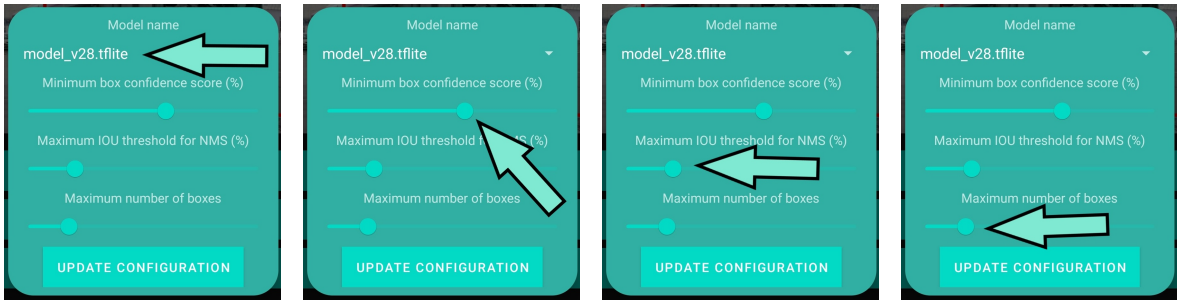
which is located in the bottom left corner of the first screen A.2a. After pressing the button, the configuration dialog is displayed A.2b. If any region outside the configuration dialog is pressed, the changes are discarded.



(a) Configuration button (b) Configuration dialog

Figure A.2: Open configuration dialog

The model's name can be set by choosing an available name from the spinner A.3a, the confidence score can be set by using the slider A.3b, the maximum IOU used by NMS can be set by using the slider A.3c, and the number of boxes can be set by using the slider A.3d.



(a) Set model name (b) Set confidence score (c) Set IOU for NMS (d) Set number of boxes

Figure A.3: Configuration parameters

At the end, in order to save the changes made to the configuration, the update button must be pressed A.4.

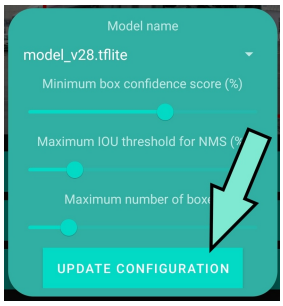
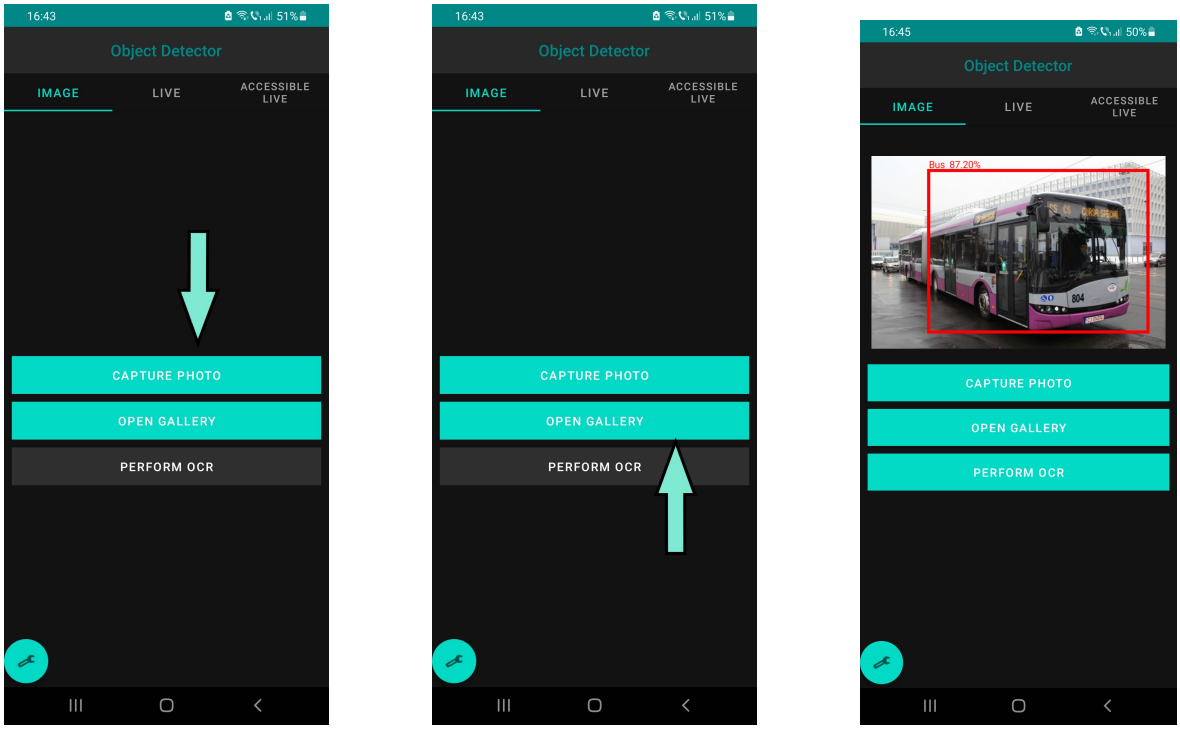


Figure A.4: Update configuration button

Perform object detection on static images

In order to perform object detection on static images, firstly an image must be loaded either by taking it with the mobile device's camera or by accessing the gallery and choosing an image. This can be achieved by pressing the capture photo button A.5a or by pressing the open gallery button A.5b. The results will be shown once they are available as in A.5c.



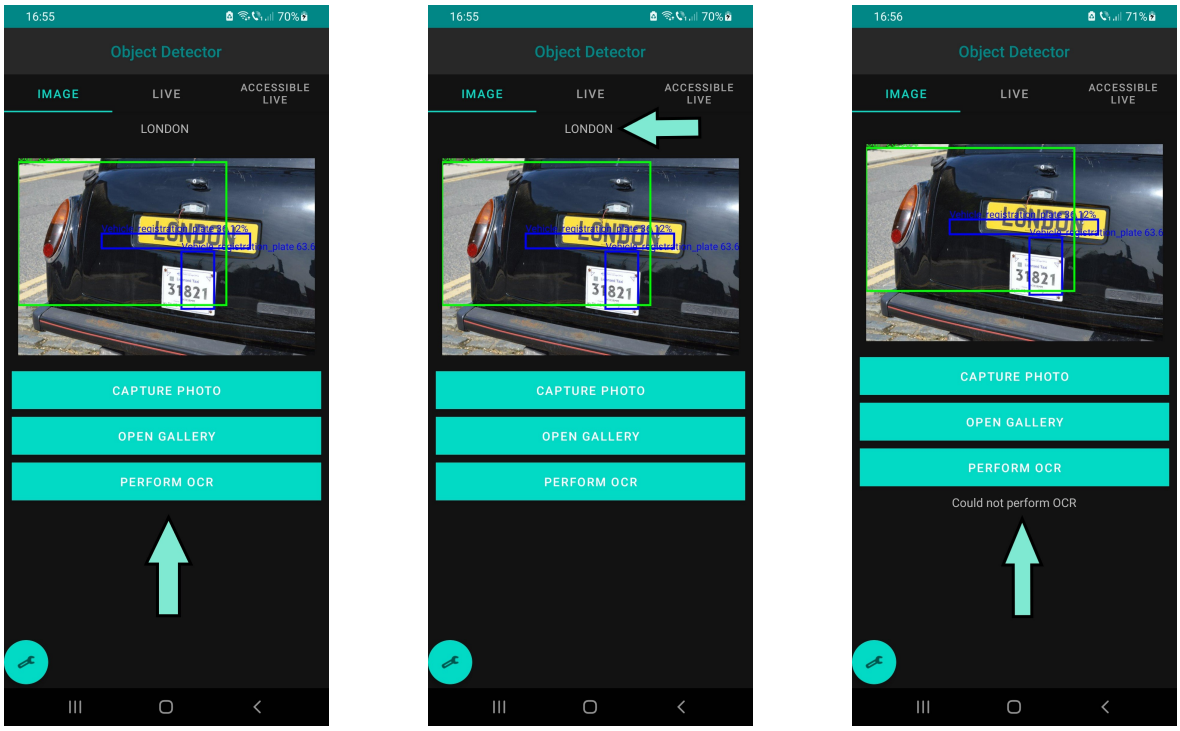
(a) Capture photo button (b) Open gallery button (c) Result

Figure A.5: Object detection on static image

Perform OCR on static images

In order to perform OCR A.6, an image must be already loaded by using the previous functionality. Then the perform OCR button A.6a must be pressed. If the

OCR API request is successful, then the result is shown as in A.6b, otherwise, an error message is shown as in A.6c.



(a) Perform OCR button (b) OCR results position (c) OCR error position

Figure A.6: Perform OCR on an image

Perform object detection on live images

In order to use this functionality, the live screen must be selected A.1b, as explained earlier. The live feed from the camera fills the screen, therefore the user needs only to move the mobile device and the predictions will be drawn on the screen automatically.

Perform accessible object detection on live images

In order to use this functionality, the accessible live screen must be selected A.1c, as explained earlier. The live feed from the camera is not shown on the screen, but the predictions are automatically converted to sound and played. Also, if any text is detected using the OCR API, it is also converted to sound and played automatically. Similar to the simple live object detection, the user must move the mobile device in order to perform object detection on the live feed from the camera.

B. Results visualization

In general, green bounding boxes represent cars, red bounding boxes represent busses, and blue bounding boxes represent registration plates.

Firstly we present the result on some images from Cluj-Napoca.

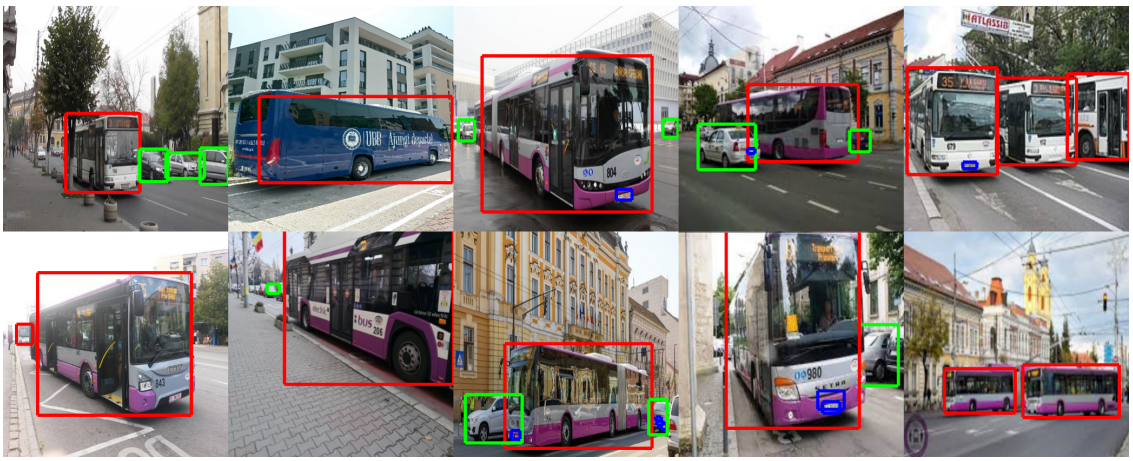
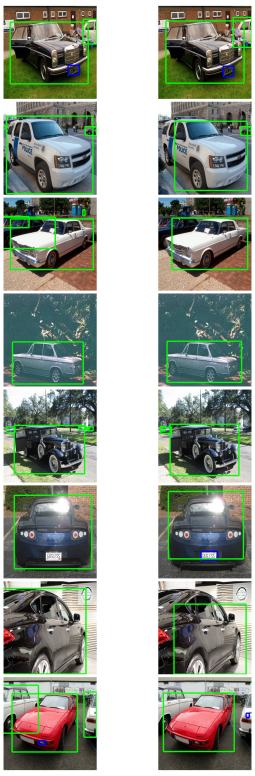


Figure B.1: Images from Cluj-Napoca

For the next sets of images, for each pair of columns, the column on the left represents the ground truth and the right column represents the prediction. All of the images are from the test set, meaning that the model has not seen them during training. It is interesting to note that for some images, the model finds objects that were not correctly annotated in the original image. Another pattern is that sometimes the model interprets the front of a bus as a car.

The results for every batch are available at *model v42 fine tuned* results.



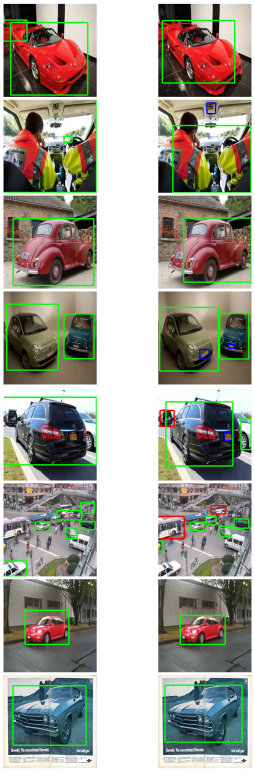
(a) Batch 3

(b) Batch 4

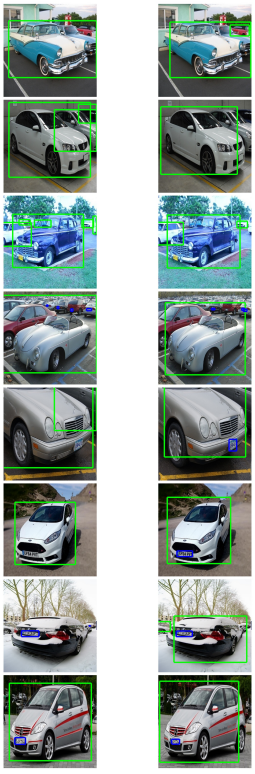
(c) Batch 6

(d) Batch 10

(e) Batch 11



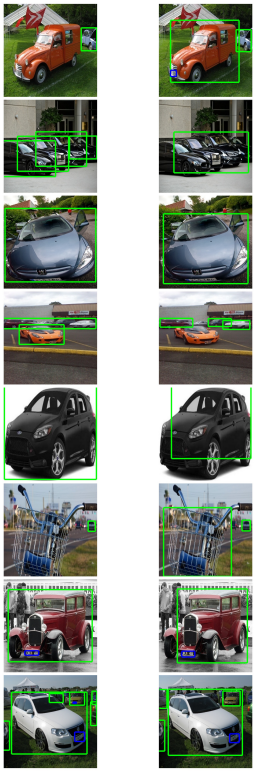
(f) Batch 12



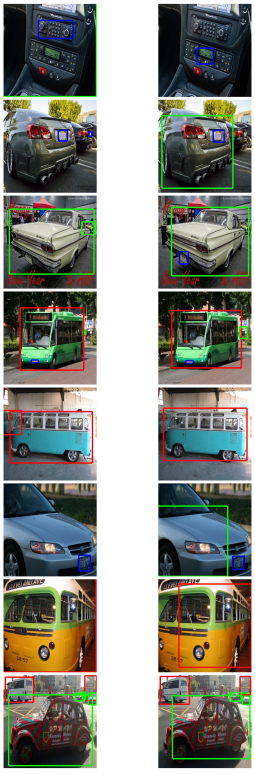
(a) Batch 15

(b) Batch 17

(c) Batch 18



(d) Batch 20

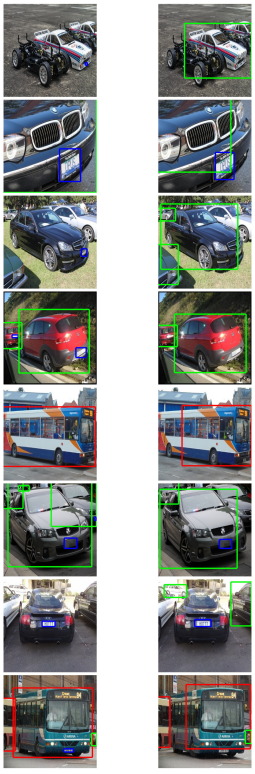


(e) Batch 24

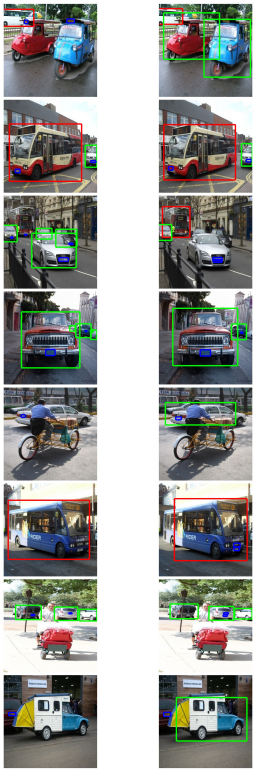
(f) Batch 25

(a) Batch 28

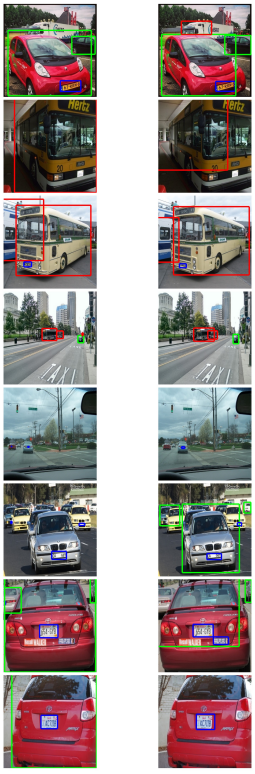
(b) Batch 29



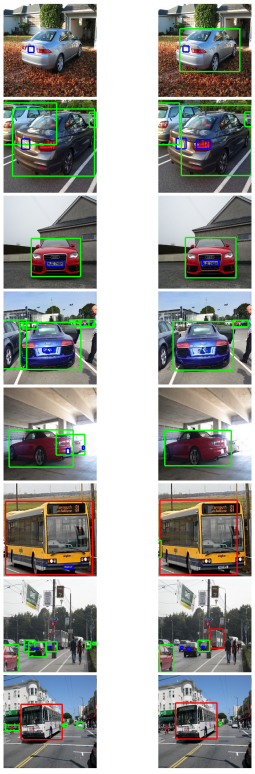
(c) Batch 30



(d) Batch 31



(e) Batch 32

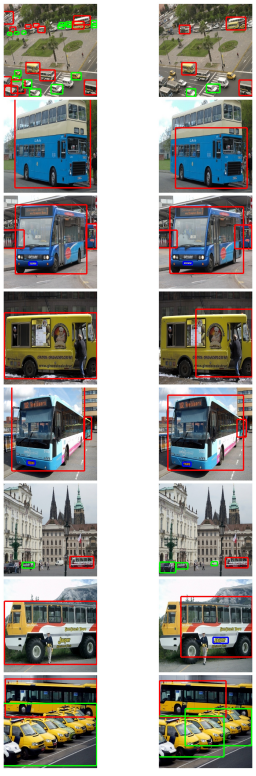


(f) Batch 36

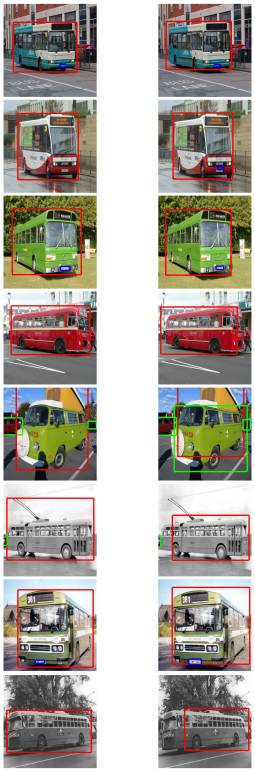
(a) Batch 38

(b) Batch 43

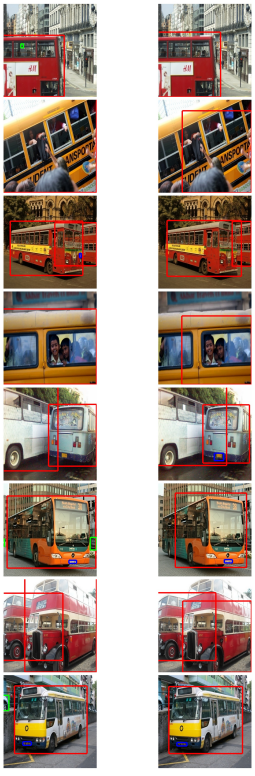
(c) Batch 55



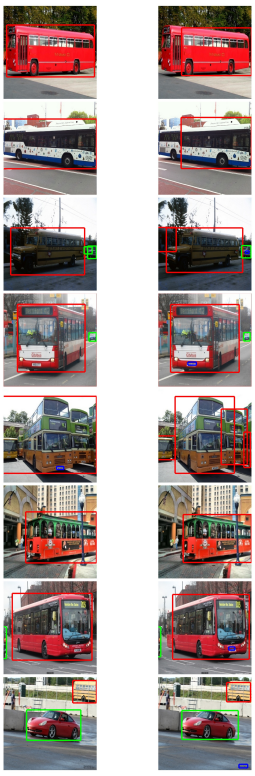
(d) Batch 61



(e) Batch 62



(f) Batch 63

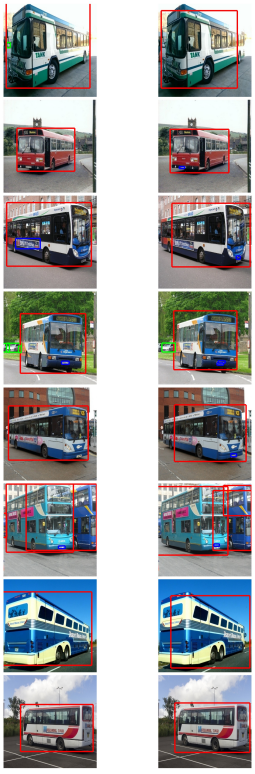


(a) Batch 64

(b) Batch 65

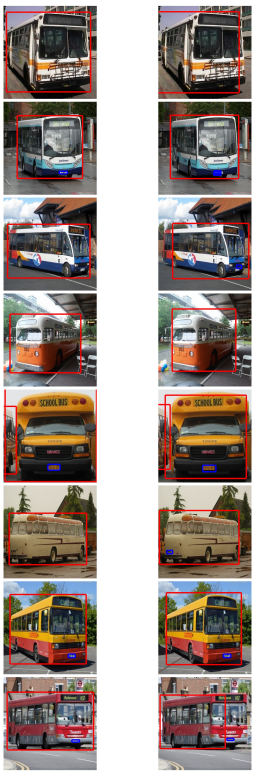
(c) Batch 66

(d) Batch 67



(e) Batch 68

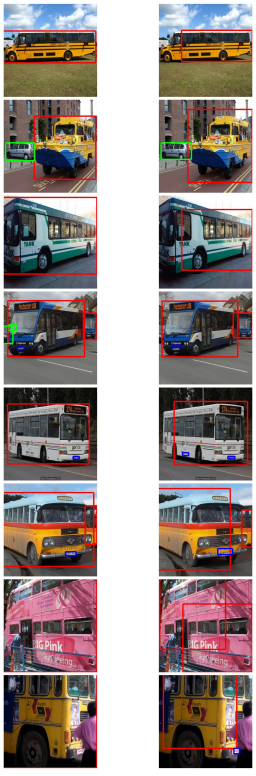
(f) Batch 69



(a) Batch 72

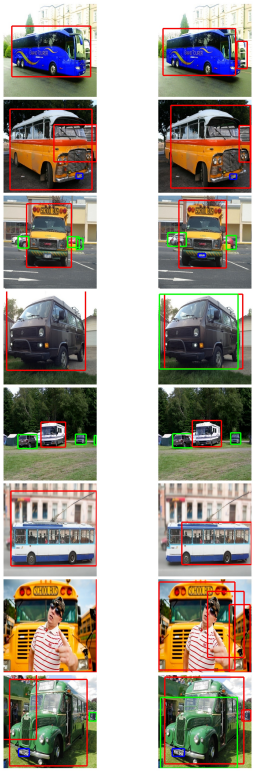
(b) Batch 73

(c) Batch 76



(d) Batch 77

(e) Batch 78



(f) Batch 79