Abstract

In some institutions, office buildings, or government facilities the flow of incoming and outgoing traffic of people and cars needs to be monitored and recorded for security purposes as well as practicality and automation of entry pass for vehicles. In this project, a data set containing 1000 car images is collected, labeled, and then split into training set and testing set. The size of this data set would allow for a transfer learning approach and fine tuning of models. The next step is to train various models belonging to the Yolo and Faster RCNN families to perform the task of plate detection only. Once the models are trained and optimized they were used to crop images of plates from the original images of cars. These cropped images were used to train models for the task of digit recognition similar to those trained for plate detection. The training process was repeated for different structures and parameters of the models in order to obtain the best performance possible. Evaluating these models relies on the use of the mAP (mean average precision) which is used in the original papers of YOLO and Faster-RCNN. The evaluation of the final model (plate detection + digit recognition) will rely on the accuracy of performing the identification of the license plate numbers. This project aims to provide Algerian academics and software developers with a benchmark data set for further research on the topic and for evaluation of future models. It would also provide users of the application with a reliable and practical security tool.

Dedication

This work is wholeheartedly dedicated to my beloved parents Mokhtar and Fouzia, and my brothers Ahmed and Seddik and my sister Meriem for the support on all levels, who have been my source of inspiration; whose affection, love, encouragement and prays of day and night make me able to get such success and honor. I also extend my gratitude and congratulations to my partner Malik for the good work that he did despite facing hard personal challenges as well as my friends, relatives, mentor, teachers, classmates who have been all along.

Mehieddine Boudissa

Dedication

I dedicate my dissertation work to my family and many friends. A special feeling of gratitude to my loving parents, Mohammed and Ouiza whose words of encouragement and push for tenacity ring in my ears. My sisters Dihia, and Lydia have never left my side and are very special. I also dedicate this dissertation to my many friends and classmates who have supported me throughout the process. I will always appreciate all they have done, especially Djamel Eddine Zidane and AbdelMalek Bouaoune who reminded me, all along, with the importance of our religion and kept motivating and pushing me to read Quran and never neglect it whatever how desperate the situation may seem. I dedicate this work and give special thanks to my partner Mehieddine for being there for me throughout my struggle and kept faith with me.

Malik KISSOUM

Acknowledgment

In the name of Allah, the Most Beneficent and the Most Merciful, all gratitude goes to Him before and after. For the strength and the preservation from all sickness. Secondly, we would like to express our special thanks of gratitude to our teacher and supervisor Pr. Khouas who gave us the opportunity to fulfill this project and for his guidance and professionalism all along. We thank Dr. Andrew NG from coursera as well, who provided insights and expertise that greatly assisted the development of the project through his excellent courses and occasional tweets.

Contents

List of Tables

List of Figures

List of Abbreviations

MLP Multi Layer Perceptron

CNN Convolutional neural networks

BCE Binary cross-entropy

MLE Maximum likelihood estimate

GD Gradient descent

SGD stochastic gradient descent

Adam Adaptive moment estimate

RGB Red Blue Green

ILSVRC ImageNet Large Scale Visual Recognition Challenge

GPU Graphical processing unit

CPU Central processing unit

YOLO You Only Look Once

RCNN Region Convolutional Neural Networks

IoU Intersection over Union

PASCAL Pattern Analysis, Statistical modeling and Computational Learning

VOC Visual Object Classes

RPN Region Proposal Network

COCO Common Objects in Context

ROI Region of Interest

ALPR Automatic License Plate Recognition

CUDA Compute Unified Device Architecture

FPS Frames Per Second

FC Fully Connected

mAP mean Average Precision

BFLOP Billion Floating Operations

DSC Depth-wise Separable Convolution

Introduction

The world nowadays is moving towards automation at an unprecedented rate. Almost every month, a new break-through is made in the path towards creating smart computer systems with the ability to learn and make decisions of their own. This progress does not only manifest itself in theoretical work and research papers, but also in real world applications such as the latest voice recognition software found in Amazons' Alexa, or the computer vision systems used by Teslas' autonomous cars. Most experts and speculators predict that soon enough, all of the aspects of ordinary life will be dependent upon the use of these artificially intelligent machines. Some are optimistic and consider this as a practical solution for a wide range of problems in various fields such as medicine, transportation, telecommunication, and even politics and economics. Others express fears that this technology may produce more problems that it would solve. This work is an attempt to bring a contribution to the field, and specifically the topic of deep convolutional neural networks, by developing a real world Automatic License Plate Recognition (ALPR) system to identify Algerian license plate numbers using image inputs of the cars. ALPR systems help identify vehicle license plates in an efficient manner without the need of major human resources. Recently, ALPR systems has become more and more important. It can be used by government agencies to find cars that are involved in crime, look up if annual fees are paid or identify persons who violate the traffic rules. Many countries (e.g. U.S., Japan, Germany, Italy, U.K, France, ...) have successfully applied ALPR in their traffic management. Several private operators are also benefiting from ALPR systems.

Recognition tasks are extremely simple for humans, whereas for computers, it rather a very tedious work since all what a computer can understand are numbers. Recent advancements in computer vision has given computers the ability to extract semantically meaningful information from images. In this project, we take advantage of this ability to develop an ALPR system specific to Algerian cars. The system is mainly split into three major stages: data collection and labeling, license plate detection and finally character (digit) detection. The two last stages uses various deep learning techniques and are further elaborated in chapter three and four along side with the first stage. Traditional computer vision techniques employ features chosen by humans to represent the underlying features of the image. These techniques require sophisticated human-designed models to translate raw input pixels into useful recognition responses. Using a deep learning approach, these underlying human-engineered features are automatically selected by the algorithm. The techniques used throughout the project are relatively outdated considering the high rate of

innovation in deep learning. Both works showed practical state-of-the-art results in the tasks of object detection. This project also includes the creation of a labeled data set of Algerian license plates, which is the first one ever of this kind. This data set will allow for other students or researchers to conduct similar works and will be used as a benchmark to track advancements in Algerian ALPR systems.

Chapter 1

Feed-Forward neural networks

To understand the technique used in this report, it is necessary to understand basic neural networks functioning. Let us begin with a classification example. Classification is the process of organizing data into groups or categories. practical examples are spam filters that assign a given email to the "spam" of "non-spam" class, or assigning a diagnosis to a a given patient based on observed characteristics (sex, blood pressure, presence or absence of certain symptoms, ...). The observed characteristics are called **features**, and the classes or categories are referred to as **labels**. The collection of the features and corresponding labels is called **training set**. A sample from the training set is called **an example**. Given a scenario with a training set of labeled data $(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ denotes the training example composed of multiple features, say $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$, and $\mathbf{y}$ the corresponding label. A 'perceptron' is a basic computational function that can be used to perform classification. Perceptrons are the building blocks of neural networks, and the best way to get familiar with them is with an example. Assume that at at some university's admission office the students are evaluated with two pieces of information, the results of an entrance test, and their grades during their previous school years. Observe a 2D plot of some sample students' information, see fig. 1.1. Where The abscissa represents the grades at the entrance test, and the ordinate represents their previous average grade. The blue dots represent the students who got accepted and the red dots represent those who got rejected. Our job is to develop a model that can separate the two groups and predict which students will be accepted and which ones will be rejected.

The two groups of points on the figure can be easily separated with a line, where most students above the line get accepted and most students under the line get rejected, see fig. 1.2. Therefore, this line can be our model for accepting and rejecting students.
The model makes a couple of mistakes since there are a few blue points that are under the line and few over the line, but they are considered as noise (they might have been accepted or rejected by mistake) and add no new information to our model. The problem is how to find the best line that does the separation.

We start by labeling the axis $\mathbf{x} = \{x_1, x_2\}$, where x_1 is TEST axis and x_2 is GRADES

Figure 1.1: *Test vs grades [1].*

Figure 1.2: *Boundary line separating accepted students represented with blue points and rejected students represented with red points [1].*

axis. The boundary line separating the students has a linear equation specifically: $2x_1 + x_2 - 18 = 0$. Passing the grades in the equation gives rise to a score, if the score is positive (the student gets plotted above the line), the student gets accepted. On the other hand, if the score is negative (the student is plotted under the line) the student gets rejected. This is called a prediction.
In a more general case, our boundary will be an equation of the following form:

$$w_1 x_1 + w_2 x_2 + b = 0.$$

Abbreviating this equation into vector notation:

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{1.1}$$

Where $\mathbf{w} = \{w_1, w_2\}$. We refer to $\mathbf{x}$ as the input, $\mathbf{w}$ as the weights and b as the bias. Here $\mathbf{y} = \{0, 1\}$ is the label, where 0 indicates the student being rejected whereas 1 indicates the student being accepted. Finally, our prediction is going to be called $\hat{y}$ and it will be what the algorithm predicts what the label will be, namely:

$$\hat{y} = \begin{cases} 1, & w \cdot x + b \geq 0 \\ 0, & w \cdot x + b < 0 \end{cases} \tag{1.2}$$

The goal of the algorithm is to have $\hat{y}$ resembling to y as closely as possible. Reorganizing the equations in a graph and generalizing for n features, gives rise to fig. 1.3. The bias is considered a dummy input with value 1 to the Perceptron with weight w.

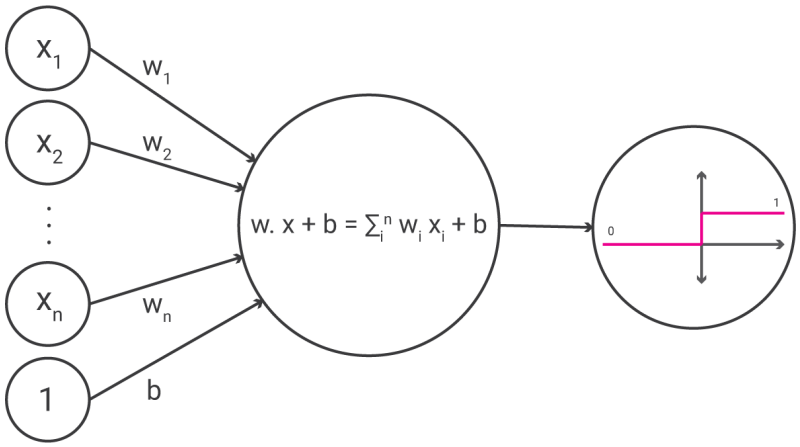


Figure 1.3: *Perceptron graph representation.*

1.1 Cost function

In order to estimate the accuracy of the algorithm, or otherwise stated, determine how well a certain prediction given by the algorithm is, we may establish a cost function, which

measures the error that the algorithm makes on some prediction (cost function is often referred to as error function or loss function). There are more than one choice for such a function. Equation (1.3) can be used, it is called "The Mean Squared Error".

$$L(w,b) = \frac{1}{2}\sum_{i=1}^{n} ||y_i - \hat{y}_i||^2. \tag{1.3}$$

This function, becomes large when our algorithm approximates y badly, and small when the approximation is accurate. Additionally, notice that if we set $L_x = \frac{1}{2}(y-\hat{y})^2$ we have that:

$$L(w,b) = \sum_{i=1}^{n} L_{x_i}. \tag{1.4}$$

This property will be important in the algorithm described in section 1.4.

Another way of defining a cost function is using the "The Maximum Likelihood Estimation" technique, since the sigmoid function deals with probabilities. We take the joint probability of an entire training set, assuming the training examples being independent events:

$$L(w,b) = p(y^{(1)}, y^{(2)}, \ldots, y^{(n)} | x^{(1)}, \ldots, x^{(n)}) = \prod_{i=1}^{n} p(y^{(i)} | x^{(i)}) \tag{1.5}$$

where $x^{(i)}$, $y^{(i)}$ represent the i^{th} training example and label respectively. Thus by maximizing the joint probability, or respectively minimizing the $(-\log)$ of the likelihood, we can get an estimate of the parameters *w and b*. This is referred to as maximum likelihood estimate (MLE). Now, in our worked example, the neural network can be treated as a random variable having a Bernoulli distribution, therefore eq. (1.5) can be rewritten as follows [21]:

$$L(w,b) = -\sum_{i=1}^{n} y_i \log(\hat{y}) + (1-y)\log(1-\hat{y}). \tag{1.6}$$

Equation (1.6) is usually the cost function used for Bernoulli distributed labeled data. It is often referred to as **binary cross-entropy** or BCE for short. For multi-class classification (predicting multiple classes, say k classes), a similar idea can be used considering a multinoulli distribution on the data set where $p(y|\mathbf{x}) = \prod_{i=1}^{k} p_i^{[y=i]}$, where $[y=i]$ evaluates to 1 if $x=i$, 0 otherwise. This leads to following cost function using MLE

$$L(w,b) = -\sum_{j=1}^{k}\sum_{i=1}^{n} y_{i,j} \log(\hat{y}_{i,j}). \tag{1.7}$$

1.2 Gradient descent

In order to minimize the cost function we rely on optimization algorithms from numerical methods as it is unpractical to minimize manually. The technique used in deep learning is the gradient descent. The gradient of a differentiable function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ at a point $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is a vector in $\mathbb{R}^n$ of the form [20]:

$$\nabla f(x) = (\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_n}(x)) \tag{1.8}$$

It is a well known result that, given a point $x \in \mathbb{R}^n$, the gradient at that point indicates the direction of steepest ascent. Given that f is differentiable at x, the vector $-\nabla f(x)$ indicates the direction of steepest descent of the function f at the point x. In order to obtain the minimum value of the function, the gradient descent strategy tells us to start at a given $x_0 \in \mathbb{R}^n$, calculate the value of $\nabla f(x_0)$, and then proceed to calculate a new point $x_1 = x_0 - \alpha \nabla f(x_0)$, where $\alpha > 0$ is called the **learning rate**. We then repeat this process, creating a sequence $\{x_i\}$ defined by our initial choice of x_0, the learning rate α, and the rule: $x_{i+1} = x_i - \alpha \nabla f(x_0)$. This sequence continues until we approach a region close to our desired minimum. The method of gradient descent when taken continuously over infinitesimally small increments (that is, taking the limit $\alpha \to 0$) usually converges to a local minimum. However, depending on the location of the initial x_0, the local minimum achieved may not be the global minimum of the function. Furthermore, since when carrying out calculations on an unknown function we must take discrete steps (which vary in length depending on the learning rate), we are not even guaranteed a local minimum but rather may oscillate close to one, or even 'jump' past it altogether if the learning rate is too big. Still, even with these possible complications, gradient descent is a surprisingly successful method for many real life applications and is the most standard method of training for feed-forward neural networks and many other machine learning algorithms. Given that our cost function indicates how poorly our neural network approximates a given function, by calculating the gradient of the cost function with respect to the weights and biases of the network and adjusting these parameters in the direction opposite to the gradient, we will decrease our error and therefore lead us closer to an adequate network (in most cases) [20].

1.2.1 Gradient calculation

Before applying the gradient descent technique, we can clearly see that the an output of 0 or 1 is problematic since the derivatives would be 0. Therefore, the gradient descent technique will not work. To remedy this, following the MLE, a Bernoulli distribution has been defined on y, therefore the neural net needs to predict $\hat{y} = p(y = 1|x) = \sigma(x)$. For this number to be a valid probability, it must lie in the interval $[0, 1]$.

A good approach would ensure the existence of a strong gradient whenever the model has the wrong answer. For consistency with the perceptron's decision rule (eq. (1.2)), a very positive linear combination of the input x has to have a probability close to 1 and vise versa (see fig. 1.4), otherwise [21]:

$$\lim_{x\to+\infty}\sigma(x)=1, \qquad \lim_{x\to-\infty}\sigma(x)=0. \tag{1.9}$$

This approach is based on the sigmoid function:

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

This function is suitable for the problem at hand, namely binary classification. However, depending on the output $\hat{y}$ other functions might be used. For example if a neural network is used to predict continuous non-bounded function that takes values in the interval $]-\infty,+\infty[$, then a more clever choice is the linear activation function, which is defined as: $f(x)=x$. Another example is the multi-class classification, where a multinoulli distribution is defined over the training data. The function used is the softmax defined as

$$softmax(x_i)=\frac{e^{x_i}}{\sum_j e^{x_j}}$$

Where x_i represents a training example from class i [21]. Here the output consists of j outputs rather than a single one. See fig. 1.5 to illustrate the output layer.

Step function (perceptron's decision rule) Sigmoid function

Figure 1.4: *sigmoid vs step function. The two plots clearly show cast the continuity of the sigmoid.*

Now, let us apply the gradient descent technique to our network. Our goal is to calculate the gradient of L at a point $x=(x_1,\ldots,x_n)$ given by the partial derivatives, see eq. (1.8). In addition, the property mentioned in eq. (1.4) now become important: we are only going to calculate the value of ∇L_x for a given labeled data point and then add the values of the gradient together, see below.

$$\nabla L(w,b)=\nabla(\sum_{i=1}^{n}L_x)=\sum_{i=1}^{n}\nabla L_{x_i}. \tag{1.10}$$

The error produced by each point is simply: $L_x = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$. In order to calculate the derivative of this error with respect to the weights, we will first calculate $\frac{\partial}{\partial w_j}\hat{y}$, where $\hat{y} = \sigma(w \cdot x + b)$ [1].

$$\begin{aligned}
\frac{\partial}{\partial w_j}\hat{y} &= \frac{\partial}{\partial w_j}\sigma(w \cdot x + b) \\
&= \sigma(w \cdot x + b)(1 - \sigma(w \cdot x + b)) \cdot \frac{\partial}{\partial w_j}(w \cdot x + b)^1 \\
&= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(w \cdot x + b) \\
&= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(w_1 x_1 + \ldots + w_j x_j + \ldots w_n x_n + b) \\
&= \hat{y}(1-\hat{y}) \cdot x_j.
\end{aligned}$$

Now we can go ahead and calculate the derivative of the error L at a point x, with respect to the weight w_j.

$$\begin{aligned}
\frac{\partial}{\partial w_j}L_x &= \frac{\partial}{\partial w_j}[-y\log(\hat{y}) - (1-y)\log(1-\hat{y})] \\
&= -y\frac{\partial}{\partial w_j}\log(\hat{y}) - (1-y)\frac{\partial}{\partial w_j}(1-\hat{y}) \\
&= -y\frac{1}{\hat{y}} \cdot \frac{\partial}{\partial w_j}\hat{y} - (1-y)\frac{1}{1-\hat{y}} \cdot \frac{\partial}{\partial w_j}(1-\hat{y}) \\
&= -y(1-\hat{y}) \cdot x_j + (1-y)\hat{y} \cdot x_j \\
&= -(y-\hat{y})x_j
\end{aligned}$$

A similar calculation will show that :

$$\frac{\partial}{\partial b}L_x = -(y-\hat{y})$$

Therefore, since the gradient descent step simply consists in subtracting a multiple of the gradient of the error function at every point, then this updates the weights in the following way [1]:

$$w_i' = w_i + \alpha(y-\hat{y})x_i \tag{1.11}$$

$$b' = b + \alpha(y-\hat{y}) \tag{1.12}$$

[1]The sigmoid has a nice derivative: $\sigma'(x) = \sigma(x)(1-\sigma(x))$

1.3 Neural network architecture

In our work example, the target function was a simple linear function. However, in real world situations the input data is much more complex and often cannot be separated with a line. That is where neural nets shine. Neural networks also referred to as **feed-forward** neural nets or **multilayer perceptron** (MLPs) are as the name indicates stacks of perceptrons, where each **unit** receives the input x, calculates the inner product with a set of weights and apply a non-linearity to the result, then these results are fed to a next layer of units that does the same calculations and so on. The overall length of the chain gives the **depth** of the model. The final layer of such a network is called the **output layer**, whereas the intermediate layer are referred to as **hidden layers**. The goal of the feed-forward network is to approximate some function f^*. The training example specify directly what the output layer must do at each point x; it must produce a value that is close to y. Therefore, the function computed after the linear combination is important. This function and the functions used in the hidden layers are referred to as **activation functions**. For example for a classifier, the function maps an input x to a category y, a natural choice of activation is the sigmoid; whereas in a regression problem, where the output is continuous non-bounded that takes values in the interval $]-\infty,+\infty[$, a more clever choice is the linear activation function, which is defined as: $f(x) = x$ [21]. The figure below depicts the architecture described above.

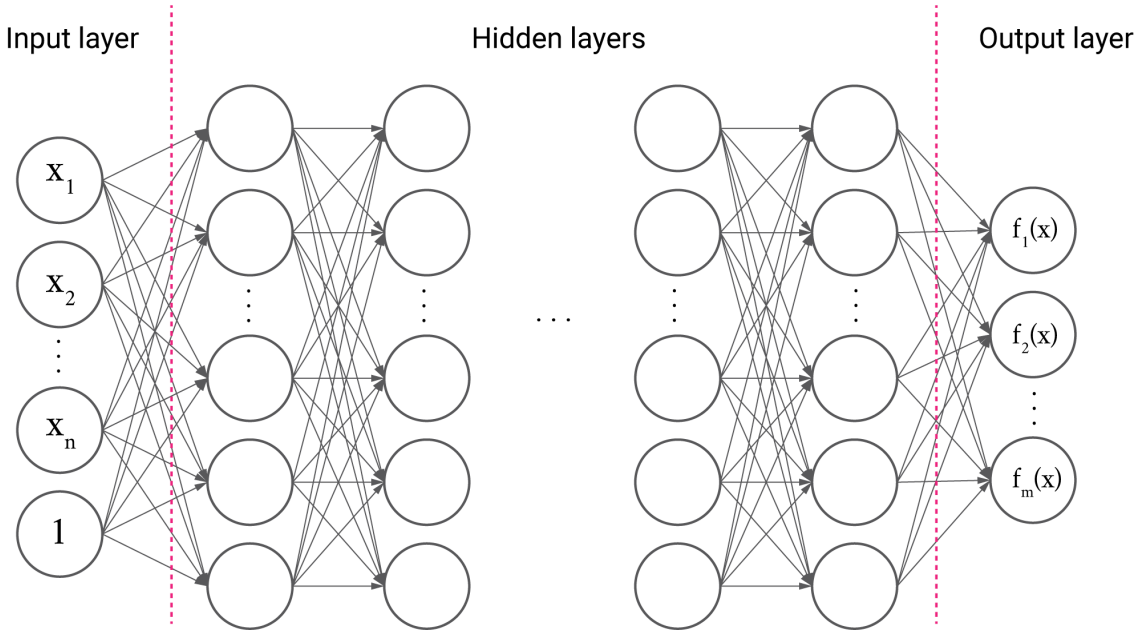


Figure 1.5: *A visual representation of a feed-forward network which approximates some function f. In this approach, the network is shown as a directed weighted graph [20].*

For notation, set $w^n_{a,b} \in \mathbb{R}$ as the weight between the a^{th} unit in the $(n-1)^{th}$ layer with k units to the b^{th} unit in the n^{th} layer with j units.

$$w_n = \begin{pmatrix} w^n_{1,1} & \cdots & w^n_{1,k} \\ \vdots & \ddots & \vdots \\ w^n_{j,1} & \cdots & w^n_{j,k} \end{pmatrix} \tag{1.13}$$

The bias can be added as a dummy unit with input $x_{n+1} = 1$, which is a constant $b \in \mathbb{R}^j$. In order to calculate the output a_n of the n^{th} layer, we use the formula:

$$a_n = \sigma(w_n \cdot a_{n-1} + b_n).$$

In the above equation, the activation function σ is applied element-wise to each element of the resulting vector. As the computations are carried out along the network's layers, the final function f calculated by a network of depth N is [20]

$$f(x) = \sigma(w_N \sigma(\ldots \sigma(w_2 \sigma(w_1 \cdot x + b_1) + b_2)\ldots) + b_N)$$

1.4 Back-propagation

In order to train a neural network, the same techniques are used as in section 1.2.1. First we define a cost function (which is the same as in the perceptron algorithm eq. (1.6),but with a much more complex $\hat{y}$), we calculate the feed-forward pass (we calculate the output $\hat{y}$), and then calculate the the gradient of the cost function L with respect to every single weight and bias in the network, we get the following gradient vector $\nabla L = (\ldots, \frac{\partial}{\partial w^l_{i,j}} L, \ldots)$. Then applying the gradient step using eq. (1.11):

$$w'^l_{i,j} = w^l_{i,j} - \alpha \frac{\partial}{\partial w^l_{i,j}} L$$
$$b'^l_j = b_j - \alpha \frac{\partial}{\partial b^l_j} L$$

The big challenge of applying gradient descent to neural networks is calculating these partial derivatives. This is where back-propagation comes in. This algorithm first tells us how to calculate these values for the last layer of connections, and with these results then inductively goes "backwards" through the network, calculating the partial derivatives of each layer until it reaches the first layer of the network. Hence the name "back-propagation" [20].

For the purpose of this section it is useful to consider the values of each layer before the activation function step. Consider:

$$z_j^l = \sum_k w_{j,k}^l a_k^{l-1} + b_j^l \qquad \text{so that} \qquad a_j^l = \sigma(z_j^l).$$

Additionally, we define the following:

$$\delta_j^l = \frac{\partial}{\partial z_j^l} L \tag{1.14}$$

This value will be useful for propagating the algorithm backwards through the network and directly related to $\frac{\partial}{\partial w_{i,j}^l} L$ and $\frac{\partial}{\partial b_j^l} L$ by the chain rule. since we have:

$$\frac{\partial L}{\partial w_{i,j}^l} = \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{i,j}^l} = \delta_j^l a_i^{l-1} \tag{1.15}$$

$$\frac{\partial L}{\partial b_j^l} = \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l. \tag{1.16}$$

The value a_j^{l-1} has already been calculated through the forward pass. The only remaining term to calculate is δ_j^l and we obtain our gradient. Our first step is calculating this value for the last layer of the network, that is, δ_j^N for a network with N layers. Since $a_j^N = \sigma(z_j^N)$, again using the chain rule:

$$\delta_j^N = \frac{\partial L}{\partial a_j^N} \frac{\partial a_j^N}{\partial z_j^N} = \frac{\partial L}{\partial a_j^N} \sigma'(z_j^N) \tag{1.17}$$

which can be easily calculated by a computer if we know how to calculate σ' (which should be true for any practical activation function). Now we will only need to "propagate" this backwards in the network in order to obtain δ_j^{N-1}. In order to do so, we apply the chain rule once again [20]:

$$\begin{aligned}
\delta_j^{N-1} &= \frac{\partial L}{\partial z_j^{N-1}} \\
&= \sum_i^k \frac{\partial L}{\partial z_i^N} \frac{\partial z_i^N}{\partial z_j^{N-1}} \\
&= \sum_i^k \delta_i^N \frac{\partial z_i^N}{\partial z_j^{N-1}}
\end{aligned}$$

If we focus on the term $\frac{\partial z_i^N}{\partial z_j^{N-1}}$, we find that:

$$\begin{aligned}\frac{\partial z_i^N}{\partial z_j^{N-1}} &= \frac{\partial(\sum_k w_{i,k}^N a_k^{N-1} + b_i^N)}{\partial z_j^{N-1}} \\ &= \frac{\partial(w_{i,j}^N \sigma(z_j^{N-1}))}{\partial z_j^{N-1}} \\ &= w_{i,j}^N \sigma'(z_j^{N-1})\end{aligned}$$

which, again, can be easily calculated by a computer given the network. Therefore:

$$\delta_j^{N-1} = \sum_i^k \delta_i^L w_{i,j}^N \sigma'(z_j^{N-1}). \tag{1.18}$$

This formula tells us how to calculate any δ_j^l in the network, assuming we know δ^{l+1}. We finally have a way to calculate all the δ_j^l 's, given that we know what the values of δ_j^{l+1} are. Thus, by propagating this method backwards through the layers of the network, we are able to find all our desired partial derivatives, and can therefore calculate the value of ∇L as a function of the weights and biases of the network and execute the method of gradient descent [20].

1.5 Problems related to neural nets

Neural networks are extremely powerful function approximators, but during the design of the architecture care should be taken since there are many parameter one can tune (depth, number of units in each layer, ...). Therefore, a complex design namely high number of units in each layer and a deep network can lead to **overfitting**. Over-fitting is the case where the overall cost is really small (The network is doing very well on the training set) but the generalization of the model to unseen data is poor and unreliable. There are many solutions proposed to break this effect such as *dropout* which consists of randomly zeroing the output of some units in each layer to force the algorithm to take different routes through the network. This has the effect of training smaller portions of the network, and thus smaller functions with reduced complexity are learned. It has the effect of reducing the high variance of the overall neural net. This is referred to as **regularization**.
Another famous problem neural nets suffer from is **local minimum** problem, The error surface of a complex network is full of hills and valleys. Because of the gradient descent, the network can get trapped in a local minimum when there is a much deeper minimum nearby. A suggested solution is to increase the number of hidden units. This technique works because of the higher dimensionality of the error space, making the chance to get trapped smaller [10].

Another issue in deep neural nets is the **vanishing gradients** problem. As we learned from back-propagation, each of the neural network's weights receive an update proportional to the partial derivative of the error function with respect to the current weight in each training iteration. The problem is that in some cases, the gradient will be vanishingly small, eventually preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. As the network trains, the weights can be adjusted to very large values. The total input of a hidden unit or output unit can therefore reach very high (either positive or negative) values, and because of the sigmoid activation function the unit will have an activation very close to zero or very close to one [10]. And since back-propagation computes gradients using the chain rule, this has the effect of multiplying N of these small numbers to compute gradients of the "front" layers in an N-layer network, meaning that the gradient decreases exponentially with N while the front layers train very slowly.
To remedy this problem other activation functions might be used in the hidden layers. The behavior of the hidden layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f. Therefore the choice of the activation function in those layers is irrelevant, which makes the use of other activation possible. Many functions have been proposed to escape the trap of vanishing gradients, namely the ReLU function is of popularity in deep learning. The ReLU stands for rectified linear unit defined as $ReLU(x) = \max(0, x)$.

1.6 Batch and stochastic gradient descent

Batch gradient descent is just another name for the gradient descent (GD) discussed so far. It involves calculations over the full training set to take a single step as a result of which it is very slow on very large training data due to the size of the weight matrices that take up large memory portions. Thus it become very computationally expensive to do batch GD. One can take advantage of the property mentioned in section 1.1, eq. (1.4). Therefore, instead of going through the entire data-set, at each iteration, we select a few elements from the training set, commonly selected by randomly sampling from all the available labeled data, calculate the gradient, update the network's weights and repeat the process until the network arrives at satisfactory results. The gradients computations are faster as there is much fewer data to manipulate in a single time. This technique is referred to as **stochastic** gradient descent (SGD). One downside though of SGD is, once it reaches close to the minimum value it does not settle down, instead bounces around which gives us a good value for model parameters but not optimal which can be solved by reducing the learning rate at each step which can reduce the bouncing and SGD might settle down at

global minimum after some time [10].

1.7 Adam optimizer

Adam is a an optimization algorithm and is an extension to the stochastic gradient descent. It is the most preferred optimizer within the deep learning community because it almost always work faster than SGD. The method computes individual adaptive learning rates for different parameters from estimates of first (mean) and second (variance) moments of the gradients; the name Adam is derived from adaptive moment estimation [15]. The algorithm updates exponential moving averages of the gradient (m_t) and the squared gradient (v_t) where two parameters $\beta_1, \beta_2 \in [0,1)$ control the exponential decay rates of these moving averages. The moving averages themselves are estimates of the first moment (the mean m_t) and the second raw moment (the variance v_t) of the gradient [15]. The update rule of the adam optimizer is as follow, first calculate the running average of the weights as follows (eq. (1.19)):

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \frac{\partial L}{\partial w^l_{i,j}} \tag{1.19}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot (\frac{\partial L}{\partial w^l_{i,j}})^2 \tag{1.20}$$

Where m_0 and v_0 are initialized to zero vectors. This leads to moment estimates that are biased towards zero, especially during the initial timesteps. To counteract this bias, both m_t and v_t are divided by $(1-\beta^t)$ (eq. (1.21)) [15]. The square operation in the second equation is actually an element-wise square not the ordinary square operation.

$$\hat{m}_t = m_t/(1-\beta_1^t), \qquad \hat{v}_t = v_t/(1-\beta_2^t) \tag{1.21}$$

Finally, the weights are updated according to eq. (1.22)

$$w_t = w_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \tag{1.22}$$

The same equations apply to the bias term b. The reason why adam is effective is because it is invariant to the scale of the gradients; rescaling the gradients with a factor of c will scale $\hat{m}_t$ with a factor c and $\hat{v}_t$ with a factor of c^2, which cancel out [15]. The other reason is that if the gradients using SGD with a high learning rate oscillates a lot in some dimensions, the adam optimizer has the effect of dumping those oscillations since it is taking the mean of the gradient in that direction, summing up positive and negative numbers making the gradients move faster towards the minimum. Also in eq. (1.22), we can notice that if $\sqrt{\hat{v}_t} + \varepsilon$ is large due to large variation of the gradients, then the term

$\hat{m}_t$ will be divide by a large number making it small and this has the effect of dumping as well the oscillations. See figure fig. 1.6. The ε term is added for numerical stability in case $\hat{v}_t = 0$ [3].

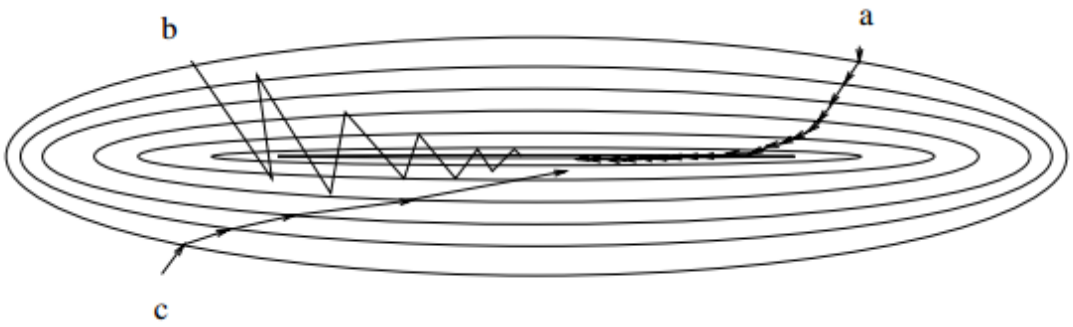


Figure 1.6: *The descent in weight space. The concentric ellipsis represents the counters of the cost function. a) SGD with a small learning rate, b) SGD with a large learning rate, c) Adam with a large learning rate.*

Chapter 2

Convolutional Neural Networks

CNNs, also known as convolutional neural networks, are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels. CNNs have been tremendously successful in practical applications. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. CNNs are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [21].

2.1 Convolution operation

The convolution operation is well known in the engineering terminology, which, in its most general form, is an operation on two functions of a real-valued argument. defined as:

$$s[n] = y[n] * x[n] = \sum_{k=-\infty}^{k=\infty} y[k]x[n-k] \tag{2.1}$$

We are interested in the discrete convolution operation, since data on a computer is presented as discrete values rather than continuous signals. The eq. (2.1) presented above is for discrete time signals.

In convolutional neural network terminology, the first argument to the convolution is often referred to as **the input**, and the second argument as **the kernel**. The output is sometimes referred as the **feature map**. The input is usually a multidimensional array of data (Red Green Blue channel (RGB) images), and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. These multidimensional arrays are referred to as tensors. Finally, we often use convolution over more than one axis

at a time. For example if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

$$S[m,n] = I[m,n] * K[m,n] = \sum_i \sum_j I[i,j]K[m-i,n-j]. \tag{2.2}$$

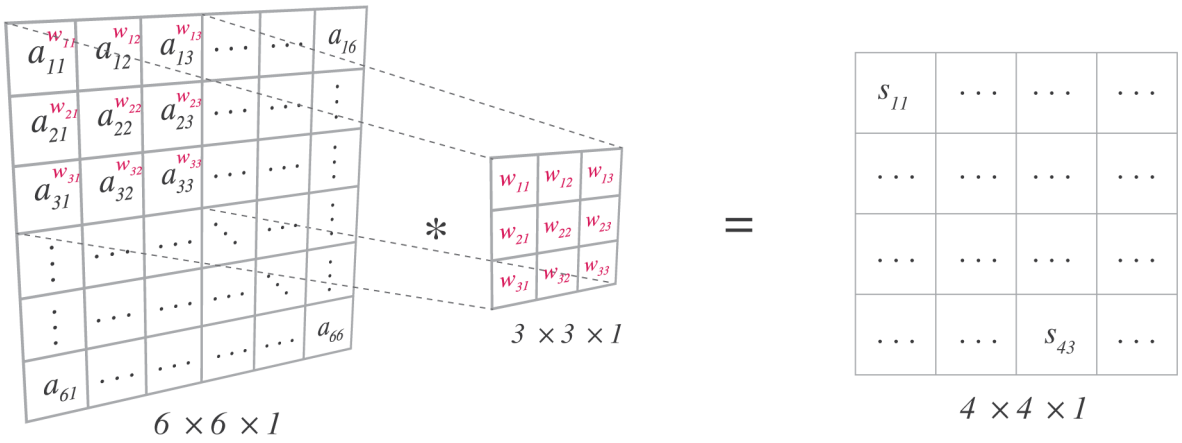
Convolution is commutative, meaning we can equivalently write:

$$S[m,n] = K[m,n] * I[m,n] = \sum_i \sum_j I[m-i,n-j]K[i,j]. \tag{2.3}$$

While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel:

$$S[m,n] = I[m,n] * K[m,n] = \sum_i \sum_j I[m+i,n+j]K[i,j]. \tag{2.4}$$

Many machine learning libraries implement cross-correlation but call it convolution. See fig. 2.1 for an example of convolution applied to a 2D tensor (gray-scale image).



$$s_{11} = a_{11}\,w_{11} + a_{12}\,w_{12} + a_{13}\,w_{13} + a_{21}\,w_{21} + a_{22}\,w_{22} + a_{23}\,w_{23} + a_{31}\,w_{31} + a_{32}\,w_{32} + a_{33}\,w_{33}$$

$$s_{43} = a_{43}\,w_{11} + a_{44}\,w_{12} + a_{45}\,w_{13} + a_{53}\,w_{21} + a_{54}\,w_{22} + a_{54}\,w_{23} + a_{63}\,w_{31} + a_{64}\,w_{32} + a_{65}\,w_{33}$$

Figure 2.1: *An example of 2-D convolution [21].*

2.2 CNN architectures

Convolution leverages three important ideas that can help improve a machine learning system: sparse connectivity, parameter sharing and equivariant representations.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit, see fig. 2.2. CNNs, however, typically have sparse interactions (also referred to as **sparse connectivity** or sparse weights). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters, and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping k several orders of magnitude smaller than m [21]. For graphical demonstrations of sparse connectivity, see fig. 2.3 and fig. 2.4. Rearranging each vector as a matrix, the relationship between the nodes in each layer are more obvious, see fig. 2.4.

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has tied weights, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere fig. 2.2. That is the reason, traditional nets are referred as to Fully connected (FC) networks or Dense networks.

In a CNN, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. In fig. 2.4, each of the color coded image quarters are connected to a single color coded node in the next layer. All of these connections have exactly the same shared weights, see fig. 2.1, the weights w_{11} through w_{33} do not change as the filter slides through the image. This does not affect the runtime of forward propagation —it is still $O(k \times n)$— but it

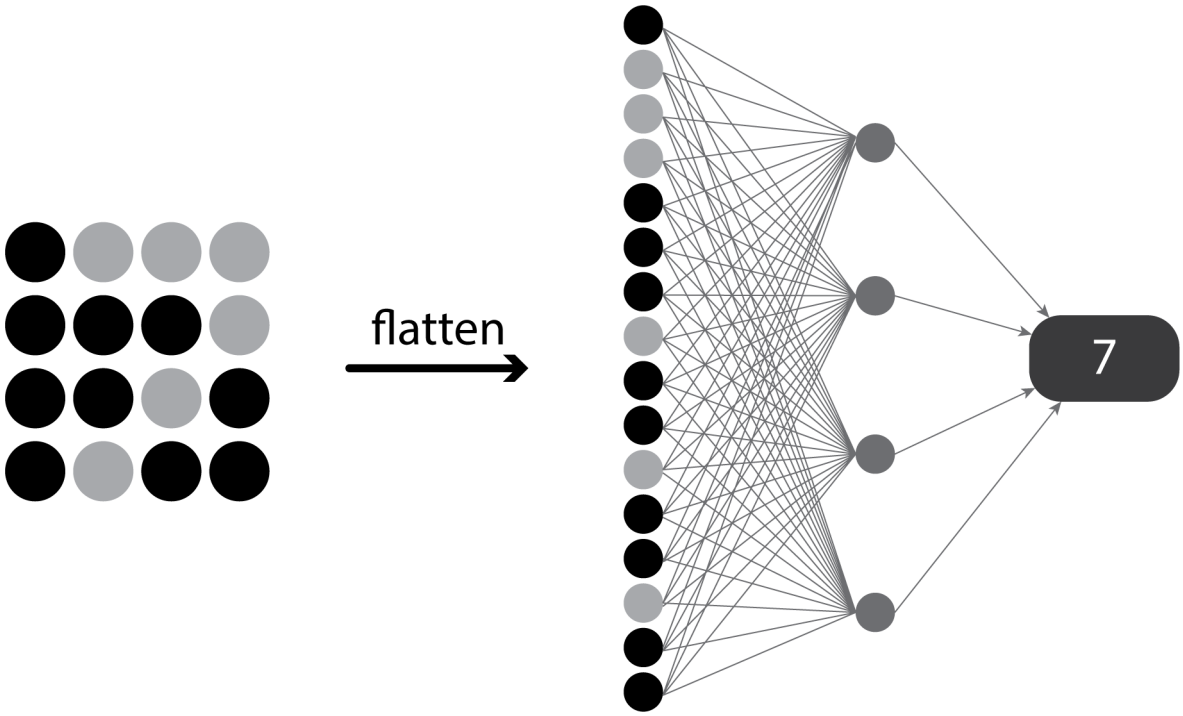


Figure 2.2: *Traditional neural network connections. The last layer has been replaced by a black box for simplicity*

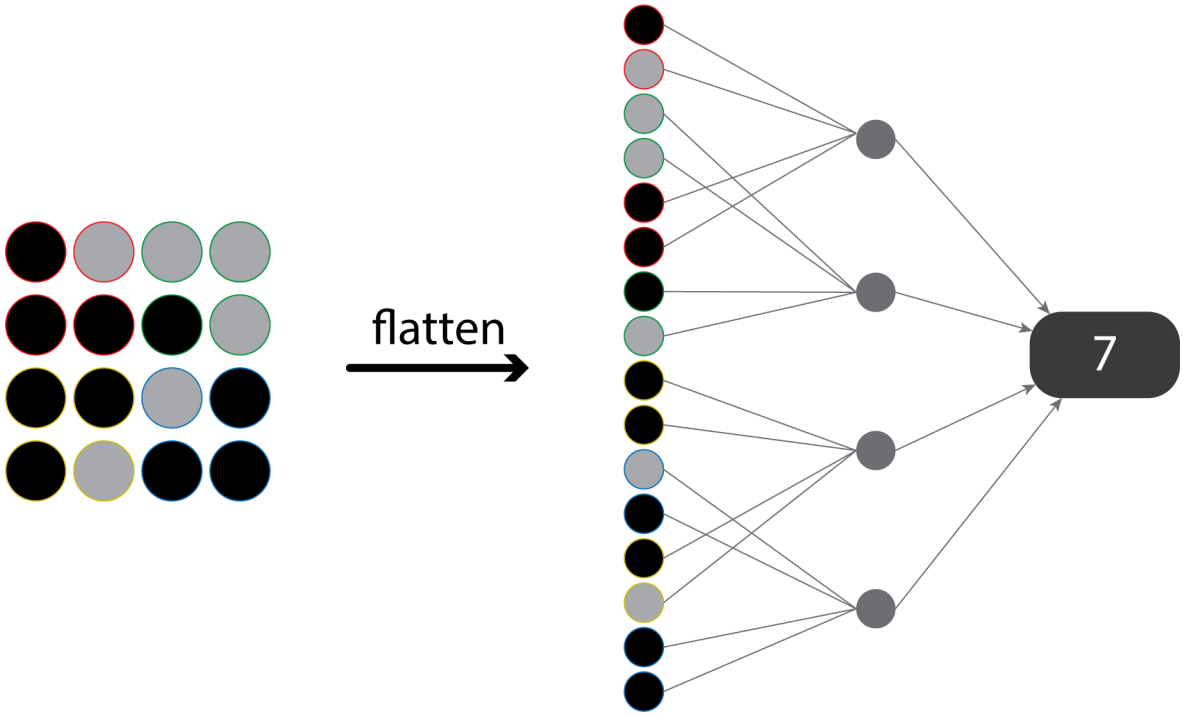


Figure 2.3: *Sparse connectivity. In contrast to the previous figure, fewer lines connect the input and hidden layers*

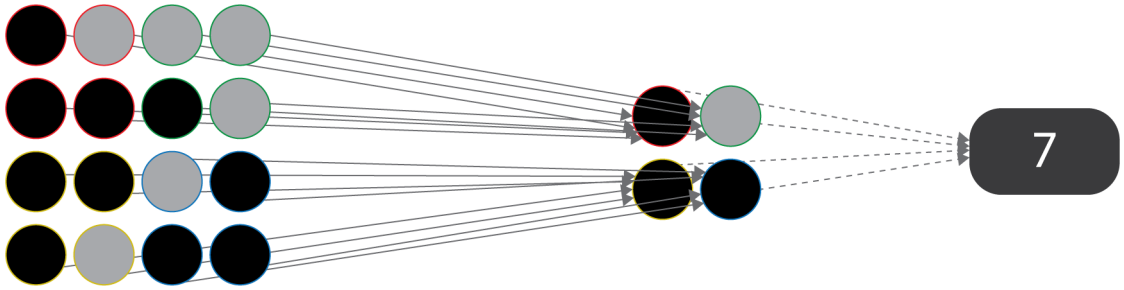


Figure 2.4: *Sparse connectivity after rearrangement.*

does further reduce the storage requirements of the model to k parameters. The particular form of parameter sharing causes the layer to have a property called **equivariance to translation**.

To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function $g(x)$ if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, that is, shifts it, then the convolution function is equivariant to g. For example, let I be a function giving image brightness at integer coordinates. Let g be a function mapping one image function to another image function, such that $I' = g(I)$ is the image function with $I'(x,y) = I(x-1,y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I, then apply convolution, the result will be the same as if we applied convolution to I, then applied the transformation g to the output. With images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image.

Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations (section 2.8). To illustrate these principles in action, we shall use a hand picked filter that used to detect edges in a image, see below.

$$K = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

These filters are called high pass filters. They enhance high frequency components in an image. Frequency in images just like in signals is the rate of change of the intensity,

which areas in neighboring pixels that rapidly changes for example from very dark to very light (in grayscale images). See fig. 2.5 to see the effect of applying the above filter to a grayscale image.

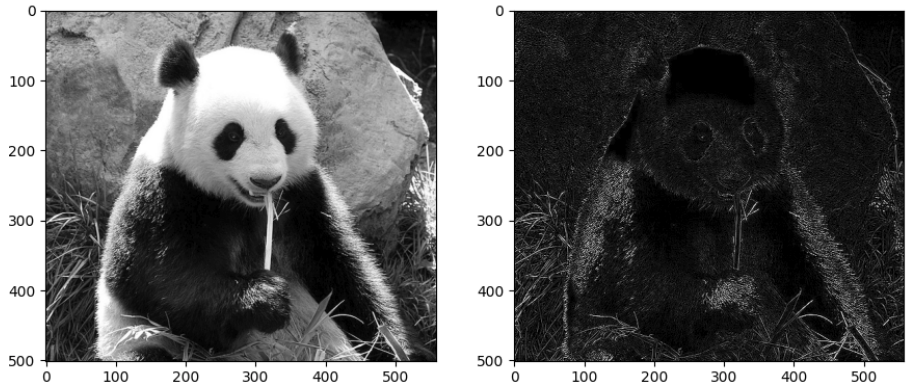


Figure 2.5: *2D convolution on a practical example.*

As we can see in fig. 2.5, where there is no change or little change of intensity in the original picture, the high pass filter block those areas out and turn the pixels black. But in the areas where a pixel is way brighter than its immediate neighbors, the high pass filter enhance the change and create a line. This has the effect of emphasizing edges. Edges are just areas in an image where the intensity changes very quickly. This images has been obtain by convolving the filter K with the image in the left, as we can see the three principles discussed above apply to this filter. The values of K didn't change while convolving (shared parameters). Space connectivity where the filter looks only to a small portion of the image at a time. And the equivariant translation, where we clearly see that no matter the position of the edge in the image the filter successful highlight it.

2.3 Convolutional layer

The convolutional layer is produced by applying a series of many different image filters, also known as convolutional kernels, to an input image.

In the example shown in fig. 2.6, 4 different filters produce 4 differently filtered output images. When we stack these images, we form a complete convolutional layer with a depth of 4, see fig. 2.7.

In case of colored images, computer interprets them as 3D-tensor $(Height \times width \times channels)$. Here, channels are the RGB channels. When performing convolution, the kernel K is itself chosen to be three dimensional as well. A typical kernel K would be $3 \times 3 \times 3$. The resulting output feature map would be $(Height \times Width)$. In order to depict multiple patterns in the image, instead of having a single kernel, multiple kernel are defined. Now each resulting output feature map can be considered as an image channel and stack them to

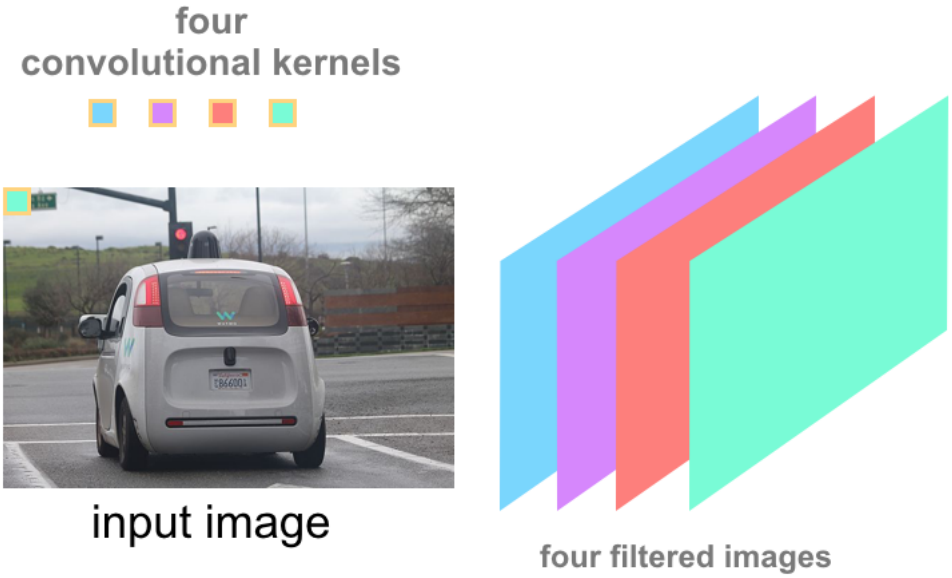


Figure 2.6: *Multiple filters for multiple pattern detection [1].*

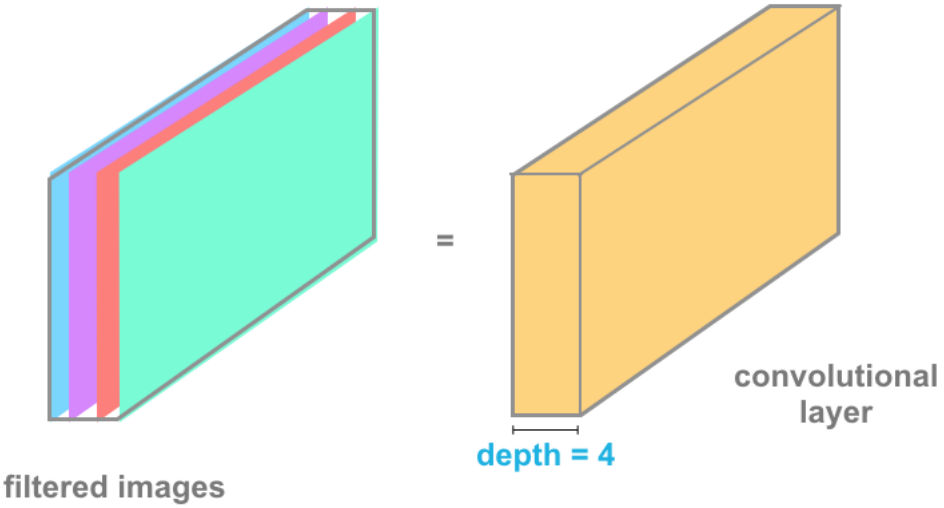


Figure 2.7: *A complete convolutional layer with 4 filters [1].*

get a 3 dimensional array. The latter 3D array can be used as input to another convolutional layer to discover patterns within the patterns that we discovered in the first convolutional layer. This operation can be repeated multiple times to discover various patterns within the input image.

In CNNs, inference works the same way as old plain neural network. Both convolutional and Dense layers have weights and biases and initial randomly generated. Therefore, in the case of CNNs where the weights take the form of convolutional kernel or filters, those kernels are randomly generated and so are the patterns that they're initially designed to detect. As with FC networks, when we construct a CNN, we will always specify a loss function. In the case of multiclass classification, this will be categorical cross-entropy loss (eq. (1.7)). Then as we train the model through back propagation, the filters are updated at each iteration to take on values that minimizes the loss function. In other words, the CNN determines what kind of patterns it needs to detect base on the loss function.

2.4 Stride and padding

The behavior of a CNN can be controlled by specifying the number of filters and the size of each filter, these are referred to as **hyper-parameters**. For instance, to increase the number of nodes in a convolutional layer, you could increase the number of filters. To increase the size of the detected patterns, you could increase the size of the filters. But there are more hyper-parameters than we can tune. One of these hyper-parameters is referred to as the stride of the convolution. The stride is just the amount by which the filter slides over the image. In the previous example of fig. 2.1, the stride was one. We move the convolution window horizontally and vertically across the image one pixel at a time [1]. The width and height of the output of the convolution is given by eq. (2.5), with an input image of $n \times n$, with an $f \times f$ filter:

$$(n-f+1) \times (n-f+1) \tag{2.5}$$

If we introduce the stride parameter s, eq. (2.5) can be rewritten as follow:

$$(\lfloor \frac{n-f}{s} \rfloor + 1) \times (\lfloor \frac{n-f}{s} \rfloor + 1) \tag{2.6}$$

One downside of the convolution operation is the shrinking input dimensions. Indeed, according to eq. (2.5), the input dimension shrinks each time by few pixels which can be an undesirable effect in very deep networks, where the image can shrink to very small dimensions. Another downside of the convolution is, the top left pixel (or corners of an image in general) is only involved in one pass of the filter, whereas if we take a pixel in the middle, then many 2×2 regions will overlap that pixel. It is as if the pixels at the corners are used much less in the output, so information is thrown away near the edge of

the image. Therefore to solve both of this problems, before applying the convolution we can pad the image with additional boarders, for instance 1 pixel, see fig. 2.8. Therefore the width and height of the output feature map is calculated as:

$$\lfloor \frac{n-f+2p}{s} \rfloor + 1 \times \lfloor \frac{n-f+2p}{s} \rfloor + 1 \tag{2.7}$$

Now with this additional boarder of zeros, the output feature maps' dimensions can be made equal to the input's dimension by setting the appropriate padding value. And the corner pixels contribute more in the output feature map.

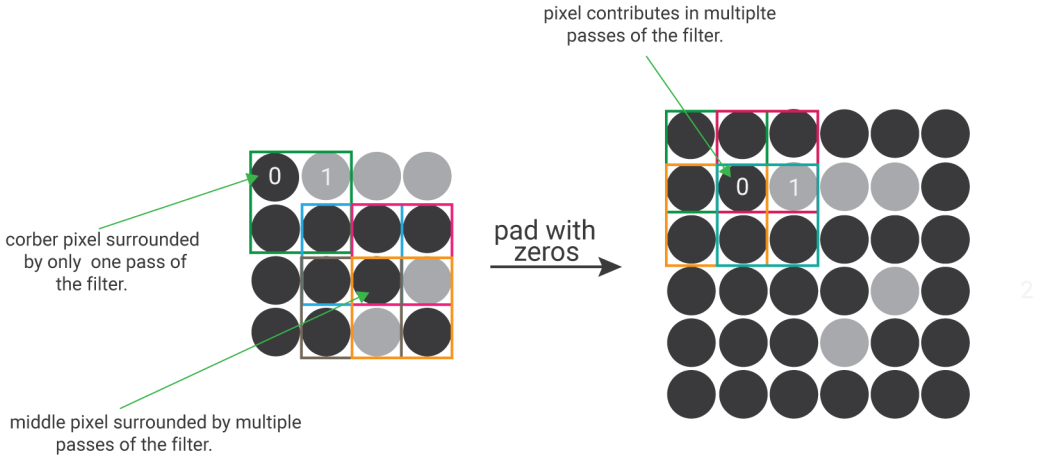


Figure 2.8: *Padding example.*

2.5 Pooling

Pooling function is the next type of layer in CNNs. It replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include average pooling of a rectangular neighborhood [21]. See fig. 2.9 on how to perform max pooling.

We clearly see from fig. 2.9, As in the convolution operation, we slide a window across the image typically a 2×2 window. The value of the corresponding node in the max pooling layer is calculated by just taking the maximum of the pixels contained in the window. The pooling function is applied independently on every feature map in the input stack. The output is a stack with same number of feature maps with width and height reduced by a factor of two.

In all cases, pooling helps to make the representation approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is.For example, when determining whether an image contains a face,

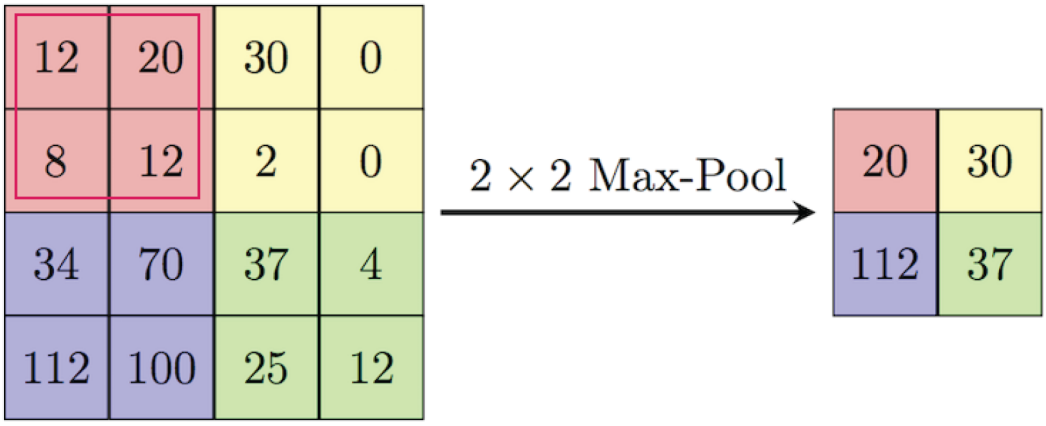


Figure 2.9: *Maxpooling example.*

we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. Another improvement that pooling brings is the computational efficiency of the network. The reason being is that pooling reports summary statistics for regions spaced with stride s (typically 2 is used), therefore the next layer has roughly s times fewer inputs to process and reduces the memory requirements for storing parameters [21].

Therefore, most CNNs are composed of only those two layers: Pooling and convolution. We begin with convolution layers which detects regional patterns in an image using a series of filters. Typically, just like fully connected networks, an activation function is applied to the output feature maps. ReLU activation function is used as it has proven to be extremely efficient in object classification tasks. Then pooling layers follow the convolutional layers to reduce the dimensionality of their input tensors. CNNs are designed with the goal of taking an input image and gradually making it much deeper than it is tall or wide. As the network gets deeper, it is actually extracting more and more complex patters and features that help identify the content and objects in an image. CNNs are usually referred to as **feature extractors**. Another issue that rises when training CNNs, is the input image dimensions. Since training requires large data-sets of thousands of images, it no surprise that these images are of different sizes and shapes. Therefore CNNs requires a fixed sized input due to batch training. Indeed, instead of passing one image at a time through the network, we usually pass batches of images which are just stacks of images. But in order to do that, all the images have to have the same width and height. So, we have to pick an image size and resize all of our images to that same size before doing anything else.

2.6 Case studies

In the few last chapters, we learned about the basic building blocks such as convolutional layers, pooling layers and fully connected layers of conv nets. It turns out a lot of the past few years of computer vision research has been on how to put together these basic building blocks to form effective convolutional neural networks, focusing on the *object classification task*. One of the best ways to get intuition on how to build conv nets is to read or to see other examples of effective conv nets, and it turns out that a net neural network architecture that works well on one computer vision task often works well on other tasks as well. Indeed those same networks discussed in this section are used as feature extractors or otherwise called **backbones** for object detection networks since an object detection requires the classification of objects and their localization. Therefore, instead of training those networks from scratch they build upon those already good Classifiers, see section 2.7.

2.6.1 VGG-16

VGG16 is a CNN model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition". The model achieves 92.7% top-5 test accuracy in ImageNet [34], which is a data-set of over 14 million images belonging to 1000 classes [14]. It was one of the famous model submitted to one of the most prestigious computer vision competition. It makes the improvement over state of the art at that time by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using graphical processing tchnology. see fig. 2.10.

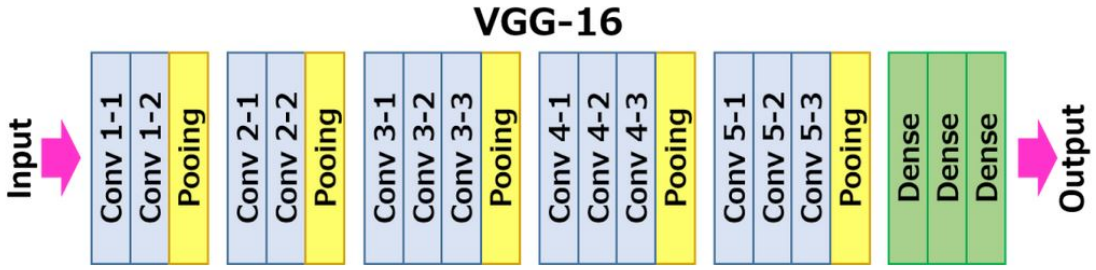


Figure 2.10: *VGG-16 structure by layers.*

The ConvNet configurations are outlined in fig. 2.11. The nets are referred to their names (A-E). All configurations follow the generic design present in architecture and differ only in the depth: from 11 weight layers in the network A (8 convolutional layers and 3 fully connected layers) to 19 weight layers in the network E (16 convolutional layers

and 3 fully connected layers). The width of convolutional layers (the number of channels) is rather small, starting from 64 in the first layer and then increasing by a factor of 2 after each max-pooling layer, until it reaches 512. See fig. 2.11.

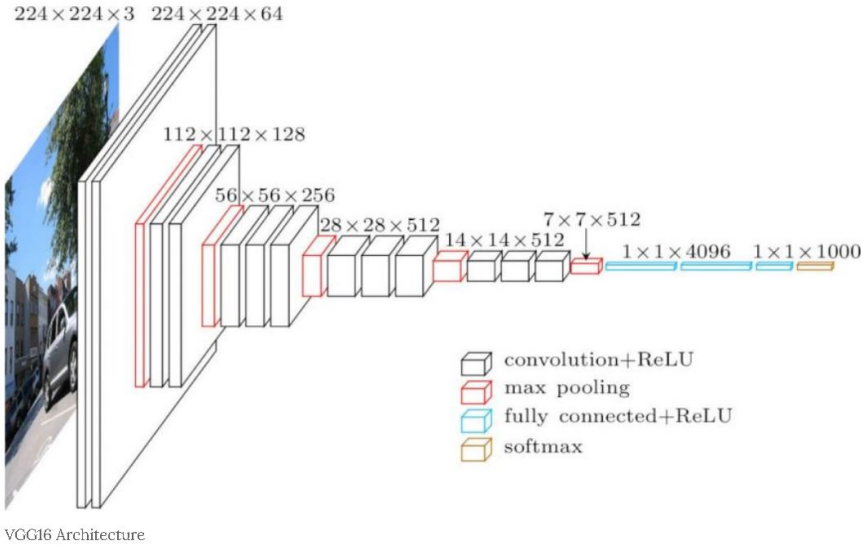


Figure 2.11: *VGG-16 network configuration.*

2.6.2 Mobilenet

MobileNet is a CNN architecture model used for object detection and image Classification, generally used in small applications.There exists a variety of models designed for the same purpose as well but the reason why MobileNet stand out is that it requires much less computation power to run or apply transfer learning to.This characteristic is what makes it optimal to run on embedded systems in general, computer systems without GPU or low computational efficiency, as well as Mobile devices which don't have the necessary hardware to run more costly models. Needless to say, the use of Mobilenet comes with a significant compromise in the accuracy of the results. It is also best suited for web browsers as browsers have limitation over computation, graphic processing , and storage. Mobilenet architecture is distinguished by an essential features know as "Depth-wise Separable Convolution" [9].

Before we get into the definition of "Depth-wise Separable Convolution" we need to go over some aspects of the convolution operation. Let's consider an input matrix of shape $D_f \times D_f \times M$ as shown in fig. 2.12. If our input was an RGB image then M would be equal to 3. If we apply a convolution using a filter of shape $D_k \times D_k \times M$ we would obtain an output of size $D_G \times D_G \times 1$ if we apply the same convolution using N filters of the same shape and concatenate the results we would obtain an output shape of $D_G \times D_G \times N$, see fig. 2.13.

Since the multiplication operation is more expensive relative to the addition, let's

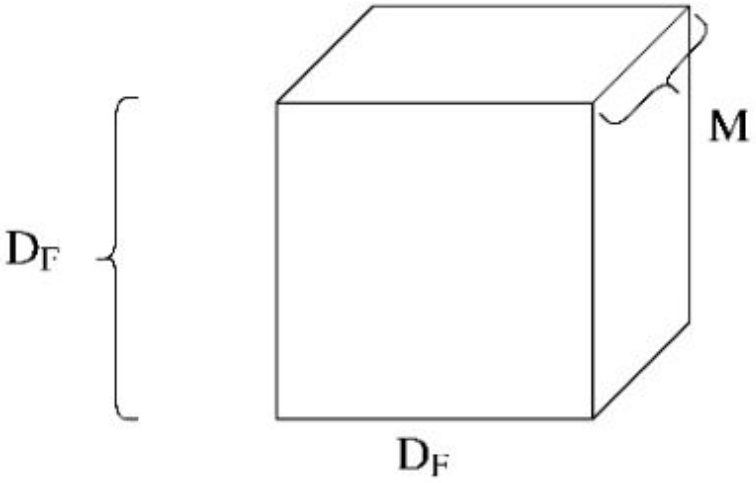


Figure 2.12: *Convolution input.*

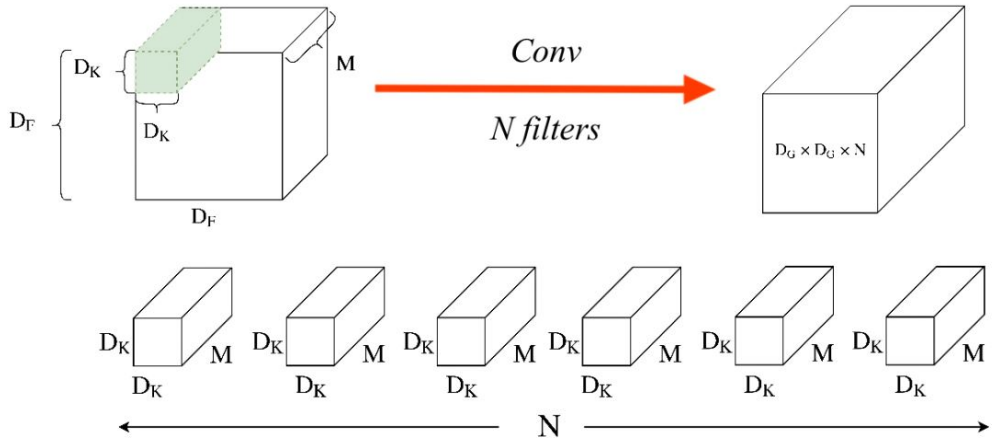


Figure 2.13: *Convolution operation.*

consider the cost of the convolution operation with regards to the number of multiplications. For one convolution step for one kernel the number of multiplications is $D_k \times D_k \times M$, for and entire convolution step for one filter the number of multiplications is $D_G \times D_G \times D_k \times D_k \times M$ therefore when we account for N filters the number of multiplication for a convolutional layer is $D_G^2 \times D_k^2 \times M$ With this in mind, we can introduce the concepts of "Depth-wise Convolution" and "Point-wise convolution", which when put together yield a " Depth-wise Separable Convolution ". Unlike simple convolution, Depth-wise Convolution applies convolution to single input channel at a time, using M filters of shape $D_k \times D_k \times 1$ see fig. 2.14. Point-wise convolution applies N filters of shape $1 \times 1 \times M$ to the output of the Depth-wise convolution and by concatenating the results we obtain the same output shape as simple convolution, see fig. 2.15 [9].

computing the number of multiplications for the entire process gives $M \times D_G^2 \times (D_k^2 +$

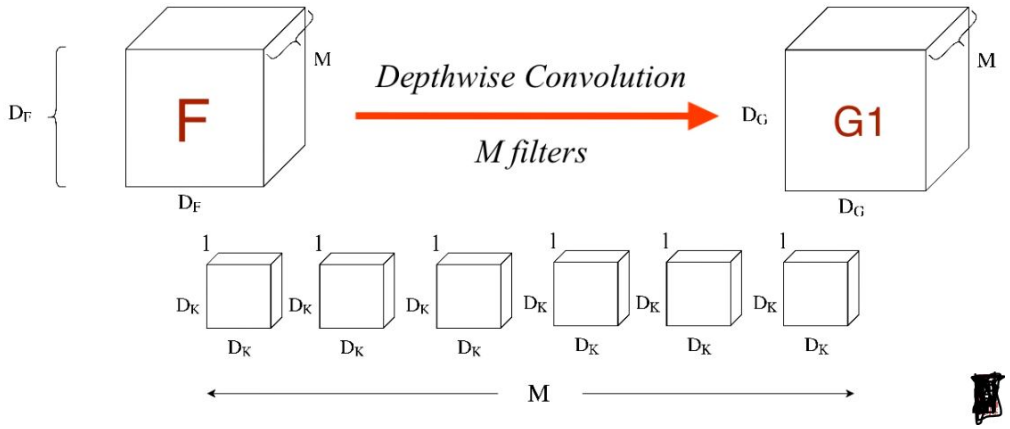


Figure 2.14: *Depth-wise Convolution.*

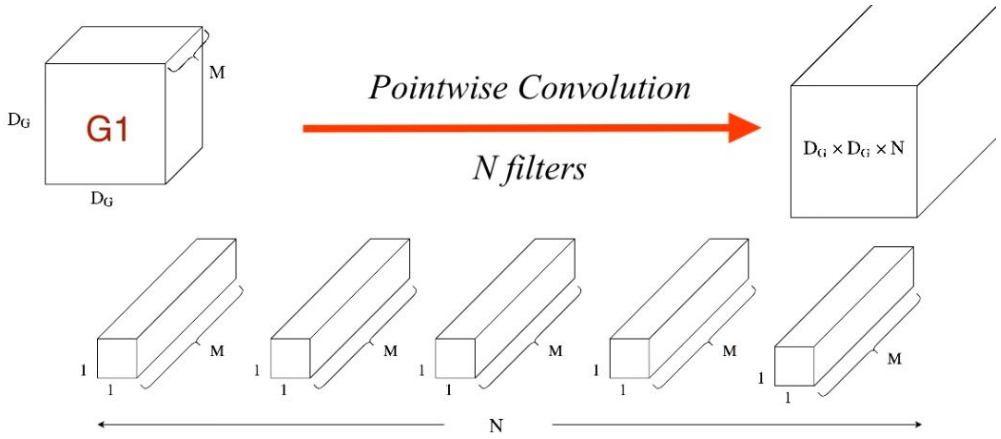


Figure 2.15: *Point-wise Convolution.*

N) Which is less than the cost of simple convolution. But to get a an understanding of how much computational power is reduced we should compute the ratio

$$\frac{number\ of\ multiplications\ for\ DSC}{number\ of\ multiplications\ for\ simple\ conv}$$

Which is found to be

$$\frac{1}{N}+\frac{1}{D_k^2}$$

By taking an example of $D_k = 3$ and $N = 1024$ we get a ratio of approximately $\frac{1}{9}$ which signifies a substantial decrease in computational requirements.

2.6.2.1 Mobilenet model structure

The Mobilenet model is composed of convolutional and Max Pool layers where the full structure is demonstrated in the table seen in fig. 2.16 [9].

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 2.16: *mobilenet structure table.*

2.6.3 Inception

The Inception network was an important milestone in the development of CNN classifiers. Prior to its inception (pun intended), most popular CNNs just stacked convolution layers deeper and deeper, hoping to get better performance.The Inception network on the other hand, was complex (heavily engineered). It used a lot of tricks to push performance; both in terms of speed and accuracy. Its constant evolution lead to the creation of several versions of the network. The popular versions are : Inception v1, Inception v2, Inception v3, and Inception Res-Net. Each version is an iterative improvement over the previous one. Understanding the upgrades can help us to build custom classifiers that are optimized both in speed and accuracy.

2.6.3.1 Inception V1

The Problem addressed by the developers of this model is the extreme large variation in the size of the salient parts in the image. For instance, An image of a dog can have any of the forms shown in fig. 2.17. The area occupied by the dog is different in each image. This significant variation in the location of the relevant features of the object we wish to detect and classify requires choosing the right kernel size for the convolution operation, which becomes a complicated task. A larger kernel is preferred for information that is distributed more globally, and a smaller kernel is preferred for information that is distributed more locally. Considering the fact that very deep networks are prone to over-fitting in addition to the difficulty they pose in performing back-propagation across the layers it goes without saying that naively stacking large convolution operations is computationally expensive and will not improve network performance on new data.

The authors of the original paper suggested the use of multiple filters of different sizes in one layer rendering the network "wider" rather than “deeper” [12].

Figure 2.18 explains the core idea of this model. It performs convolution on an input, with 3 different sizes of filters $(1 \times 1.3 \times 3.5 \times 5)$. Additionally, max pooling is also performed. The outputs are concatenated and sent to the next inception layer.

As stated before, deep neural networks are computationally expensive. To make it cheaper, the authors limit the number of input channels by adding an extra 1×1 convolution before the 3×3 and 5×5 convolutions. Though adding an extra operation may seem counter intuitive, 1×1 convolutions are far less expensive than 5×5 convolutions, and the reduced number of input channels also help (similar to the method used in Mobilenet). Do note that however, the 1x1 convolution is introduced after the max pooling layer, rather than before [11], see fig. 2.19

Using the dimension reduced inception module, a neural network architecture was built. This was popularly known as GoogLeNet (Inception v1). The architecture is shown in fig. 2.20.

Figure 2.17: *Examples of dog images.*

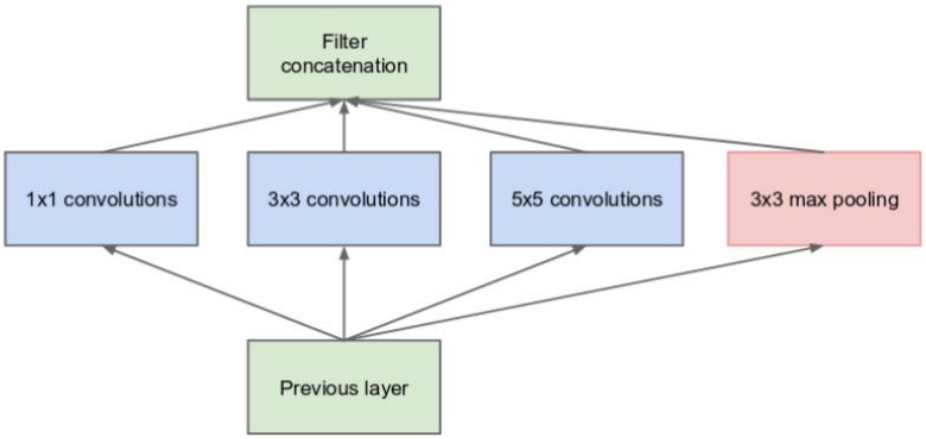


Figure 2.18: *Inception network layer general structure.*

GoogLeNet has 9 such inception modules stacked linearly. It is 22 layers deep (27, including the pooling layers). It uses global average pooling at the end of the last inception module.

Needless to say, it is a pretty deep classifier. As with any very deep network, it is subject to the vanishing gradient problem.

To prevent the middle part of the network from "dying out", the authors introduced two auxiliary classifiers (The purple boxes in fig. 2.20). They essentially applied softmax to the outputs of two of the inception modules, and computed an auxiliary loss over the same labels. The total loss function is a weighted sum of the auxiliary loss and the real loss. Weight value used in the paper was 0.3 for each auxiliary loss [11]. Needless to say, auxiliary loss is purely used for training purposes, and is ignored during testing.

$$totalloss = real_loss + 0.3aux_loss_1 + 0.3aux_loss_2$$

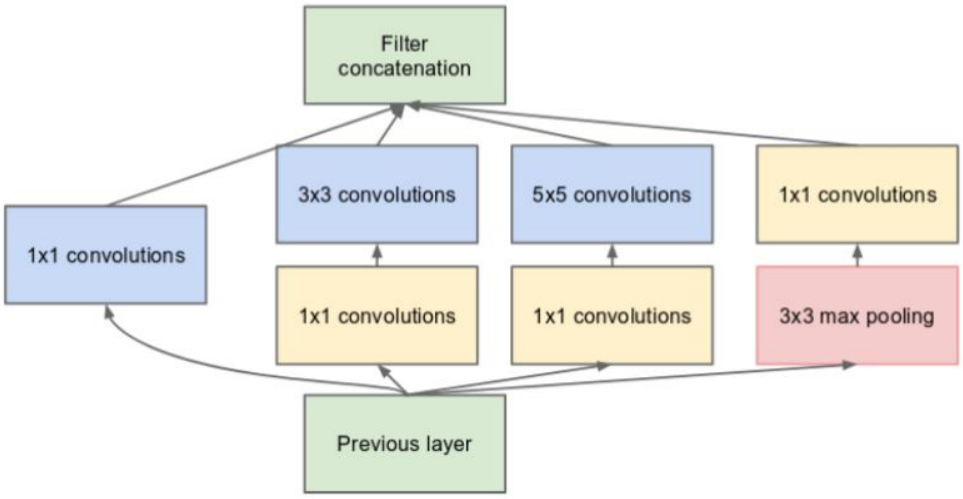


Figure 2.19: *Inception layer with added* 1×1 *convolution operation.*

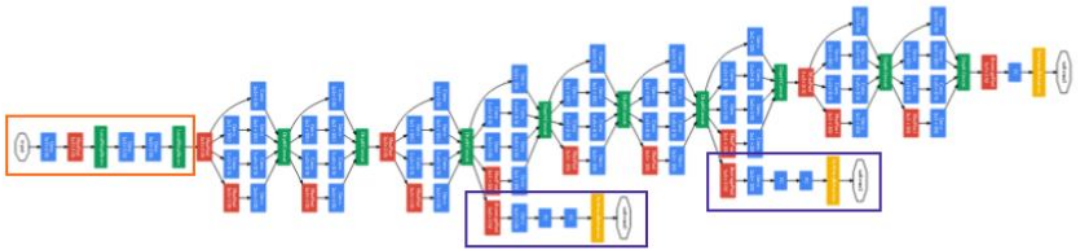


Figure 2.20: *Inception network structure.*

2.6.4 Res-Net

ResNet, short for Residual Networks is a classic neural network used as a backbone for many computer vision tasks. This model was the winner of ImageNet challenge in 2015. The fundamental breakthrough with ResNet was that it allowed us to train extremely deep neural networks with 150+layers successfully [27]. Prior to ResNet, training very deep neural networks was difficult due to the problem of vanishing gradients.

AlexNet, the winner of ImageNet 2012 and the model that apparently kick started the focus on deep learning had only 8 convolutional layers, the VGG network had 19 and Inception or GoogleNet had 22 layers and ResNet 152 had 152 layers.

However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient extremely small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly. For this reason the creators of Res-Net introduced the idea of "Skip Connections"

2.6.4.1 Skip Connection

ResNet first introduced the concept of skip connection. fig. 2.21. illustrates skip connection. The figure on the left is stacking convolution layers together one after the other. We still stack convolution layers as before but we now also add the original input to the output of the convolution block. This is called skip connection. We must note that the addition operation occurs before the output goes through the ReLu (Rectification linear unit) function [27]. The main two reasons why skip connections work are :

- Skip connections mitigate the problem of vanishing gradient by allowing this alternate shortcut path for gradient to flow through.
- They allow the model to learn an identity function which ensures that the higher layer will perform at least as good as the lower layer, and not worse.

Figure 2.21: *Skip connection.*

2.7 Transfer learning

Usually, training very deep networks from scratch is a very tedious task; huge data-sets are required for the task to better generalize to real life situations. Modern CNNs usually take 2-3 weeks to train across multiple GPUs. However, it has been revealed that deep networks trained on natural images exhibits a curious phenomenon in common: on the first layer they learn general features similar to color blobs and edges. Such first layer features appear not to be *specific* to a particular data-set or task, but *general* in that they are applicable to many data-sets and tasks [22]. This means it may be useful to transfer this knowledge to other similar tasks. This technique is referred to as **transfer learning**. Deep CNNs are good candidates for this task because they are usually trained on general tasks (like image classification of daily life objects) and have many adjustable layers. As reference [22] states ,the transferability of features decreases as the distance between the base task and target task increases, but that transferring features even from distant tasks can be better than using random features. A final surprising result is that initializing a

network with transferred features from almost any number of layers can produce a boost to generalization that lingers even after "fine-tuning" to the target data-set. One of the strategies used when using transfer learning is referred to as **fine-tuning**. This simply means retraining the whole or parts of the pre-trained CNN. This is done by retraining with the new data-set without changing the architecture or reinitializing the weights (but some new layers might be added or changed depending on the task at hand). The existing weights are said to be *fine-tuned* to the new task at hand [23].

2.8 Data augmentation

Data augmentation is a strategy that enables practitioners to increase the diversity of data available for training models, without actually collecting new data. This is achieved by applying random (but realistic) transformations such as image rotation, cropping, padding, and horizontal flipping .... This is a good practice, since augmenting the size of the training set means more data to learn from, which makes the network even robust.

Chapter 3

CNN application: Object detection

The concept of convolution and convolutional neural networks has been applied to many real life problems: including object classification, object detection, speech recognition, disease depiction in medical images, self driving cars, and many more. In this is chapter, we will focus on the theory behind the state-of-the-art detection systems: YOLO object detection which stands for you only look once and R-CNN which stands for Region-CNN. Object detection is the task of detecting, meaning classifying and localizing instances of semantic objects of a certain class (in our case Algerian car license plates along with their digits). An object detection algorithm should not only be able to classify an object but as well as localizing it in an image by drawing a bounding box around it, see fig. 3.1.

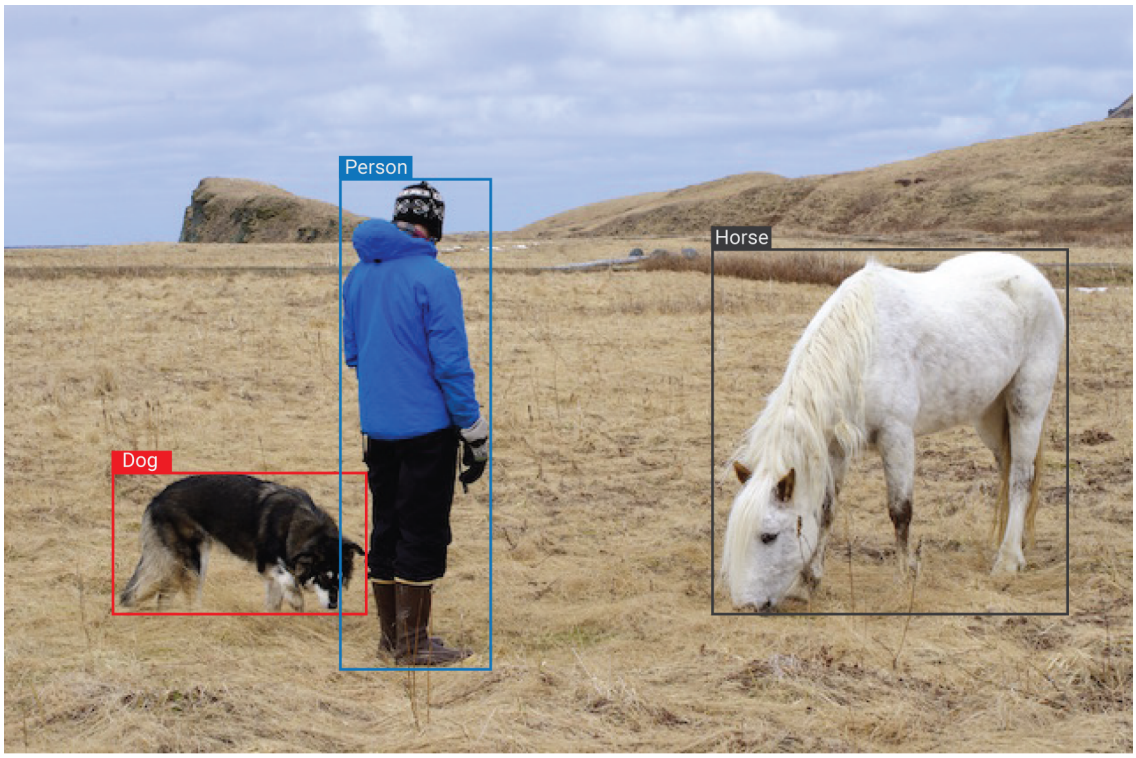


Figure 3.1: *Example of what an object detection system should accomplish.*

3.1 YOLO: you only look once

Over the past few years, the YOLO algorithm have evolved quite a lot going from YOLOv1 all through version four. The different improvements that this algorithm went through are just the fruits of many research developments in the deep learning field incorporated into YOLO algorithm to make it more robust and less prone to errors. In this section we shall present the version three of YOLO. Version four has only been developed in April 2020 during the middle of the pandemic. Many techniques have been included in this last paper which makes a bit difficult since we have to go through all the new details. Therefore we shall only present version three.

3.1.1 Bounding boxes

The YOLO algorithm divides the input image into an $S \times S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object [26]. Each grid cell predicts B bounding boxes, using anchor boxes. Anchor boxes are predefined boxes of certain width and height. They are defined to capture the scale and aspect ratio of specific object classes you want to detect. Anchor boxes are typically chosen based on object sizes in the training data [4], see fig. 3.2. Anchor boxes have been introduced to solve two issues (second issue will be discussed in section 3.1.2). Objects in the YOLO algorithm are associated with grid cells that their centers fall into. If two objects' centers fall into the same grid cell, we will not be able to predict both objects. Therefore, we can associate each grid cell with multiple anchor boxes, each responsible to detect only one object in that cell. A typical number of boxes used is three, see fig. 3.2. That is the first issue anchor boxes solves.

Each bounding box is associated with a confidence score, which reflects how confident the network is that the bounding box contains an object (also called objectness) [26]. This should be ideally 1 if there is an object otherwise 0 [25].Then b_x, b_y, b_h, b_w that defines the bounding box, where b_x and b_y represents the box's center coordinates and b_h, b_w, the height and width respectively [5]. And the class confidence scores. For instance if we are building a self driving car object detection system, we may want to detects cars, pedestrians and motorcycles. Therefore, each grid cell will be associated with an $((5 + number\ of\ classes\ to\ detect) \times number\ of\ anchor\ boxes)$ dimensional vector. As we can see on fig. 3.2, the anchor boxes capture the scale and aspect ratio of cars and pedestrians. Indeed, most cars and humans will have approximately the same scale and aspect ratio. The vector y is composed of the objectness score as well as the bounding boxes and the class probabilities repeated for each anchor box. Here two anchor boxes have been used. YOLOv3 uses 3 anchor boxes. The image has been divided into a 3×3 grid just for illustration. The vector y represents the manual labeling for the central cell.

Anchor box 1 is associated with the pedestrian while the second one is associated with the car.

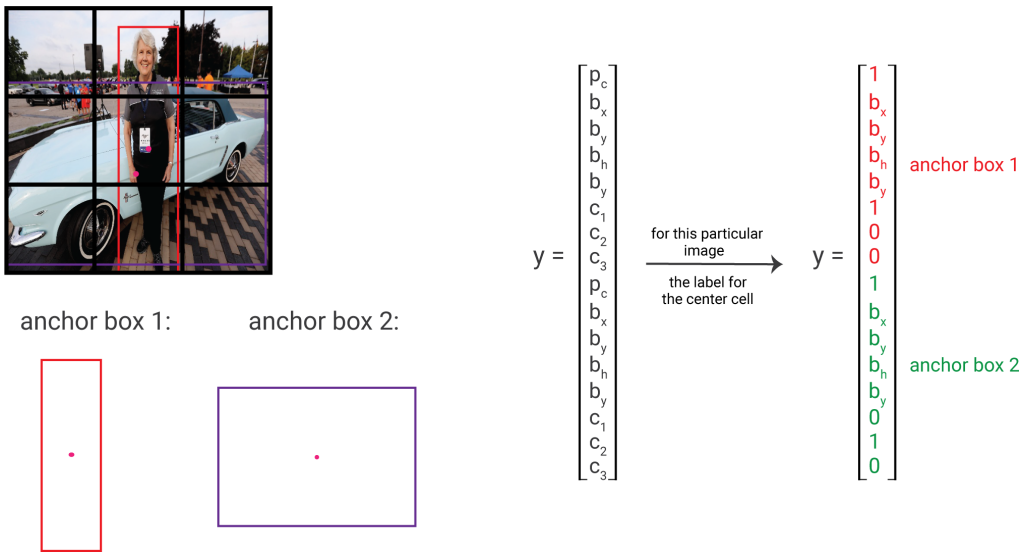


Figure 3.2: *Example of anchor boxes.*

3.1.2 Network design

The network is a series of convolutional and pooling layers chosen so that the network eventually maps the input image $W \times H \times 3$ to an output volume $S \times S \times ((5 + number\ of\ classes\ to\ detect) \times number\ of\ anchor\ boxes)$. YOLO's convolutional layers down-sample the image by a factor of 32, 16, 8. The YOLOv3 network has therefore 3 outputs instead of one, but we will be focusing on only one as the same calculation happen at each scale. The exact architecture is discussed in chapter 4. Now, to train the convolutional neural network, we pick an image size of 416×416. This number has been chosen because we want an odd number of locations in our feature map so there is a single center cell. Objects, especially large objects, tend to occupy the center of the image so it's good to have a single location right at the center to predict these objects instead of four locations that are all nearby [24]. By using an input image of 416×416 we get an output feature map of 13×13. The second issue anchor boxes address is the training instability [24]. In fact, during the early epochs of training if b_x *and* b_y are randomly initialized, the network struggles to converge to the right ground truth box's center. To overcome this problem, YOLO predicts location coordinates b_x *and* b_y relative to the grid cell. This bounds the ground truth to fall between 0 and 1. We use sigmoid activation to constrain the network's prediction to fall in this range. The network predicts B bounding boxes at each cell in the output feature map. The network predicts 5 coordinates for each bounding box t_x, t_y, t_h, t_w *and* t_0, see fig. 3.3. If the cell is offset from the top left corner of the image by $(c_x; c_y)$ and the anchor box has width and height pw and p_h, then the predictions

correspond to [24]:

$$b_x = \sigma(t_x) + c_x \tag{3.1}$$

$$b_y = \sigma(t_y) + c_y \tag{3.2}$$

$$b_w = p_w e^{t_w} \tag{3.3}$$

$$b_h = p_h e^{t_h} \tag{3.4}$$

Since we constrain the location prediction, the parametrization is easier to learn, making the network more stable [24], see fig. 3.4. The question that naturally rises is: How, at the beginning, do we get p_w and p_h? Otherwise, how to assign an anchor box to a ground truth object ? The answer to this question is given is section 3.1.3 as we need to define an important function to proceed, see section 3.1.3.

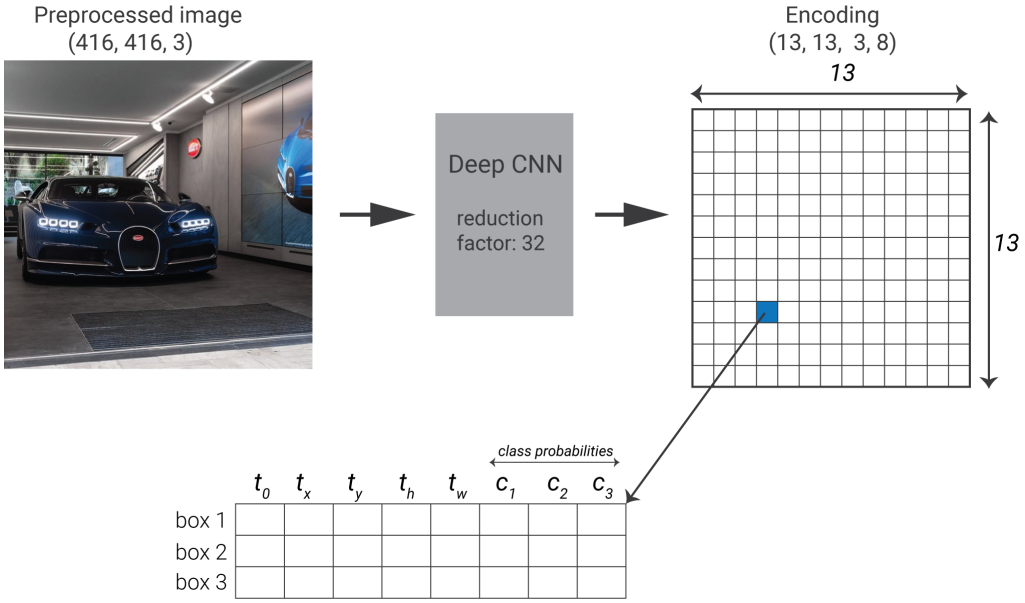


Figure 3.3: *The true output of YOLOv3 after introducing the training instability issue.*

In fig. 3.3, the network outputs a $13 \times 13 \times (8 \times 3)$ in this case, or simply put $13 \times 13 \times 24$ output volume. Each grid cell outputs three bounding boxes.

During training, we optimize the following multi-part loss function. As we can see in eq. (3.5), the first sum is over scales, meaning different regions of the network. Indeed, the network used in YOLOv3 does have only one output but three. The architecture of the network is discussed further in chapter 4.

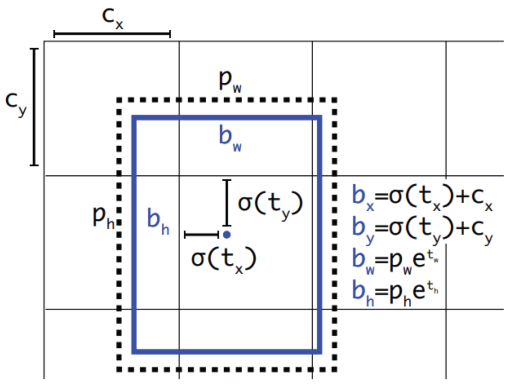


Figure 3.4: *Bounding box calculation [24].*

$$\sum_{scales} \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{obj} \left[(t_x - \hat{t}_x)^2 + (t_y - \hat{t}_y)^2 + (t_w - \hat{t}_w)^2 + (t_h - \hat{t}_h)^2 \right]$$
$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{obj} \left[-\log(\sigma(t_o)) + \sum_{k=1}^{C} BCE(\hat{y}_k, y_k \right]$$
$$+ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{noobj} \left[-log(1 - \sigma(t_o)) \right] \quad (3.5)$$

where $1_{i,j}^{obi}$ denotes if object appears in cell i and that the j^{th} anchor box in cell i is "responsible" for that prediction. If an anchor box is not assigned to a ground truth object, it incurs no loss for coordinate or class predictions, only objectness. In cells that contain an object, the bounding box coordinates are calculated using the sum-squared loss function. Each box predicts the classes the bounding box may contain using multi-label classification. In other words, binary cross-entropy loss is used (BCE). The same binary cross-entropy loss is used to for objecness prediction as eq. (3.5) states it.

3.1.3 Processing the algorithm's output

After training, the network at inference time will find multiple detections. In fact, for each cell in the $S \times S$ grid, using B anchor boxes, the algorithm will infer B bounding boxes for each cell, which makes a total of $B \times S^2$. Therefore an object can be detected multiple times. **Non-max suppression** is an algorithm that cleans up those detections and makes sure each object gets detected only once. Before discussing it though, let us introduce an important function called **Intersection over Union** (IoU for short) that calculates how much a box or a rectangle overlaps another. So, IoU calculates the area defined by the intersection of the two boxes and divide it by the area defined by their union, see fig. 3.5.

IoU is an evaluation metric used to measure the accuracy of an object detection system on a particular data set. Indeed, most object detection algorithm will judge a detection to

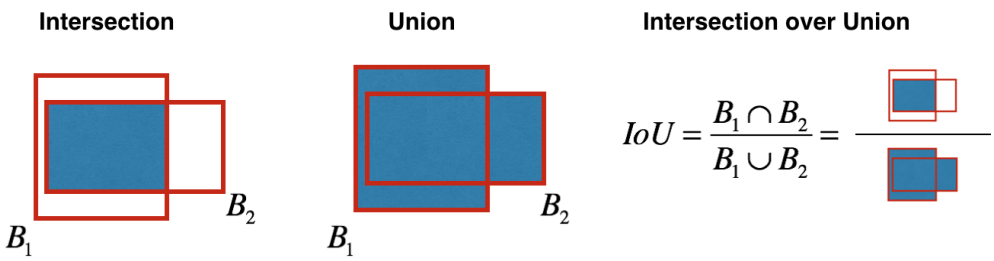


Figure 3.5: *Intersection over union metric.*

be correct if the IoU between the ground truth box and the detected box is more than 0.5, see fig. 3.6. We often see this evaluation metric used in object detection challenges such as the popular PASCAL VOC challenge [6].

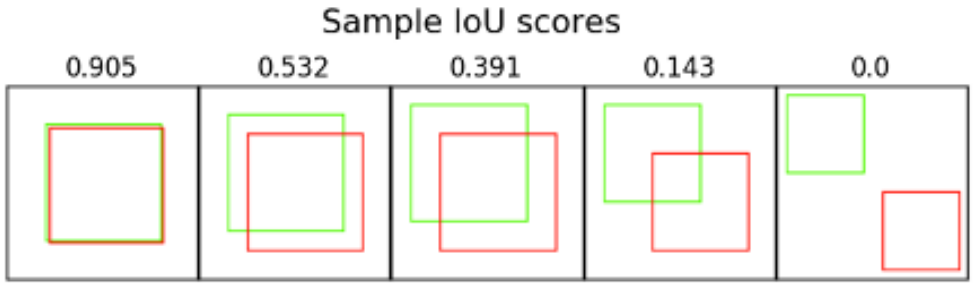


Figure 3.6: *Sample IoU scores.*

Back to our original question, how non-max suppression works. First, all the boxes having an $objectness \times the\ class\ probability$ less than or equal to some threshold are discarded (typical value of threshold is .6). While there any remaining boxes, we pick the box with the largest $objectness \times the\ class\ probability$ and output it as a prediction. Then we discard any remaining box with $IoU \geq 0.5$ with the box outputted in the previous step. This algorithm ensures that each object is detected only once.

In section 3.1.2, we discussed how the bounding boxes are being computed, and we finished it with a question: How does the anchor boxes being assigned to ground truth objects at the beginning? YOLOv3 assigns the anchor with the highest IoU overlap with a ground truth box.

3.2 Faster R-CNN

Several object detection techniques and models have been developed over the years. Each with its benefits and drawbacks. In this section we shall explore the faster region-CNNs technique to tackle this task. Faster R-CNN model is composed of two networks:

region proposal network (RPN) for generating region proposals and a network using these proposals to detect objects [33]. The main different here with its' predecessor Fast R-CNN is that the later uses an algorithm called "selective search" to generate region proposals [19]. The time cost of generating region proposals is much smaller in RPN than selective search, since the RPN network does a significant part of computation which is overlapping with the computation needed for the object detection network. in short, RPN ranks region boxes (called anchors) from most likely to less likely to contain an object and proposes the ones most likely containing objects [33]. The architecture is shown in fig. 3.7.

3.2.1 Anchors

In the default configuration of Faster R-CNN, it considers 9 anchors at each position of an image. fig. 3.8 shows 9 anchors at the position $(320, 320)$ of an image with size $(600, 800)$. The colors represent three scales or sizes: $128 \times 128,\ 256 \times 256,\ 512 \times 512$. For each color we have three boxes that have height width ratios $1:1,\ 1:2$ and $2:1$ respectively. These two parameters are called "scales" and "aspect ratios", and they have a significant effect on the performance of our model. The RPN selects a position in a given image at every stride of 16 where it generates those 9 anchors. In an image of the same size as fig. 3.8 there will be 1989 (39×51) positions. This leads to 17901 (1989×9) boxes to consider. This number of anchors is hardly smaller than the technique of of sliding window and pyramid. The advantage here is that we can use region proposal network to significantly reduce the number of boxes that will be considered by the classifier network [33].

These anchors work well for Pascal VOC [18] data set as well as the COCO data set [35]. We have the freedom to design different kinds of anchors/boxes. For example, you are designing a network to detect passengers/pedestrians, you may not need to consider the very short, very big, or square boxes. A uniform set of anchors may increase the speed as well as the accuracy.

3.2.2 Region Proposal Network

The input to the RPN module is the feature map of an image, the RPN then generates centers on the original image for each "pixel" in a feature map obtained from a forward pass through a pre-trained CNN. It then generates 9 anchors around each center according to the specified scales and aspect ratios [33]. The output of a RPN is a set of probabilities for each anchor that determine the probability of a certain anchor being an object or not. It also outputs a set of error estimations for the anchors which overlap with a ground truth box. these outputs will be examined by a classifier and regressor to eventually check the occurrence of objects. To be more precise, RPN predicts the possibility of an anchor being background or foreground, and refines the dimensions of an anchor [33]. Since the RPN performs a

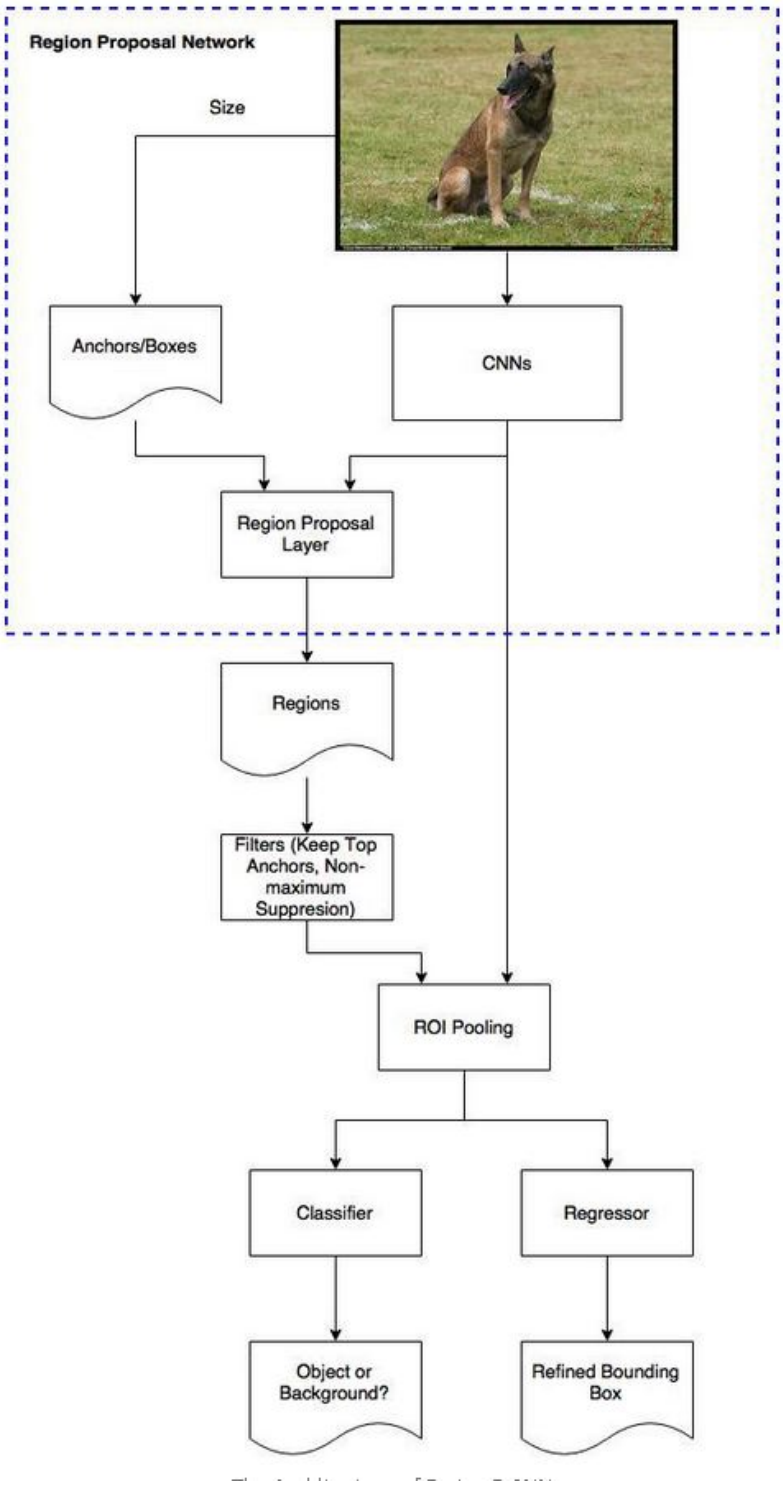


Figure 3.7: *Faster R-CNN model structure.*

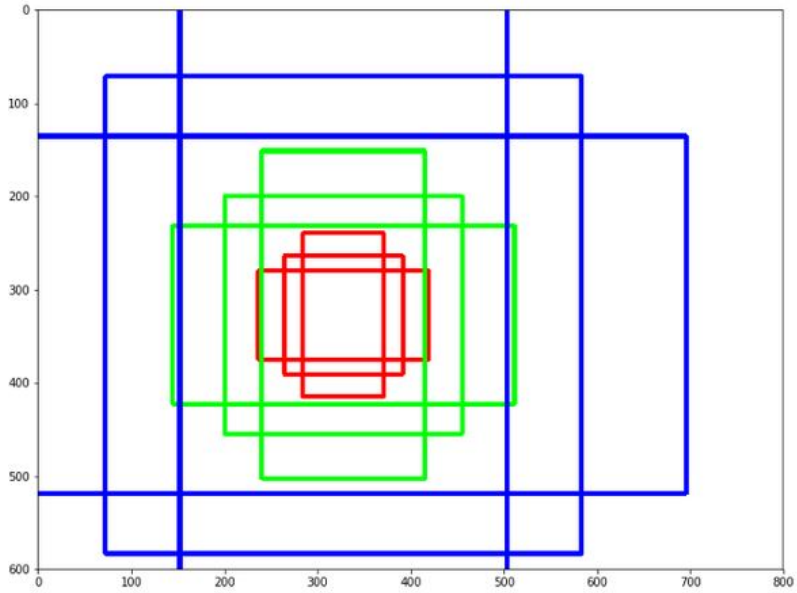


Figure 3.8: *Example of anchors at single location.*

classification task, it will go through a training process for which we must have a clear definition of the data-set and the labels. In this case our data-set is the anchors defined for each image. As for the labels; the basic idea is that we want to label the anchors having the higher overlaps with ground-truth boxes (the bounding box surrounding the object we wish to detect as foreground), and the ones with lower overlaps as background. For this we use the IOU (Intersection Over Union) function. If the value of the IOU is higher than a certain threshold then it would be labeled as foreground otherwise it is labeled as background [33].

RPN also performs a regression task on the same anchors in order to correct the dimensions and location of these same anchors. For each anchor it computes an estimation of error on the dimensions called t_w for the width and t_h for the height as well as on the location of the anchor center t_x and t_y such that

$$t_w = log(\frac{w}{w_a}) \tag{3.6}$$

$$t_h = log(\frac{h}{h_a}) \tag{3.7}$$

$$t_x = \frac{x - x_a}{w_a} \tag{3.8}$$

$$t_y = \frac{y - y_a}{h_a} \tag{3.9}$$

x, y, w, h are the ground truth box center coordinates, width and height. $x_a, y_a, h_a and w_a$ and anchor boxes center coordinates, width and height [33].

The final and most important component of the training process is the loss function

$$L(p_i,t_i) = (\frac{1}{N_{cls}})\sum_i L_{cls}(p_i,p_i^*) + \lambda(\frac{1}{N_{reg}})\sum_i p_i^* L_{reg}(t_i,t_i^*) \tag{3.10}$$

where p_i is the predicted probability of objectness and p_i^* is the actual score. t_i and t_i^* are the predicted coordinates and actual coordinates respectively. The ground-truth label p_i^* is 1 if the the anchor is positive and 0 if the anchor is negative [33].

3.2.3 ROI Pooling

Region of interest pooling (also known as RoI pooling) purpose is to perform max pooling on inputs of non-uniform sizes to obtain fixed-size feature maps (e.g. 7×7). This layer takes two inputs:

- Fixed-size feature map obtained from a deep convolutional network with several convolutions and max-pooling layers.
- An $N \times 5$ matrix of representing a list of regions of interest, where N is the number of RoIs. The first column represents the image index and the remaining four are the co-ordinates of the top left and bottom right corners of the region.

For every region of interest from the input list, it takes a section of the input feature map that corresponds to it and scales it to some predefined size (e.g., 7×7). The scaling is done by:

- Dividing the region proposal into equal-sized sections (the number of which is the same as the dimension of the output).
- Finding the largest value in each section.
- Copying these max values to the output buffer.

The result is that from a list of rectangles with different sizes we can quickly get a list of corresponding feature maps with a fixed size. Note that the dimension of the RoI pooling output does not actually depend on the size of the input feature map nor on the size of the region proposals. It's determined solely by the number of sections we divide the proposal into. One of the benefits of ROI pooling is processing speed. If there are multiple object proposals on the frame (and usually there'll be a lot of them), we can still use the same input feature map for all of them. Since computing the convolutions at early stages of processing is very expensive, this approach can save us a lot of time [33]. The fig. 3.9 below shows the working of ROI pooling.

The output of ROI pooling will be the input to a classifier network, which is a copy of the pre-trained backbone network we used to obtain the feature map, and which will be

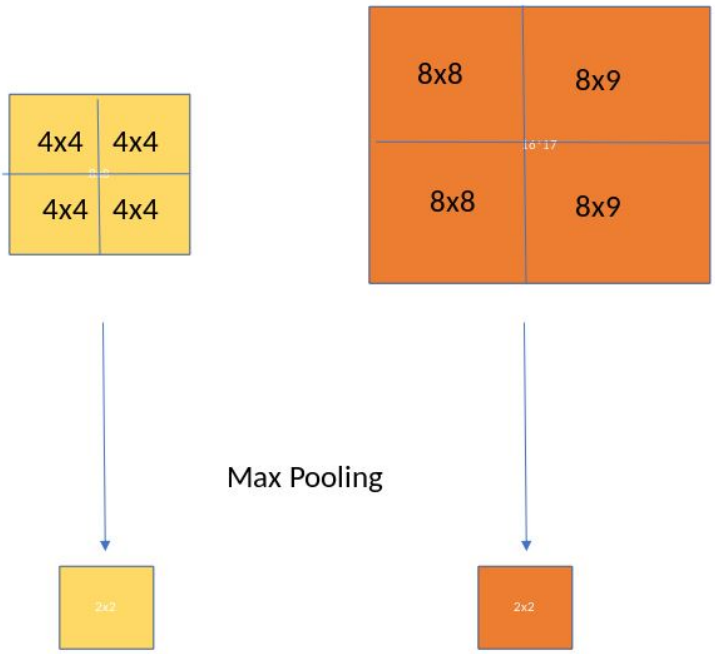


Figure 3.9: *RoI pooling operation.*

referred to as "Fast RCNN classifier network". This network will further branch out to a classification head and regression head. The loss function for the Fast-RCNN network is defined in the same way as the RPN loss, except for the significance of the variables. p_i is the predicted class scores for every class of objects we want to detect and p_i^* is the actual score. t_i and t_i^* are the predicted cooridinates and actual coordinates, respectively. The ground-truth label p_i^* is 1 for a certain class of objects if the region outputted by the ROI layer contains that object and 0 if it does not.

3.2.4 Faster RCNN training

For training the entire Faster RCNN model there must be a well defined loss function which encapsulates all of the losses mentioned before. It can be considered as an estimation for error in the model, regardless of where the error occurs. The total loss is defined as nothing more than the sum of both losses (RPN loss and Fast RCNN classifier network loss).

$$\text{Total loss} = \text{RPN loss} + \text{Fast RCNN classifier loss} \tag{3.11}$$

The training process would proceed using the same optimization and regularization techniques discussed earlier [33].

Chapter 4

Design and implementation of ALPR system

4.1 Workflow description

For building the ALPR application, we used a part-by-part approach rather than and end-to-end approach. The reason being is the lack of labeled data sets designed for this specific application, especially when it comes to Algerian license plates for which there are no published data sets. Needless to mention that the task at hand is relatively less tedious since Algerian plates contain no characters others than digits. Our system defines a "detect plate, detect digits" pipeline. Each step is given a dedicated module that runs in a sequential and independent way. The first module, called the "plate network", takes the full raw input image that detects plates in the image, crops the detected plates bases upon their bounding boxes then passes them to the second module as inputs. The detected plates are cropped with an extra margin around them to avoid the exclusion of important details (may be digits) for digit detection. The second module, called "the digits network" receives the detected plates from the first module, detects and recognize the 10 digit classes from 0 to 9, which are specific to Algerian license plates. The position of the bounding boxes outputted by the last module determines the order of each digit and a final prediction is given. The full system flow is given in fig. 4.1. We proceeded to dividing this task at hand into three main parts :

- Data collection and labeling.
- Plate detection and localization.
- Digit recognition.

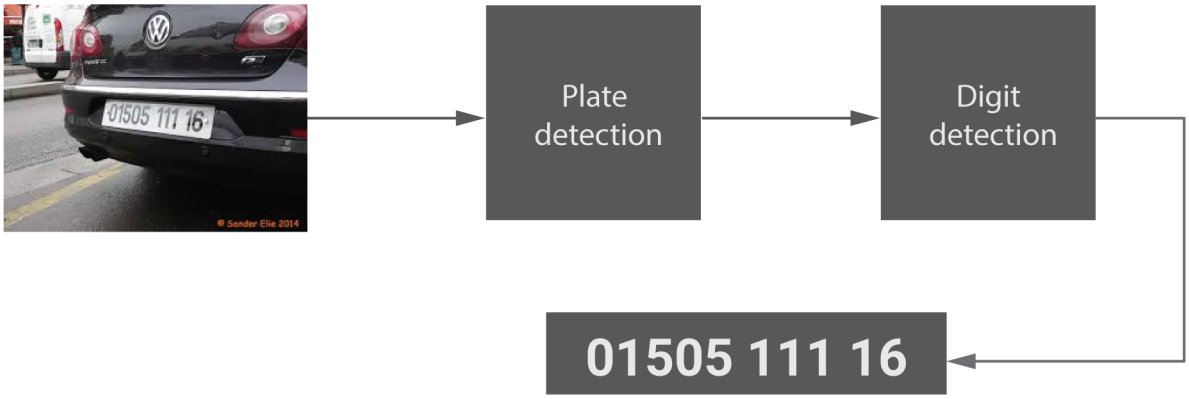


Figure 4.1: *The entire system flowchart.*

4.2 Data collection

Data collection includes the manual collection of images of license plates in different positions, angles, lighting, distance, size, color .... The data set contains close to 1000 images ready to be used for labeling. Figure 4.2 below shows some examples.

Figure 4.2: *Image collection example*

The second model which performs digit recognition will be trained on cropped images of plates only, which will be obtained by passing the original set of images through the plate detector and using the obtained plate bounding boxes to crop the plates.

4.3 Data Labeling

"Labeled" data is a group of samples that have been tagged with one or more labels. Labeling typically takes a set of unlabeled data and augments each piece of it with informative tags. For example, a label might indicate whether a photo contains a horse or a cow, which words were uttered in an audio recording, what type of action is being

performed in a video, what the topic of a news article is, what the overall sentiment of a tweet is, or whether a dot in an X-ray is a cancer.

Labels can be obtained by asking humans to make judgments about a given piece of unlabeled data (e.g., "Does this photo contain a horse or a cow?"), and are significantly more expensive to obtain than the raw unlabeled data. After obtaining a labeled dataset, machine learning models can be applied to the data so that new unlabeled data can be presented to the model and a likely label can be guessed or predicted for that piece of unlabeled data.

For the plate detection model, the data will be labeled manually by defining a rectangular bounding box around every license plate in each image. For the digit recognition model, the data will be labeled by defining a rectangular box around each digit in each image and assign a corresponding label to each bounding box. Note that this process is very tedious and takes months complete which is why it must be done carefully and the progress must be kept in secure storage. Labeling data for a machine learning project presents the advantage of eliminating the need for data cleaning and complicated pre-processing since the data set can be built in which ever form is suitable for training. Figure 4.3 and fig. 4.4 illustrate the labeling tools used for both models.

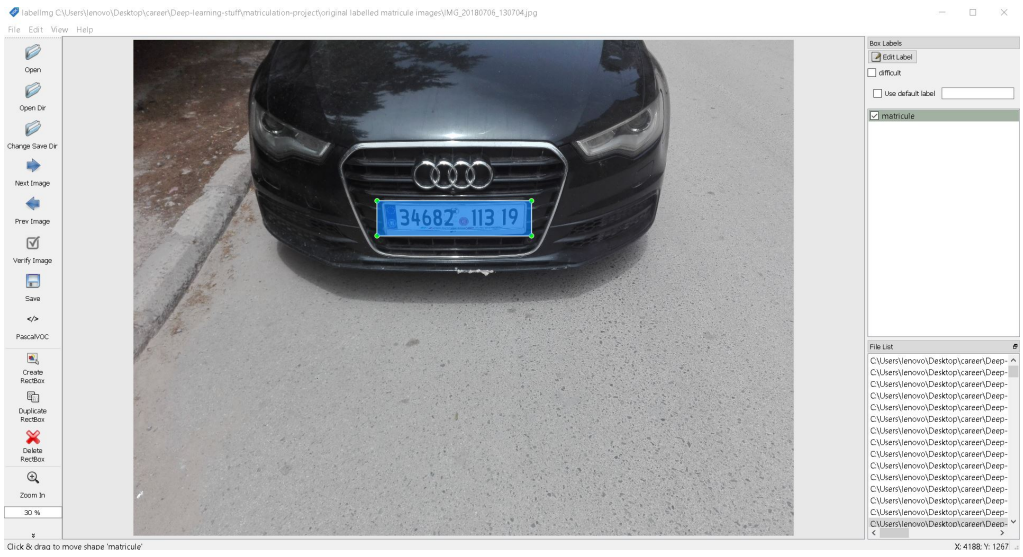


Figure 4.3: *Example of plate labeling.*

4.4 ALPR using Faster RCNN

Plate and digit detection requires training a set of models to detect and localize a license plate in any given image. The models trained belonged to two families of CNN models which are Faster RCNN and YOLOv3. In this section, we shall discuss the first family. For this family of models, there are three main parameters which can be adjusted in order

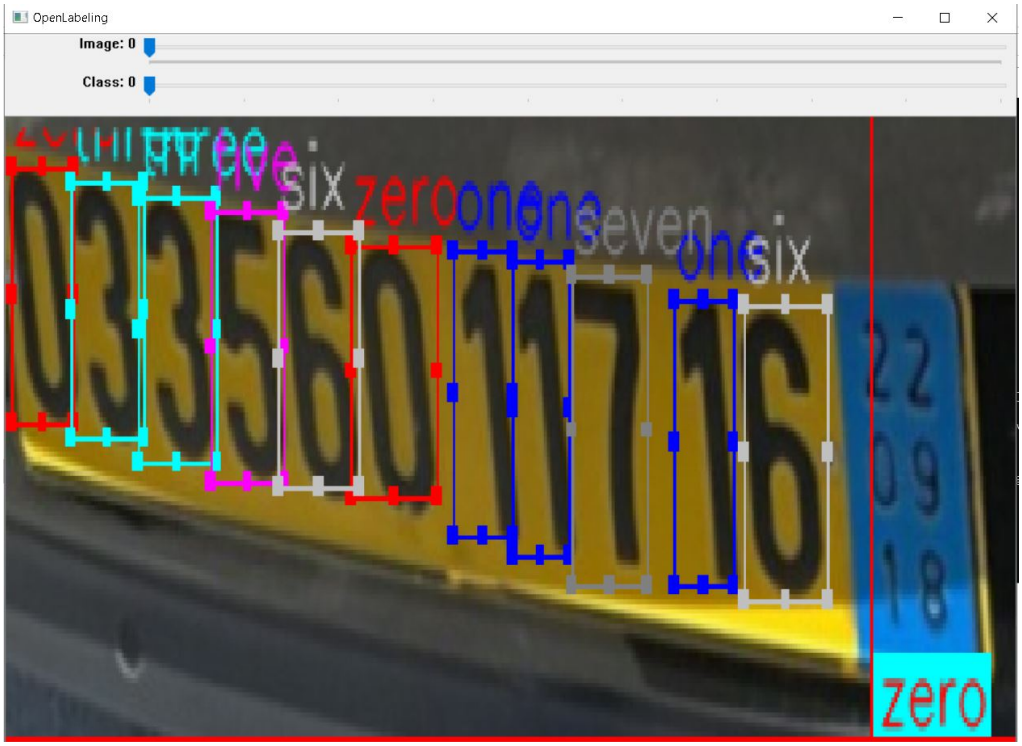


Figure 4.4: *Example of digit labeling.*

to obtain optimal results. The following parameters were chosen for both the digit and plate networks:

- Anchor scales
- Backbone network
- Number of training epochs

This choice is justified by the fact that the other parameters as specified by the original paper have been proven to be optimal in a number of previous works regardless of the application or the data set. The optimization algorithm used is the same for all instances of training. The anchor scales and ratios are :

- scales : (32, 64, 128), aspect ratios : $(0.5, 1.0, 2.0)$
- scales : (64, 128, 256), aspect ratios : $(1.0, 2.0, 4.0)$

Backbone networks are :

- VGG16
- Mobilenet
- Inception
- ResNet

Numbers of training epochs are :

- 10 epochs
- 30 epochs
- 50 epochs

4.4.1 Plate detection network

The list of parameters to be modified implies that the number of training processes launched is 24 each time with different parameters. The main coding tool is Pytorch library which is a Python package developed by Facebook. The platform used is Google Colab which is a free cloud service provided by Google, it provides Python programming environments equipped with all the tools and packages needed for building deep learning models, including Pytorch and GPU accelerators. Note that Pytorch is distinguished by its' object oriented approach to machine learning models. Whereas every model, design, or process is represented by a class.The process of training a Faster RCNN model using these tools includes the following steps :

- Implementing a data set class called "LicencePlateDataset".
- Implementing a backbone network class.
- Implementing a Faster RCNN model class.
- Implementing a trainer function.

4.4.1.1 LicencePlateDataset class implementation

The class "LicencePlateDataset" shown in fig. 4.5 inherits from the Pytorch class called "Dataset". The main function of this class is to instantiate objects which represent training examples. The *Dataset* class is characterized by the function "__getitem(index)__" which takes an index as an argument and returns a training example as an image along with its corresponding label. The function *__getitem(index)__* is overridden to match the specific format of data that the model requires. The *Dataset* class is also characterized by the function "process()" which access the data set and modifies it in a way that makes it accessible by the *__getitem(index)__* function.

The attribute "transforms" is an object from the "torchvision.transforms" class which contains specification for dimentionality and data type modifications applied on each training example. The size of the training images needs to be $500 \times 500 \times 3$ as shown in the original paper [19]. The data type of the image tensor (images are stacks of three 2D matrices representing the RGB channels) needs to be a "torch.Tensor", which is the data type required by Pytorch models. See fig. 4.6.

Running a unit test on the code is necessary to make sure that each part of our code is running correctly. The unit test for the LicencePlateDataset class is shown in fig. 4.7, along with the output. The test code creates an object from the class and prints out the label and the path of the image.

4.4.1.2 Backbone class implementation

The Backbone class instantiates backbone networks as objects. Objects from this class are characterized by the method "feed_forward(X)" which passes a given image X through the

```
class LicencePlateDataset(Dataset):


    def __init__(self , root_dir, transforms):
        self.root_dir = root_dir
        self.transforms = transforms
        self.train_set  = pd.read_csv(root_dir ,delimiter=",")
        self.ids = self._process()


    def _process(self):

        return hand

    def __getitem__(self,id):
        return img , label , img_path
```

Figure 4.5: *Skeleton code for LicencePlateDataset() class.*

```
transforms = Compose([Resize((500,500)) , ToTensor()])
```

Figure 4.6: *Code line for instantiating a "transforms" object.*

```
root_dir = "/content/drive/My Drive/plate detection/train_labels.csv"
transforms = Compose([Resize((500,500)) , ToTensor()])
make_ids  = LicencePlateDataset(root_dir , transforms)._process()
dataset= LicencePlateDataset(root_dir , transforms)
img , label , img_path   = dataset[10]
print(label)
print(img_path)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:27: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
{'labels': tensor([0]), 'boxes': tensor([[396., 166., 639., 251.]]))
/content/drive/My Drive/plate detection/train/107.jpg
```

Figure 4.7: *Test code for LicencePlateDataset() class.*

backbone network model and returns the output from the final convolutional layer. Objects from the same class are also characterized by the method “nn_base()” which defines the layers of the convolutional neural network. For each backbone network a separate class is implemented because they each have different architectures and different helper methods and special operations. fig. 4.8 shows the code of the method “nn_base()” for the simplest backbone network used which is VGG-16.

```
def nn_base(input_tensor=None, trainable=False):


    input_shape = (None, None, 3)

    if input_tensor is None:
        img_input = Input(shape=input_shape)
    else:
        if not K.is_keras_tensor(input_tensor):
            img_input = Input(tensor=input_tensor, shape=input_shape)
        else:
            img_input = input_tensor

    bn_axis = 3

    # Block 1
    x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv1')(img_input)
    x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv2')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)

    # Block 2
    x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv1')(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv2')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)

    # Block 3
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv1')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv2')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)

    # Block 4
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

    # Block 5
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv3')(x)
    # x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)

    return x
```

Figure 4.8: *Code for VGG-16 model.*

All the other backbone networks used are implemented in a similar fashion. The full code is available in a Google drive [7]. These backbone networks are designed initially with random weights and biases, but the Faster RCNN model loads up pre-trained weights and biases in order to use them for transfer learning.

4.4.1.3 Faster RCNN model class implementation

The "FasterRCNN" class defines the different parts of the Faster RCNN model which are demonstrated in fig. 3.7. The FasterRCNN class inherits from the class "GeneralizedRCNN" which is a built in class of Pytorch. GeneralizedRCNN provides some useful methods to models from the RCNN family. In addition to the FasterRCNN class, there are two additional classes implemented which are “TwoMLPHead” and “FastRCNNPredictor”. TwoMLPHead is used to create the classifier and regressor networks that come after the RoI pooling layer.fig. 4.9 summarizes the model shown in fig. 3.7.

```
class FasterRCNN(GeneralizedRCNN):

    out_channels = backbone.out_channels # Keeping track of the output size
                                         # of the backbone network.
    rpn = RegionProposalNetwork()        # initialize the rgion proposal network.
    box_roi_pool = MultiScaleRoIAlign()  # Initialize the roi pooling layer.
    rpn_head = RPNHead()                 # Keeping track of the RPN output.
    roi_heads = RoIHeads()               # Keeping track of ROI output.
    rpn_anchor_generator = AnchorGenerator() # Generate anchors.
    box_predictor = FastRCNNPredictor()     # Encapsulates the classifier and regressor
                                            # networks

class TwoMLPHead():
class FastRCNNPredictor():
```

Figure 4.9: *Skeleton code for the FasterRCNN() class.*

4.4.1.4 Trainer function implementation

The training function contains the training loop in which the training process will take place. The first network to train is the RPN. Afterwards, the regressor and classifier networks are trained on the output of the RoI layer output. The training function starts by feeding a training example through the network. A Pytorch built-in function calculate the gradients and optimizes the weights. Another Pytorch built-in function calculates the losses and the training process saves them in a list for plotting.

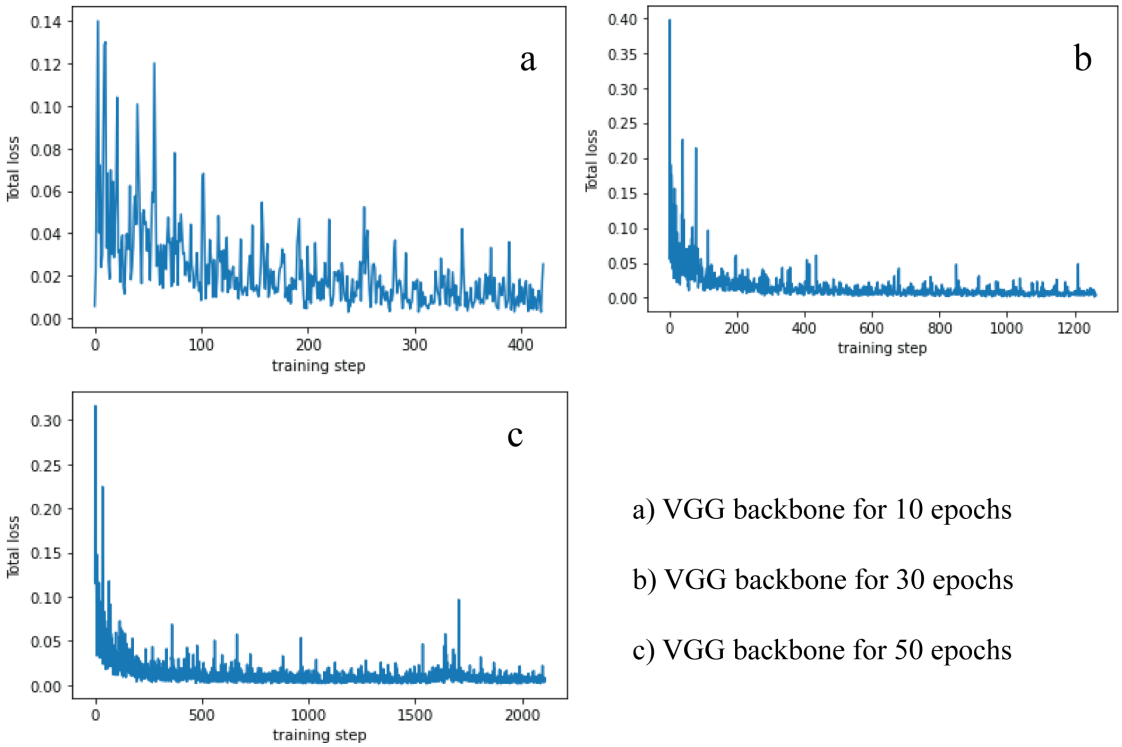
The value of the total loss for each training process was plotted with respect to training steps.

After these models are trained, they need to be tested on both the training and testing sets for analysis. The criterion chosen is the mAP(mean average precision, see **??**). It is a performance evaluation formula which takes into account the accuracy of the classification as well as the precision of the localization of objects. For each set of scales and aspect ratios the mAP on the training and test set were tabulated in table 4.1, table 4.2, table 4.3, table 4.4.

The time complexity of these models is recorded in number of seconds per frame. Keep in mind that the speed of the model does not depend on any hyper-parameter except

```
def train(train_loader , optimizer , n_epochs , device= 'cuda'):

    loss_over_time = []
    for epoch in range(n_epochs):
        index = 0
        for images , labels , _  in train_loader:

            model.train()

            labels, images = preprocess(labels, images , device = device)
            #compute loss
            loss_dict = model(images, labels)
            losses = sum(loss for loss in loss_dict.values())

            optimizer.zero_grad()
            #backbrop
            losses.backward()
            optimizer.step()
            #keeping track of the loss
            index+=1

            loss_over_time.append(losses.cpu().detach().numpy())
        print('EPOCH:{}/{} , image: {},losss: {:.4f}'.format(epoch , n_epochs ,index, losses))

    return loss_over_time
```

Figure 4.10: *Training function code.*



a) VGG backbone for 10 epochs

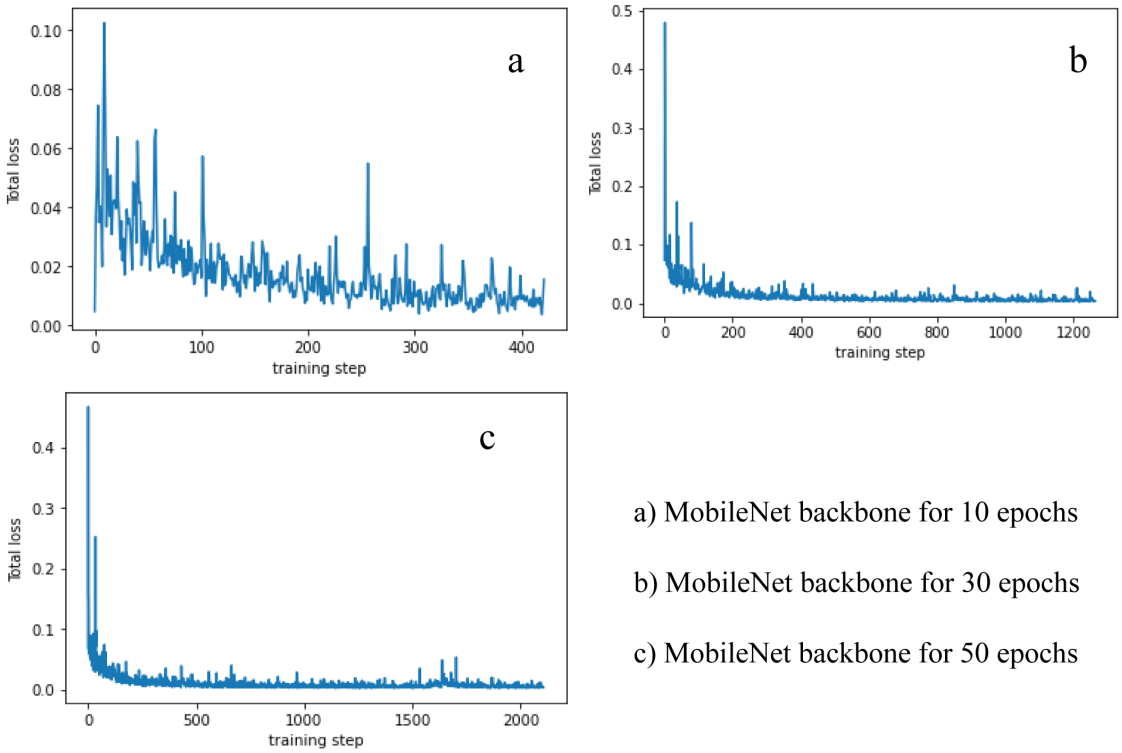
b) VGG backbone for 30 epochs

c) VGG backbone for 50 epochs

Figure 4.11: *VGG-16 for the first scales and aspect ratios.*

Table 4.1: *mAP for plate detection on training set for first set of scales and anchor ratios.*

Number of Epochs	VGG-16	Mobilenet	Inception	Res-Net
10 epochs	58%	65.1%	77.3%	78.4%
30 epochs	58.2%	65.1%	77%	78%
50 epochs	58%	65%	77%	78%

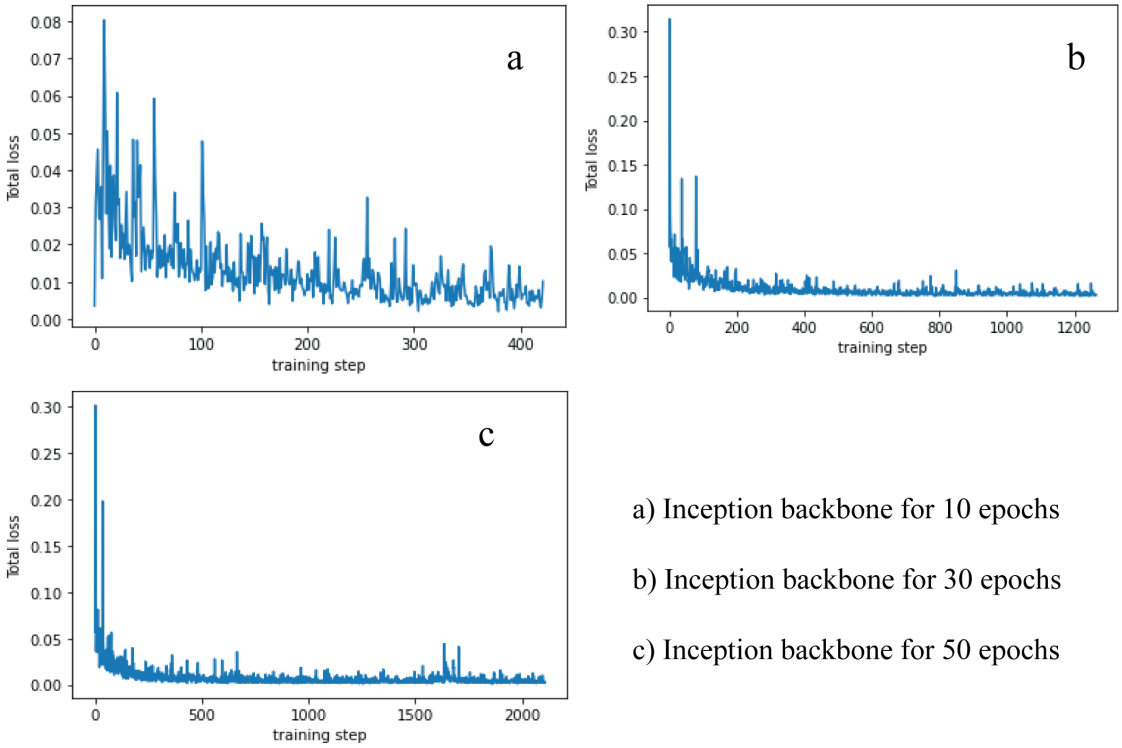


a) MobileNet backbone for 10 epochs

b) MobileNet backbone for 30 epochs

c) MobileNet backbone for 50 epochs

Figure 4.12: *mobilenet for the first scales and aspect ratios.*

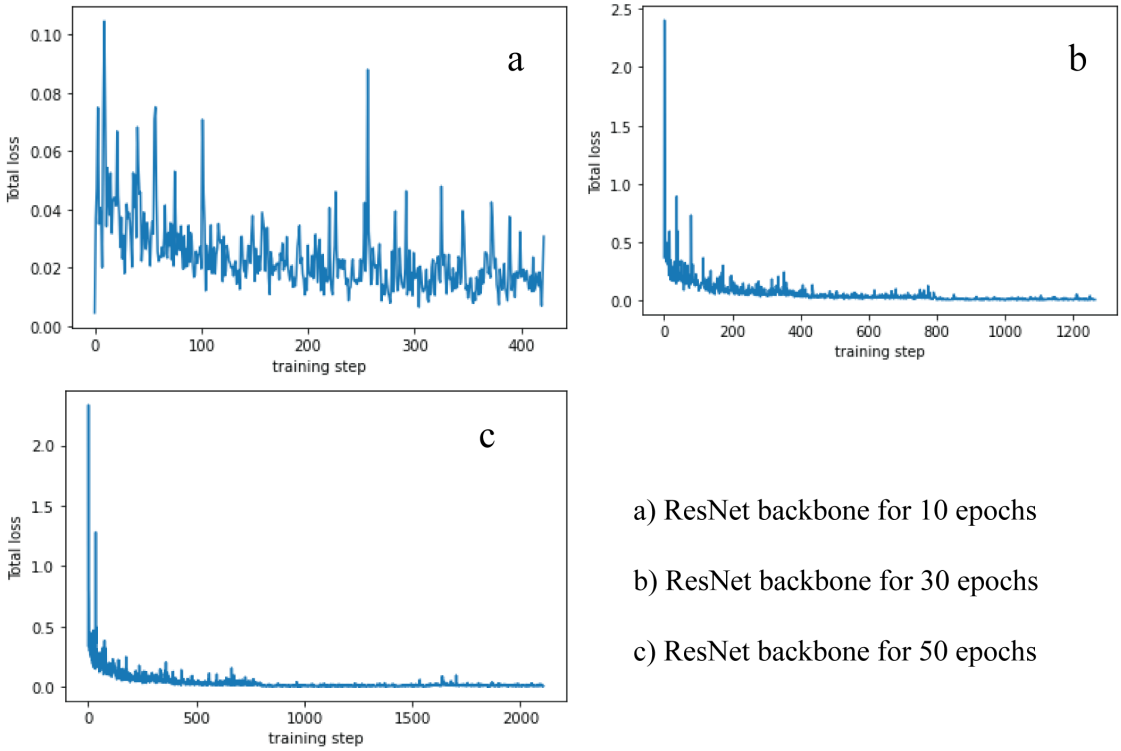


a) Inception backbone for 10 epochs

b) Inception backbone for 30 epochs

c) Inception backbone for 50 epochs

Figure 4.13: *inception for the first scales and aspect ratios.*

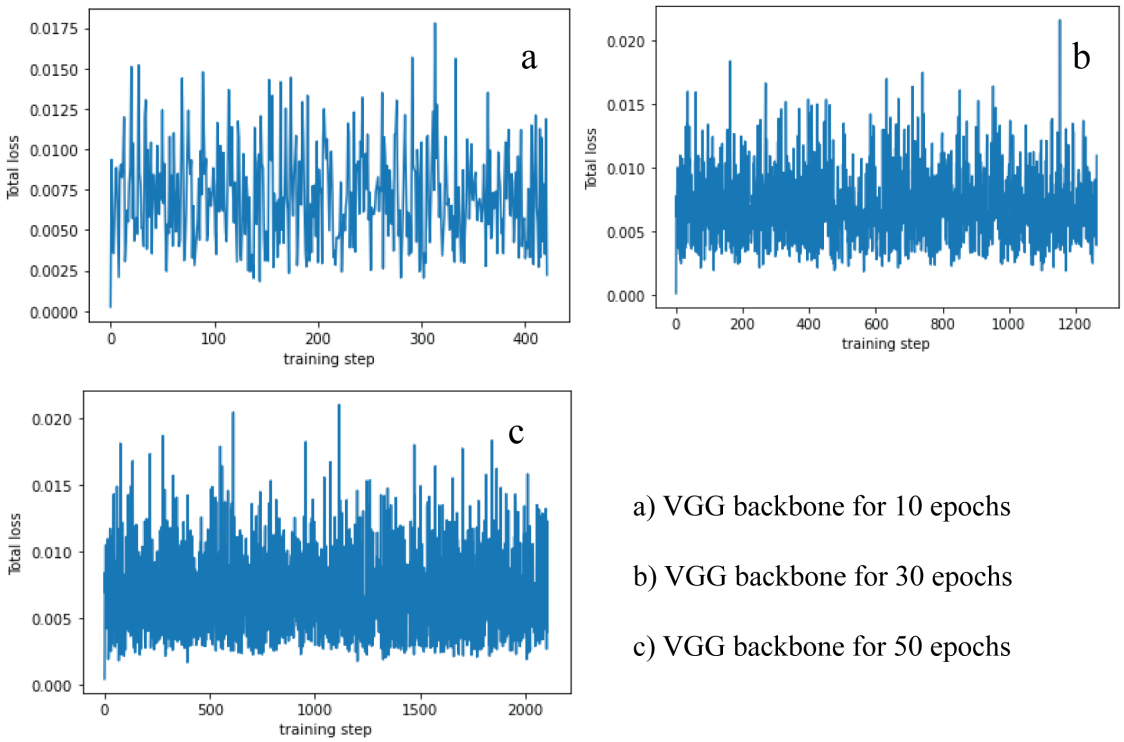


a) ResNet backbone for 10 epochs

b) ResNet backbone for 30 epochs

c) ResNet backbone for 50 epochs

Figure 4.14: *resnet for the first scales and aspect ratios.*

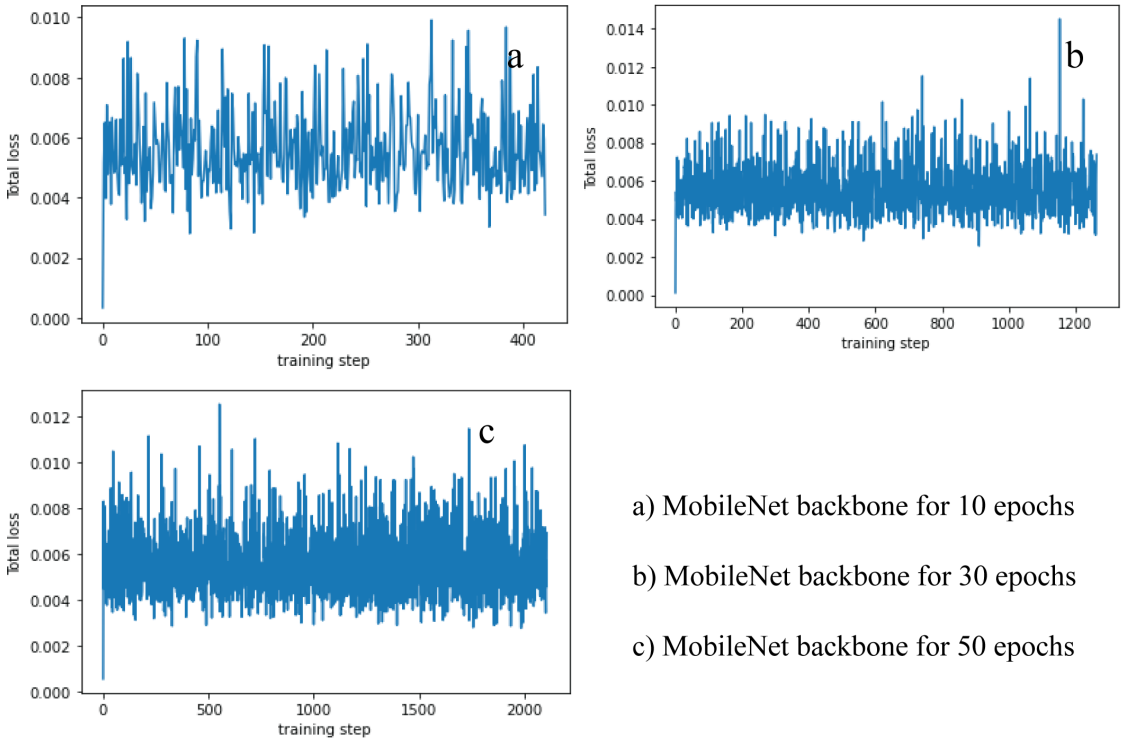


a) VGG backbone for 10 epochs

b) VGG backbone for 30 epochs

c) VGG backbone for 50 epochs

Figure 4.15: *VGG-16 for the second scales and aspect ratios.*

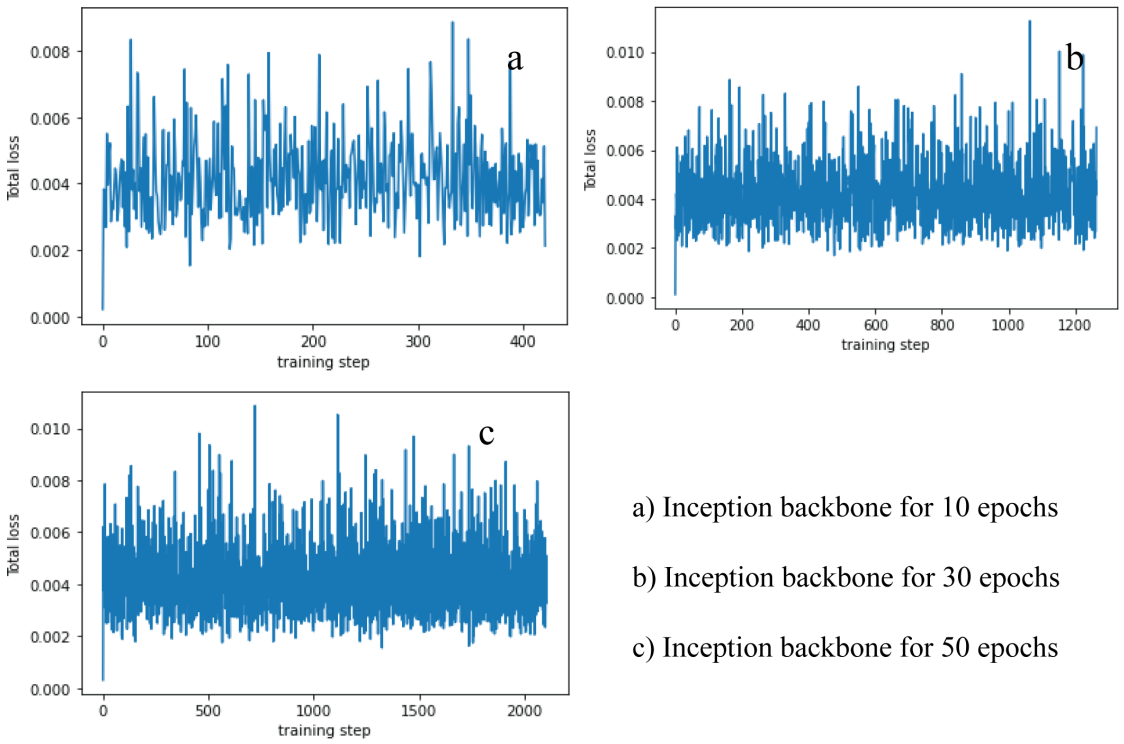


a) MobileNet backbone for 10 epochs

b) MobileNet backbone for 30 epochs

c) MobileNet backbone for 50 epochs

Figure 4.16: *mobilenet for the second scales and aspect ratios.*



a) Inception backbone for 10 epochs

b) Inception backbone for 30 epochs

c) Inception backbone for 50 epochs

Figure 4.17: *inception for the second scales and aspect ratios.*

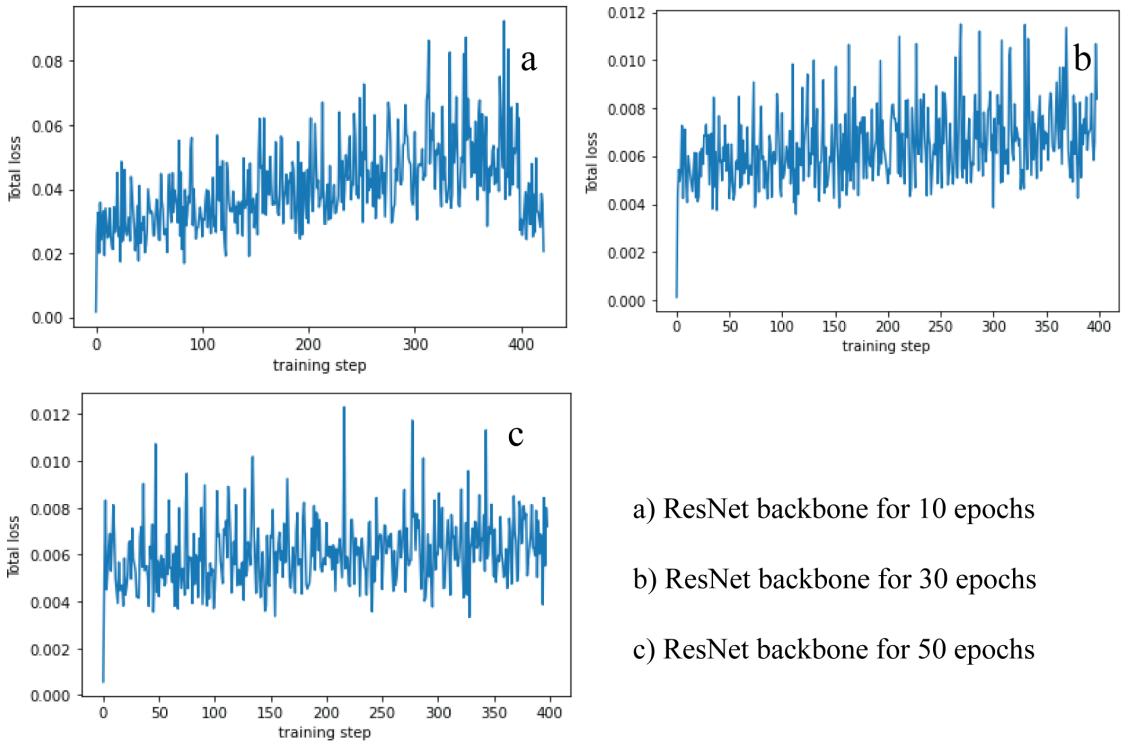


Figure 4.18: *resnet for the second scales and aspect ratios.*

Table 4.2: *mAP for plate detection on training set for second set of scales and anchor ratios.*

Number of Epochs	VGG-16	Mobilenet	Inception	Res-Net
10 epochs	27.5%	25.1%	27%	28.4%
30 epochs	27.5%	25.1%	27%	28%
50 epochs	27%	25%	27%	28%

Table 4.3: *mAP for plate detection on test set for first set of scales and anchor ratios.*

Number of Epochs	VGG-16	Mobilenet	Inception	Res-Net
10 epochs	57.5%	64.1%	75%	70.3%
30 epochs	57.2%	65%	75.1%	70%
50 epochs	57%	65%	75%	71%

Table 4.4: *mAP for plate detection on test set for second set of scales and anchor ratios.*

Number of Epochs	VGG-16	Mobilenet	Inception	Res-Net
10 epochs	28%	25.1%	27%	28.4%
30 epochs	28.2%	25.1%	27%	28%
50 epochs	28%	25%	27%	28%

for the size of the model itself and the input image size. The tests were run on a computer with an NVIDIA GPU model 930MX.

Table 4.5: *Number of seconds that each plate detection model takes to process one frame.*

VGG-16	Mobilenet	Inception	Res-Net
0.3 s	1.1 s	1.6 s	3 s

4.4.2 Digit recognition network

The same set of models were trained and the total loss graphs showed the same characteristics relevant for analysis as the ones plotted for plate detection. The same set of results were recorded for the digit recognition models.

Table 4.6: *mAP for digit recognition on training set for 1st set of scales and anchor ratios.*

Number of Epochs	VGG-16	Mobilenet	Inception	Res-Net
10 epochs	57.8%	64.9%	77.1%	77.4%
30 epochs	57.5%	64.9%	77.1%	77%
50 epochs	57.6%	64.8%	77%	77%

Table 4.7: *mAP for digit recognition on training set for second set of scales and anchor ratios.*

Number of Epochs	VGG-16	Mobilenet	Inception	Res-Net
10 epochs	25.5%	23.1%	25%	26.4%
30 epochs	25.5%	23.1%	25%	26%
50 epochs	25%	23%	25%	26%

Table 4.8: *mAP for digit recognition on test set for 1st set of scales and anchor ratios.*

Number of Epochs	VGG-16	Mobilenet	Inception	Res-Net
10 epochs	57%	64%	74.4%	70%
30 epochs	57%	65%	75%	69.9%
50 epochs	57%	65%	75%	69.9%

4.4.3 R-CNN ALPR system

The license plate recognition application is implemented using a Python script that takes an image as input and passes it through the plate detection model, then uses the output to

Table 4.9: *mAP results for digit recognition on test set for second set of scales and anchor ratios.*

Number of Epochs	VGG-16	Mobilenet	Inception	Res-Net
10 epochs	26%	23.1%	25%	26.4%
30 epochs	26.2%	23.1%	25%	26%
50 epochs	26%	23%	25%	26%

Table 4.10: *Number of seconds that each digit recognition model takes to process one frame.*

VGG-16	Mobilenet	Inception	Res-Net
0.3 s	1.1 s	1.6 s	3 s

crop the regions containing the license plates and passes them through the digit recognition model. Both models have an Inception network as backbone because it has the best mAP. This model is slightly more robust to errors that can be made by the plate detection model, because it checks if the digit recognition models' output contains a number of digits that is not consistent with reality. Note that Algerian license plates contain from 9 to 11 digits.

For the evaluation of this model, the only relevant characteristics is whether or not it reads the license plate number correctly. After testing on 100 test images containing 114 license plates, 96 license plates were read correctly therefore the accuracy is **84.21%**.

4.5 ALPR using YOLOv3

Using the other family, both the plate and digit networks are based on the YOLOv3 architecture (see section 4.5.1) with some modification and parameter choices to fit in with the new datasets. The first changed to the base architecture is done to fit the new number of output classes. The original implementation was built to predicted the 80 classes from the COCO dataset [25]. The plate and digit networks predict one and ten possible classes respectively. As discussed in section 3.1.1, the network divides the input image into an $S \times S$ grid, the value of S must be chosen carefully. Indeed, a small value would be great but then in a dataset with overlapping objects a greater value of B must be chosen and the network will struggle with small objects since tiny changes in the output value will result in the bounding box swinging a lot which makes it hard to localize. A large value of S is preferable since we want each object to fall into his grid cell to be detected alone. This way, a much smaller number of anchor boxes can be chosen because many objects are unlikely to fall into the same cell. But doing so, it will result in a large output volume which makes the training more difficult and very slow. To remedy this problems, the YOLOv3 writer were pretty genius. Instead of having one output, they put three. Each one

divides the input image into a different $S \times S$ grid. This depends on the input image shape. If the input image shape is 416×416, then the three outputs would result in a 13×13 grid at the first, a 26×26 at the second, and a 52×52 at the third output. This way, objects that have not been detected in one of the outputs are likely to be in the others. The authors also used a technique that takes a feature map from earlier in the network and merge it with upsampled features using concatenation. This method allows to get more meaningful semantic information from the upsampled feature maps and finer-grained information from earlier maps [25]. This technique helps a lot when dealing with small objects.

4.5.1 Training configuration

The convolutional network Darknet-53 (see appendix A) pre-trained on the ImageNet dataset is used as a backbone for both networks. The backbone represents the 52 first layers in the architecture (see table A.2). Since the network is huge, it is impractical to retrain; that would take ages to train due to low computational power available. The last FC layer of Darknet-53 is removed and replaced with seven additional convolutional layers, where the last one represents the first output. Next, the feature map from 2 previous layers is taken and upsampled (resized) by $2\times$. A feature map is also taken from earlier in the network and merged with the upsampled features using concatenation. This method allows us to get more meaningful semantic information from the upsampled features and finer-grained information from the earlier feature map. Then, Few more convolutional layers are added to process this combined feature map, and eventually predict a similar tensor, although now twice the size. The same design is performed one more time to predict boxes for the final scale. Thus our predictions for the 3rd scale benefit from all the prior computation as well as fine-grained features from early on in the network [25]. See fig. 4.19.

Both networks have been trained with 0.001 learning rate using the Adam optimizer with a momentum of 0.9. The plate network has been trained for 4500 iterations whereas the digits network for 8200 iterations. The number of iterations basically depends on the number of classes the network is training on. The more classes there are the more the number of iterations increases. This phenomenon is an empirical observation with no mathematical back up. Data augmentation has been used a lot through the training process. Specifically, for each 10 iterations, both networks randomly choose a new image dimension size, changes the saturation, the exposure and the hue of the images. This regime forces the networks to learn to predict well across a variety of input dimensions and gain in robustness. Both networks have been trained in the cloud using a free google service called colab. It offers 12 hours access to a virtual machine equipped with an NVIDIA Tesla K80 GPU.

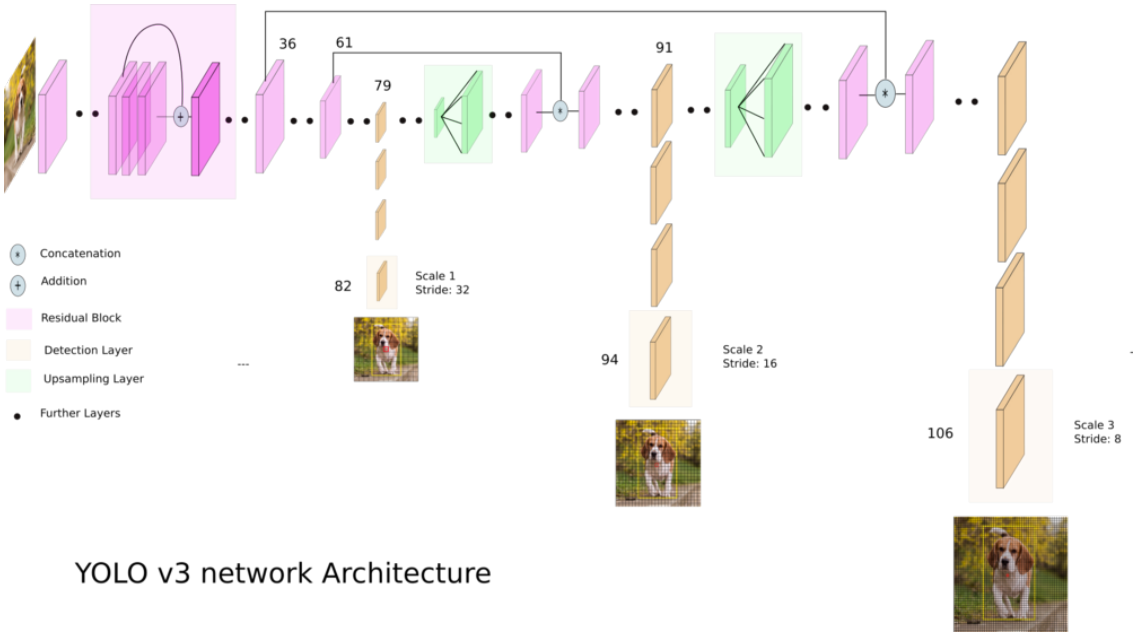


Figure 4.19: *YOLOv3 network architecture [2].*

4.5.2 Training process

The training process of both the plate and digit network has been realized using a deep learning framework called Darknet. Darknet has been developed by the YOLO paper's writers in order to train their own YOLO model. The framework is an open-source implementation hosted at `github.com`. Along with the framework, they provided detailed instructions for developers to train their own customized YOLO models. In order to use the framework, some specific files must be changed to fit the new developers model configuration. By configuration, we mean the number of classes that the model at hand is supposed to predict. Since the original implementation was built to predict 80 classes, the output volume predict by the network is $S \times S \times (3 * (5 + 80))$, whereas for the digit network for example, the number of classes to predict is 10, therefore the network must predict an $S \times S \times (3 * (5 + 10))$ output volume. The Darknet developers provide a ".cfg" file that contain the model configuration; all the layers of the network are list in this file. Namely, to indicate a convolutional layer with its input parameters, the syntax in fig. 4.20 has been followed.

where "size" represent the size of the filters applied (e.g. 3×3, 1×1, ...), "filters" represents the number of output filters constituting the output feature map, "stride" is the stride to use, "pad" is a boolean that indicates to either use padding or not. Finally, activation represents the activation function to use. In this case, only the leaky ReLU (defined by $f(x) = max(0.1x, x)$) and the linear activation functions are used. The majority of the layers are not to be changed since they represent the implementation of YOLOv3. The only layers to change are the layers designated by "yolo" and the convolutional layer just before it to fit our own custom network. See fig. 4.21.

```
[convolutional]
size=1
stride=1
pad=1
filters=45
activation=linear
```

Figure 4.20: *Convolutional layer representation in a cfg file.*

```
[convolutional]
size=1
stride=1
pad=1
filters=45
activation=linear


[yolo]
mask = 0,1,2
anchors = 10,13,  16,30,  33,23,  30,61,  62,45,  59,119,  116,90,  156,198,  373,326
classes=80
```

Figure 4.21: *Example of yolo layer in cfg file.*

The convolutional layer just before the *yolo* layer represents one of the three outputs of the network. It represents the output volume that the network is supposed to predict. Therefore, the filters parameter has to be changed in order to fit the new number of output classes. In our case, for the digit network for example, the output volume as discussed previously is $S \times S \times (3 * (5 + 10))$, meaning $S \times S \times 45$; the number of output filters outputted by this convolutional layer is therefore 45. This implies that the filter parameter should be set to 45 and this in all three output layers of the network. The *yolo* layer in the other hand contains the following parameters:

- **anchors**, represent the initial hand-picked anchor boxes to be adjusted during training. Each pair represents the width and the height of the anchor box respectively.
- **mask**, is a boolean array representing the set of anchor boxes to use at each output. In fig. 4.21 only the 6^{th}, 7^{th} and 8^{th} boxes are taken into consideration (indexing starts at 0).
- **classes**, represent the number of output classes. In our example, ten.

Another files that has to changed is “obj.data” file. Figure 4.22 depicts a snippet of this file.

```
classes=10
train=data/train.txt
names=data/obj.names
backup=/mydrive/yolo/backup
```

Figure 4.22: *obj.data file snippet.*

The first parameter is obvious. The “names” parameter points to a file location containing the names of the output classes. In our case, it is just the digits from 0 to 9, and in the case of the plate network it is just *plate*. The “backup” parameter is a pointer to a empty directory where the weights of the network will be saved into a binary file format. Finally, the “train” parameter points to a directory containing the training set. The training set is composed of images along side *.txt* files representing the label of a particular image. Figure 4.23 shows an image label example in a .txt file format.

```
0 0.036000 0.283465 0.061333 0.409449
0 0.384000 0.405512 0.085333 0.401575
1 0.472000 0.440945 0.058667 0.456693
1 0.529333 0.464567 0.056000 0.472441
1 0.690667 0.519685 0.058667 0.456693
3 0.100000 0.307087 0.072000 0.409449
3 0.169333 0.338583 0.077333 0.425197
5 0.238667 0.366142 0.072000 0.433071
6 0.309333 0.385827 0.080000 0.409449
6 0.772000 0.535433 0.082667 0.472441
7 0.597333 0.500000 0.074667 0.496063
```

Figure 4.23: *Example of YOLO image label convention.*

each line in this file represents a bounding box manually set using the tool in fig. 4.4. There are 5 numbers in each line separated with spaces. The first number represents the class of the corresponding object (e.g. 0 for digit 0 in a license plate). The next four numbers represent the bounding box coordinates with respect to the image, where the first two numbers translates to the center position of the box and the two last numbers translates to the width and height of the box respectively.

After downloading the Darknet framework into the colab environment, compiling it, and configuring all the necessary files, the training can starts. The training is launched using the command ./darknet train cfg file.

4.5.3 Inference process

The goal of training a network is to use it later on to make predictions on unseen data, this is referred to as **inference**. The Darknet framework cannot be used for that purpose since all the code it provides is hidden and the only way to access the different functions is through command lines, which is not very useful when it comes to build customized application. For this reason, the Darknet framework is only used to get the weights of the networks. For inference purposes, one can use the weights saved into binary format file and load them into a YOLOv3 network created using another deep learning framework. Frameworks like Pytorch give access to very basic building blocks of neural networks such as FC layers and convolutional layers to build custom networks from the ground up unlike Darknet. Pytorch offers way more flexibility and customization options, with it being easy to use. Therefore the following strategy has been adapted to make use of the weights obtained from the training process and make predictions:

- Rebuild YOLOv3 network using Pytorch.
- Creating a function called “predict_transform()” to interpret the YOLOv3 output.
- Creating a function called “load_weights()” to load the weights obtained form training into the Pytorch Modules.

4.5.3.1 Rebuilding YOLOv3 network

In order to achieve this, we make use of the cfg file. In fig. 4.24, the function “parse_cfg()” takes in a cfg file location as parameter, reads the file and split it into a python list at each end of line character, gets rid of the empty lines and comments, as well as stripping each line. The idea to rebuild the network is to go through each line and store every layer as a dictionary. The parameters of each layer are stored as key-value pairs in that same dictionary. As we parse through the cfg file, we keep appending these dictionaries denoted by the variable *block* to a list *blocks*. After that the returned layers list is fed as parameter to a function called “create_modules” to construct the Pytorch modules corresponding to the different layers in the cfg file.

```
def parse_cfg(cfg_file):
    with open(cfg_file, 'r') as f:
        lines = f.read().split('\n')
        lines = [line for line in lines if len(line) > 0]
        lines = [line for line in lines if line[0] != '#']
        lines = [line.strip() for line in lines]

    block = {}
    blocks = []

    for line in lines:
        if line[0] == '[':
            if len(block) != 0:
                blocks.append(block)
                block = {}
            block['type'] = line[1:-1].strip()
        else:
            key, value = line.split('=')
            block[key.strip()] = value.strip()
    blocks.append(block)

    return blocks

def create_modules(blocks):
```

Figure 4.24: *parse_cfg function implementation.*

4.5.3.2 Creating predict_transform function

Next, we implement the forward pass of the network which is typically done by overriding the "forward()" function of the "nn.Module" Pytorch class after inheriting from it. *forward* serves two purposes: calculate the outputs and transform the output feature map using eq. (3.1). The output of YOLO is a convolutional feature map that contains the bounding box attributes along the depth of the feature map. The attributes of bounding boxes predicted by a cell are stacked one by one along each other. This form is very inconvenient for output processing such as thresholding by a confidence score, adding grid offsets to centers, applying anchors etc. Another problem is that since detections happen at three scales, the dimensions of the prediction maps will be different. Although the dimensions of the three feature maps are different, the output processing operations to be done on them are similar. It would be nice to have to do these operations on a single tensor, rather than three separate tensors.

To remedy these problems, we make use of a function called predict_transform that takes in a detection feature map and turns it into a 2D tensor, where each row of the tensor corresponds to attributes of a bounding box, in the order depicted in fig. 4.25 , and the code to do the above operation is illustrated in fig. 4.26.

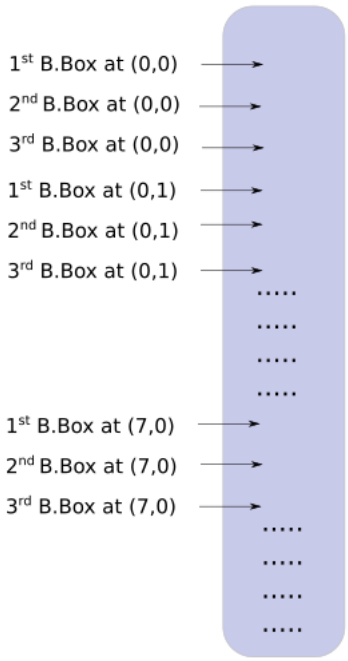


Figure 4.25: *2D tensor representation of the output feature map.*

then eq. (3.1) is applied to each bounding box through the rest of the code in **??**. Three feature maps are passed into this function which transforms them into the mentioned form. These feature maps are then concatenated into one long tensor.

After that, non-max suppression is applied to the output prediction which contains all the bounding boxes of the three outputs. The algorithm has already been explained

```
def predict_transform(prediction, inp_dim, anchors, num_classes, CUDA = True):
    batch_size = prediction.size(0)
    stride =  inp_dim // prediction.size(2)
    grid_size = prediction.size(2)
    bbox_attrs = 5 + num_classes
    num_anchors = len(anchors)

    prediction = prediction.view(batch_size, bbox_attrs*num_anchors, grid_size*grid_size)
    prediction = prediction.transpose(1,2).contiguous()
    prediction = prediction.view(batch_size, grid_size*grid_size*num_anchors, bbox_attrs)
```

Figure 4.26: *2D tensor transformation implementation.*

in chapter 3. It's implementation rely on the function "torch.ops.nms" which return a set of indices corresponding to the bounding boxes selected by the operation. Then boolean masking is applied to the 2D tensor representation of the output to extract the boxes indexed by the non-max suppression.

4.5.3.3 Loading weights to the Pytorch modules

The weights are stored in binary files in a serial fashion. Extreme care must be taken when reading the weights as they are just stored as floats one next to the other without guide or separation to indicate to which layer these set of weights belong to. In order to read the weight file, we implemented a function called "load_weights" which takes the weights file as a parameter. First, the file is read using built-in python function "open()". Then, to read specific parts of the file we use the "fromfile()" function from the famous *NumPy* library. This function takes the file read using *open()*, the data type of the data to extract and the number of data we want to extract. For example, the first 160 bytes of the weights file store 5 int32 values which constitute the header of the file. They can be extracted using *fromfile()* as shown in fig. 4.28 [].

The only layers that contain weights in the YOLOv3 network are the convolutional layers. The biases are stored first then the filters as in fig. 4.29 [**?**].

Then we loop through the modules return by the function create_modules(), and with the fromfile() function and the class methods of the different Pytorch modules we extract the weights and load them to the different modules. All the code used in this project related to YOLOv3 can be found at `https://github.com/netvor-73/Lpd` along with the weights and corresponding cfg files.

After successfully training the networks, loading the weights into the Pytorch implementation, we tested our results on different test sets and the results have been recorded below.

```
prediction[:,:,:2] = torch.sigmoid(prediction[:,:,:2])
prediction[:,:,4] = torch.sigmoid(prediction[:,:,4])

#Add the center offsets
grid_len = np.arange(grid_size)
a,b = np.meshgrid(grid_len, grid_len)

x_offset = torch.FloatTensor(a).view(-1,1)
y_offset = torch.FloatTensor(b).view(-1,1)

if CUDA:
    x_offset = x_offset.cuda()
    y_offset = y_offset.cuda()

x_y_offset = torch.cat((x_offset, y_offset), 1)
                    .repeat(1,num_anchors)
                    .view(-1,2).unsqueeze(0)

prediction[:,:,:2] += x_y_offset

#log space transform height and the width
anchors = torch.FloatTensor(anchors)

if CUDA:
    anchors = anchors.cuda()

anchors = anchors.repeat(grid_size*grid_size, 1).unsqueeze(0)
prediction[:,:,2:4] = torch.exp(prediction[:,:,2:4])*anchors

#Softmax the class scores
prediction[:,:,5:] = torch.sigmoid((prediction[:,:, 5:]))

prediction[:,:,:2] *= stride

return prediction
```

Figure 4.27: *Bounding box calculation implementation.*

```
header = np.fromfile(fp, dtype = np.int32, count = 5)
```

Figure 4.28: *The fromfile function in action.*

Figure 4.29: *Biases and filters storage fashion in a binary file [**?**].*

4.5.4 Testing results

4.5.4.1 Plate network

The plate network has been trained on 630 manually labeled images. It has been trained for 1400 iterations first where the loss went down to 0.123 with an mAP of 97.0% (see fig. 4.30) on the test set where 270 images have been used. But after inspection, we find out that the model struggled with the case of multiple plates in the same image and some fuzzy images. We restarted training for another 2500 iterations from the same point in the hope to get the loss under 0.03, but unfortunately, the training took too long without any noticeable results, therefore we stopped it at 0.052 loss with mAP of 97.4% (see fig. 4.31). In fact, these are pretty good results, and further improvements can be made with more training data and hopefully our initial problems where solved. Table 4.11 summarizes the different results mentioned above.

Table 4.11: *Plate network results using YOLOv3 model.*

Number of Epochs	Loss	mAP
1400 epochs	0.123	97.0%
4000 epochs	0.052	97.4%

Figure 4.32 shows some examples of plate detection using our network. The problems encountered here are probably due to the shape and the color of the plates which are not usual; they don't appears often in the training dataset.

4.5.4.2 Digit network

The digit network has been trained on 845 images containing 10,402 hand annotated digits. As the plate network, it has been trained for 4000 iterations at the beginning which brought the loss to 0.8205 with an mAP of 64.2% on a test set containing 145 images with 1453 annotations. The network took so long to get there, but unfortunately doing horribly on the test set. We restarted training for another 4000 iterations and finally got the network to 0.567 loss with a 65.1% mAP on the test set (The plots of the plate and digit network are very similar, therefore there is no need to show them). Table 4.12 summarizes the results mentioned above.

Figure 4.33 shows some examples of digit detection using our network. Our model struggles a lot with diagonal images due to the lack of examples.

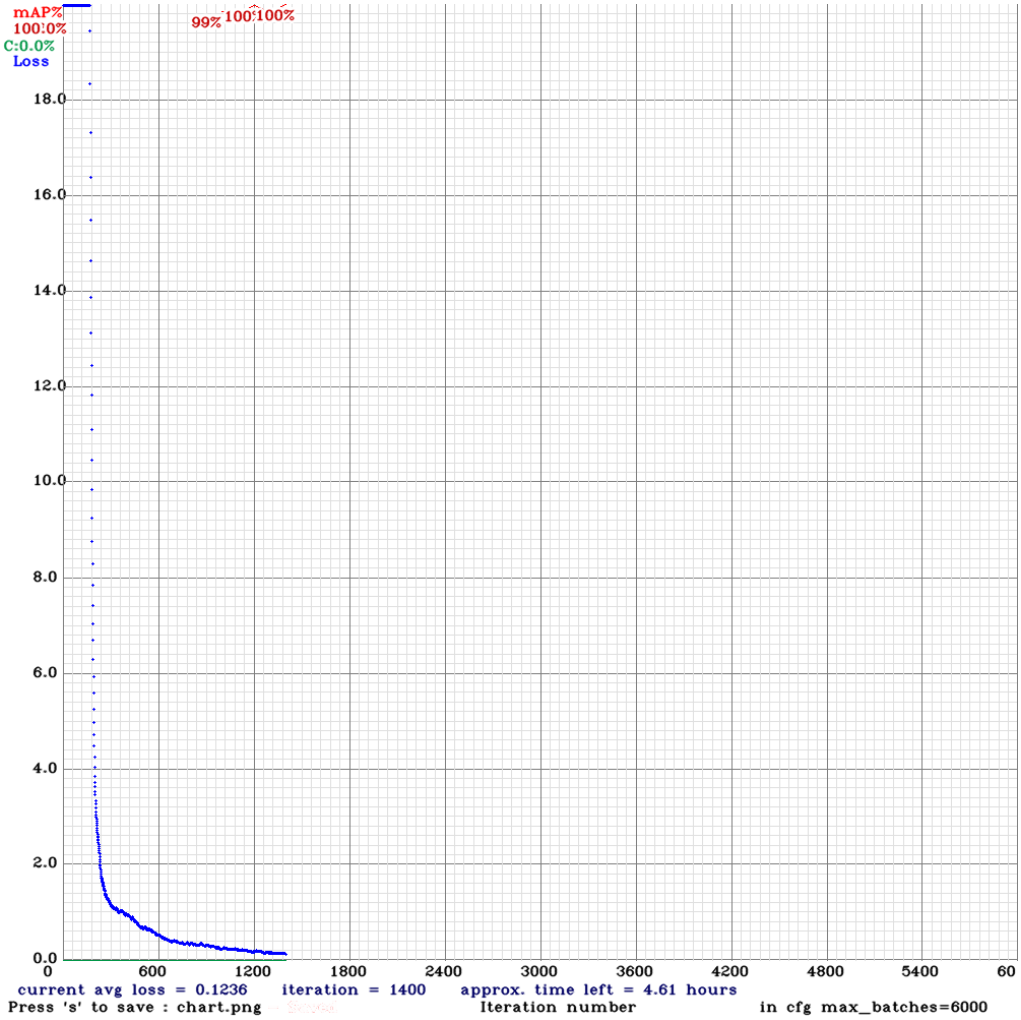


Figure 4.30: *Loss plot for the plate network after 1400 iterations.*

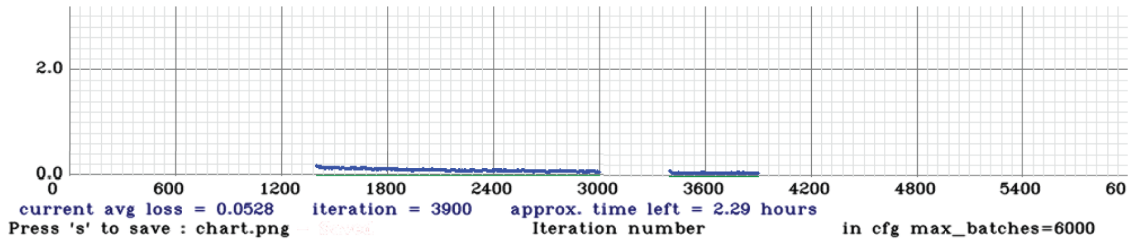


Figure 4.31: *Loss plot for the plate network after 4000 iterations. The erased blue portion was due to unstable internet connection*

Table 4.12: *Digit network results using YOLOv3 model.*

Number of Epochs	Loss	mAP
4000 epochs	0.8205	64.2%
8000 epochs	0.567	65.1%

Figure 4.32: *Plate detection examples. a) Successful detection under challenging conditions b)Complete failure.*

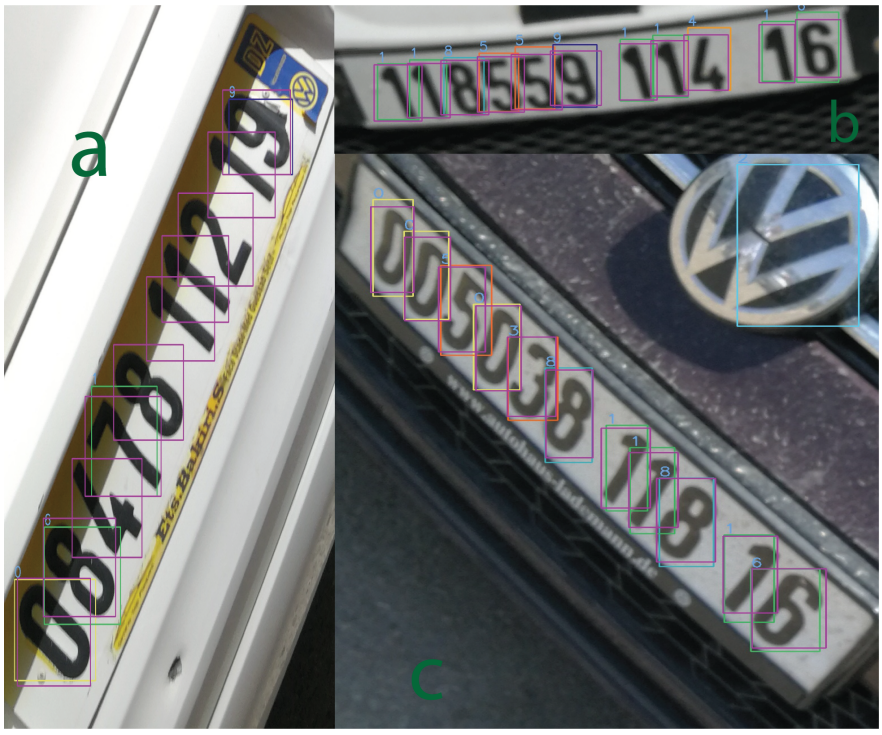


Figure 4.33: *Digit detection examples. a) Complete failure. b) Successful detection in blurry images. c) Detection of a false negative in the top right corner.*

4.5.5 YOLO ALPR speed test

The networks have been tested in the cloud on an NVIDIA Tesla K80 GPU. The speed of networks mainly depends on the input image size; the bigger the image the slower the processing time. Our networks both resize the input images to 416 pixels by 416 pixels keeping their aspect ratios. All speed tests performed in this project disregard the time it takes to save images to disk and load the networks into memory. Only the processing time used by the networks is measured, see table 4.13.

Table 4.13: *Speed test summary.*

Network	Average Time
Plate network	26.7 ms
Digit network	26.9 ms
Entire system	52.6 ms

The final system using YOLOv3 has been tested on 145 images, 102 were correctly identified, yielding an accuracy of **70.3%**.

Here are some examples of plate and digit detection by the final system.

The research goal of this project was to propose and develop an ALPR system using deep learning techniques. Given the wide variety of these techniques, two of them have been selected due to their popularity in the literature in a variety of other tasks. The exact requirements for an ALPR system depend on the application, but generally the requirements are an almost human-level prediction accuracy and a decent processing speed for real-time video streams.

4.6 RCNN result analysis

4.6.1 Total loss graphs

At first glance at the graphs in section 4.4, it appears that the first set of scales and aspect ratios for the RPN anchors produce a cost function that converges towards a certain minimum. Whereas the second set of scales and aspect ratios produce a cost function that is seemingly divergent, and keeps oscillating around the initial cost value. This is due to the fact that the first set of scales and aspect ratios produce anchors which are similar in shape and scale to those in both training data sets. Using the second set was an attempt to find a better suited one for the tasks at hand, but it seems like there needs to be a grid-search[32] operation in order to possibly find one, which requires more advanced hardware resources and more time. The first set of scales and aspect ratios was used by the authors of the original paper and obtained the best results on very large and diverse data sets of common objects such as COCO[35], PASCAL[18], and Image-NET[13].

Figure 4.34: *Examples of ALPR detection. 1st quadrant) Successful detection under normal conditions. 2nd quadrant) Successful detection under challenging conditions. 3rd quadrant) Missing digits under extreme conditions. 4th quadrant) Misclassification of foreign plate (this is normal since the system wasn't design for that purpose).*

The second characteristic to notice is the fact that less complex models start at a higher loss value but converge towards a minimum faster than more complex ones. This confirms that models with less layers and less parameters per layer train faster than ones with more layers and more parameters. This is explained by the fact that the data sets at hand are too small to influence very deep models like Res-Net.This phenomenon is called "Over-parameterization" [28], where a machine learning model contains too much parameters to train on the data set at hand. This also explains why the Res-Net model converges towards a minimum cost which is higher than that of the less complex models.

4.6.2 mAP tables

The mAP tables in section 4.4 shows that the models trained using the first set of anchors and aspect ratios perform far better than the ones using the second set of scales and aspect ratios, which is an obvious result based on the total loss curves. The best result was obtained by the model using Inception network as a backbone which is 75% and 74.4% for test sets of plate detection and digit recognition respectively. For Res-Net, the performance seems to have dropped, which is explained by the fact that Res-Net needs more data to learn the specific features of the objects. The difference between the performance on the training set and testing set is expectable since the test set is data which the model has never seen before, but what is remarkable is that this difference is bigger for Res-Net backbone models than it is for the other models. This indicates that the Res-Net backbone model has "Over-fitted"[32] to the training set. Since the other models did not over-fit, it is only reasonable to deduce that the overfitting was due to the higher complexity of the Res-Net based models, which tends to happen with highly complex machine learning models.

4.7 YOLOv3 result analysis

4.7.1 Total loss graphs

We clearly observe in the graphs, that the loss function quickly converges towards a certain minimum, in contrast to the RCNN with the second set of anchors. This is due to the design of the YOLOv3 network. Indeed, it is a single end-to-end object detection neural network, which by definition is designed to minimize a loss function. The advantage of YOLOv3 is, it would still work even without the introduction of anchor boxes but the benefit of that is the stabilization of training, which makes it converge significantly faster. Therefore YOLO does not suffer from the selection of anchors.

4.7.2 mAP tables

The mAPs results speak for them selves. The plate network is very good at detecting plates, even better than the RCNN model and way faster. It achieves around 27 ms per image processed, equivalent to about 37 FPS (frames per second) when running on video streams. This is within the requirements to run on real-time video streams without dropping frames. To achieve such high processing speeds, a relatively modern GPU is required. This is not problematic for real-world applications running on desktop; however for applications running on embedded devices, another lighter architecture should be consider which would drop mAP significantly. The digit network in the other hand is achieving real-time video processing speeds as well, but with a relatively low mAP which is way far from human performance; this caused the overall accuracy of the system to dramatically drop. It is mainly due to the unbalance within the dataset. Although the number of training examples is relatively high, some digits are under represented which is problematic. Therefore, a more balanced dataset would fix the problem. This can be achieved providing more data.

4.8 Final application

Both proposed methods in this work achieved descent accuracies as well as enough processing speeds. But as it has been demonstrated, the RCNN method despite of its high accuracy (84.21%) suffers from relatively low processing speeds. On the other side, disregarding its relatively low accuracy (70.3%), YOLO achieves real times video processing capabilities. Therefore, in the aim to compensate for each other weaknesses, a hybrid model between both methods seems the way to go. Indeed, the YOLO plate detector has been proven to be really efficient in terms of accuracy and speed, whereas the RCNN digit network has proven its robustness to extremely challenging conditions. Given these two pieces of information, these last networks have been combined together in a final Python pipeline implementing the describe flow diagram illustrated in fig. 4.1. The hybrid method runs on real-time video streams, with an accuracy of **81.36%** which almost meets human performance.

The proposed method is very general. It could easily adapt to many real-life scenarios without changing the dataset. It has been proven to work in very harsh conditions. Indeed, with a relatively controlled environment; good camera placement and descent lighting conditions the method would achieve super human-performance quite easily. In addition, the method could, as well, be adapted to different datasets without major changes. The system is extremely versatile given that it does not use any hand-crafted features but rather learn them which is quite powerful.

The accuracy of 81.36% obtained in the application testing is considered a good result for a first attempt. Although there are many ways to improve this result by working on:

- Building a bigger data set.
- Using more modern techniques and deep learning models.
- Encoding the models as C++ data structures to improve inference speed.
- Using CUDA programming language to optimize computations on GPU [14].
- Using unsupervised learning techniques to make sure that the model keeps learning from its' mistakes even after the application deployment.

There are other ways to correct the short-comings of the application without any work being done on the model. For instance, the application can provide the model with many frames of the same car, thereby making sure that if the model makes a mistake in one frame, it can correct it in another. The accuracy can also be boosted, in a human-assisted environment with the addition of a warning system. This can be easily achieved by thresholding the number of digits detected; whenever this number is less than a certain threshold, the system warns the human-assistance of a wrongly detected plate.

Conclusion

In conclusion it is reasonable to say that the Faster-RCNN and YOLO models in particular and CNNs in general are a highly effective method in addressing automation and computer vision challenges such as license plate recognition. The satisfying results obtained by both methods using relatively outdated hardware resources and methods are evidence of that. These results would also open the gate for further research and innovation on this particular application.

To recap, this work was an attempt to create a practical application to be used for Algerian license plate detection and recognition. First, the data set was built from the ground up, including both data collection and data labeling processes. Second, the Faster-RCNN and YOLO models were selected as main tools for this task. Afterwards, a series of different models with different structures and parameters were trained and analyzed in order to explore the effects of these parameters on the outcomes of the model, and to figure out the best methods in the building of the application. These models were tested on new real world data and have shown accuracies of 84.21% and 70.3% for Faster-RCNN and YOLO respectively. Finally, in the strive to achieve real world applications requirements, both methods have been combined.

In the future, this project can be improved by working on many aspects of development such as the ones mentioned in the last chapter. And can be refined to be a useful tool in the hands of different institutions, businesses, and even law enforcement or security organizations. As in fact, this work is being optimized by the authors and integrated into a real world application used for monitoring garbage trucks under the supervision of a tech start-up in Algiers called Brainiac. A direct quote form a Co-Founder of this company states the following : "Introducing similar technologies into the Algerian market is not only a savvy business idea, but also a great opportunity for researchers and young software developers to turn their innovations into real world applications". And finally, it would also be insightful to mention the following quote from Ray Kurzweil - American inventor and futurist - : “Artificial intelligence will reach human levels by around 2029. Follow that out further to, say, 2045, and we will have multiplied the intelligence a billion-fold.”

Appendix A

YOLOv3 network architecture

A.1 Feature extractor

YOLO don't use any of the already pre-trained backbone networks on image classification. Instead they trained their own classifier on the ImageNet data-set. The network uses successive 3×3 and 1×1 convolutional layers with some shortcut (skip) connections. It has 53 convolutional layers, therefore it has been named Darknet-53. Darknet is the deep learning framework used to train it. See table A.2

Darknet-53 network is pretty good one compared to other backbones that are even deeper than that which makes it way faster at inference time. See table A.1. Each network is trained with identical settings and tested at 256×256, single crop accuracy. Run times are measured on a Titan X at 256×256. Thus Darknet-53 performs on par with state-of-the-art classifiers but with fewer floating point operations and more speed. Darknet-53 is better than ResNet-101 and $1.5\times$ faster. Darknet-53 has similar performance to ResNet-152 and is $2\times$ faster. Darknet-53 also achieves the highest measured floating point operations per second. This means the network structure better utilizes the GPU, making it more efficient to evaluate and thus faster. That's mostly because ResNets have just way too many layers and aren't very efficient [25].

Table A.1: *Comparison of backbones. Accuracy, billions of operations, billion floating point operations per second, and FPS for various networks [25].*

Backbone	**Top-1**	**Top-5**	**Bn Ops**	**BFLOP/s**	FPS
ResNet-101	77.1	93.7	19.7	1039	53
ResNet-152	**77.6**	**93.8**	29.4	1090	37
Darknet-53	77.2	**93.8**	18.7	**1457**	**78**

Table A.2: *Darknet-53 [25].*

	Type	Filters	Size	Output
	Convolutional	32	3×3	256×256
	Convolutional	64	3×3 /2	128×128
	Convolutional	32	1×1	
$1 \times$	Convolutional	64	3×3	
	Residual			128×128
	Convolutional	128	3×3 /2	64×64
	Convolutional	64	1×1	
$2 \times$	Convolutional	128	3×3	
	Residual			64×64
	Convolutional	256	3×3 /2	32×32
	Convolutional	128	1×1	
$8 \times$	Convolutional	256	3×3	
	Residual			32×32
	Convolutional	512	3×3 /2	16×16
	Convolutional	256	1×1	
$8 \times$	Convolutional	512	3×3	
	Residual			16×16
	Convolutional	1024	3×3 /2	8×8
	Convolutional	512	1×1	
$4 \times$	Convolutional	1024	3×3	
	Residual			8×8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Appendix B

mAP (mean Average Precision) for Object Detection

AP (Average precision) is a popular metric in measuring the accuracy of object detectors like Faster R-CNN, SSD, etc. To understand it, its better to recap some related concepts.

B.1 Precision and recall

Precision measures how accurate is your predictions. i.e. the percentage of your predictions are correct. Recall measures how good you find all the positives. For example, we can find 80% of the possible positive cases in our top K predictions. Here are their mathematical definitions:

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

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

$TP = True\ positive, \quad FP = False\ positive, \quad TN = True\ negative, \quad FN = False\ negative.$

B.2 Average Precision

Let's create an over-simplified example in demonstrating the calculation of the average precision. In this example, the whole dataset contains 5 apples only. We collect all the predictions made for apples in all the images and rank it in descending order according to the predicted confidence level. The second column indicates whether the prediction is correct or not. In this example, the prediction is correct if $IoU \geq 0.5$.

Let's take the row with rank 3 and demonstrate how precision and recall are calculated first. Precision is the proportion of TP = 2/3 = 0.67. Recall is the proportion of TP out of the possible positives = 2/5 = 0.4. Recall values increase as we go down the prediction

Rank	Correct?	Precision	Recall
1	True	1.0	0.2
2	True	1.0	0.4
3	False	0.67	0.4
4	False	0.5	0.4
5	False	0.4	0.4
6	True	0.5	0.6
7	True	0.57	0.8
8	False	0.5	0.8
9	False	0.44	0.8
10	True	0.5	1.0

Figure B.1: *Example of precision and recall values.*

ranking. However, precision has a zigzag pattern — it goes down with false positives and goes up again with true positives.

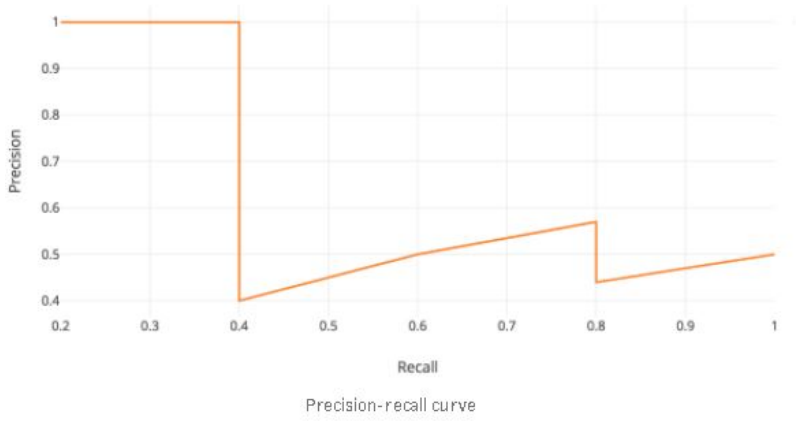


Figure B.2: *Plot of precision vs recall.*

The general definition for the Average Precision (AP) is finding the area under the precision-recall curve above.

$$AP = \int_0^1 p(r)dr$$

Precision and recall are always between 0 and 1. Therefore, AP falls within 0 and 1 also. Before calculating AP for the object detection, we often smooth out the zigzag pattern first. Graphically, at each recall level, we replace each precision value with the maximum precision value to the right of that recall level.

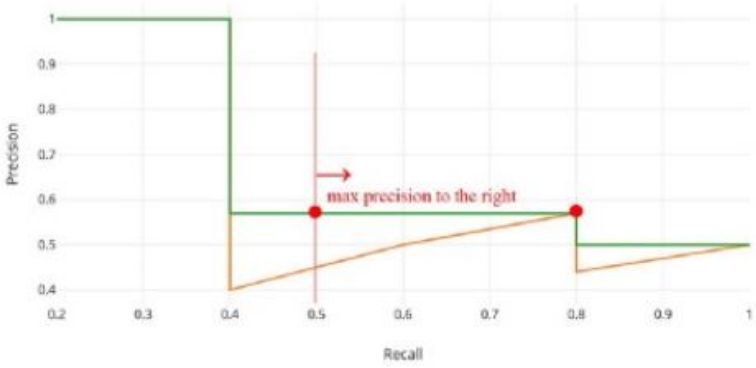


Figure B.3: *Elimination of zigzag pattern.*

Bibliography

[1] https://classroom.udacity.com/courses/ud188/lessons.

[2] https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b.

[3] https://www.coursera.org/learn/deep-neural-network?skipBrowseRedirect=true.

[4] https://www.mathworks.com/help/vision/ug/anchor-boxes-for-object-detection.html.

[5] https://www.coursera.org/learn/convolutional-neural-networks?

[6] http://host.robots.ox.ac.uk/pascal/VOC/.

[7] https://drive.google.com/drive/folders/16K68eRaGJHb3WB7Tfx-0s0h1gVQmmMSW?usp=sharing.

[8] Josh Patterson Adam Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly, 2017.

[9] Bo Chen Dmitry Kalenichenko Weijun Wang Tobias Weyand Marco Andreetto Hartwig Adam Andrew G. Howard, Menglong Zhu. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 17 Apr 2017.

[10] Patrick van der Smagt Ben Krose. *An introduction to neural networks*. The University of Amsterdam, November 1996.

[11] Sergey Ioffe Jonathon Shlens Zbigniew Wojna Christian Szegedy, Vincent Vanhoucke. Rethinking the inception architecture for computer vision. *arXiv:1512.00567*, 11 Dec 2015.

[12] Yangqing Jia Pierre Sermanet Scott Reed Dragomir Anguelov Dumitru Erhan Vincent Vanhoucke Andrew Rabinovich Christian Szegedy, Wei Liu. Going deeper with convolutions. *arXiv:1409.4842*, Sep 2014.

[13] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, 2012.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database, cvpr09. *IEEE*, 2009.

[15] Jimmy Lei Ba Diederik P. Kingma. Adam: A method for stochastic optimization. *arXiv:1412.6980v9*, January 2017.

[16] D.Kanagapushpavalli D.Renuka devi. Automatic license plate recognition. *3rd International Conference on Trendz in Information Sciences and Computing (TISC2011)*, 2011.

[17] Jarek Duda. Sgd momentum optimizer with step estimation by online parabola model. *arXiv:1907.07063*, Dec 2019.

[18] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, jun 2010.

[19] Ross Girshick. Fast r-cnn. *arXiv:1504.08083*, Sep 2015.

[20] Leonardo Ferreira Guilhoto. An overview of artificial neural networks for mathematicians.

[21] Aaron Courville Ian Goodfellow, Yoshua Bengio. *Deep learning*. MIT press, 2017.

[22] Yoshua Bengio Jason Yosinski, Jeff Clune and Hod Lipson. How transferable are features in deep neural networks? *arXiv:1411.1792v1*, November 2014.

[23] Hogne Jorgensen. *Automatic License Plate Recognition using Deep Learning Techniques*. PhD thesis, Norwegian University of Science and Technology, Department of Computer Science, July 2017.

[24] Ali Farhadi Joseph Redmon. Yolo9000: Better, faster, stronger. *arXiv arXiv:1612.08242v1*, December 2016.

[25] Ali Farhadi Joseph Redmon. Yolov3: An incremental improvement. *arXiv arXiv:1804.02767v1*, April 2018.

[26] Ross Girshick Ali Farhadi Joseph Redmon, Santosh Divvala. You only look once: Unified, real-time object detection. *arXiv arXiv:1506.02640v5*, May 2016.

[27] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Deep residual learning for image recognition. *arXiv:1512.03385*, Dec 2015.

[28] Zheng Xu W. Ronny Huang Tom Goldstein Karthik A. Sankararaman, Soham De. The impact of neural network overparameterization on gradient confusion and stochastic gradient descent. *arXiv:1904.06963*, 15 Apr 2019.

[29] Jude Shavlik Lisa Torrey. Transfer learning. *University of Wisconsin, Madison WI, USA*, 2009.

[30] Michael Nielsen. *Neural Networks and Deep Learning*. Online book, December 2019.

[31] Trevor Darrell Jitendra Malik Ross Girshick, Jeff Donahue. Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv:1311.2524v5*, October 2014.

[32] Xiuwen Liu Shaeke Salman. Overfitting mechanism and avoidance in deep neural networks. *arXiv:1901.06566*, 19 Jan 2019.

[33] Ross Girshick Shaoqing Ren, Kaiming He and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *arXiv:1506.01497v3*, June 2016.

[34] K. Simonyan and University of Oxford A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, Sep 2014.

[35] Serge Belongie Lubomir Bourdev Ross Girshick James Hays Pietro Perona Deva Ramanan C. Lawrence Zitnick Piotr Dollár Tsung-Yi Lin, Michael Maire. Microsoft coco: Common objects in context. *arXiv:1405.0312*, 1 May 2014.