Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Acknowledgment

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

List of Tables

List of Figures

List of Abbreviations

CNN Convolutional neural networks

ANN Artifitial neural networks

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, telecomunication, and even politics and economics. Others express fears that this technology may produce more problems that it would solve. In either case, the best way for academics to approach this is to explore this technology and develop a better understanding of the theory and methodology surrounding it.

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 application to identify Algerian license plate numbers using image input of the cars. The techniques used are relatively outdated considering the high rate of innovation in deep learning. The two main research papers that this work is based upon came out in 2016 and 2018. The first one is a CNN model called Faster-RCNN and the other one is called YOLO, both these works showed practical state of the art results in the tasks of object detection and classification during the time they were released. This project also includes the creation of a lebeled 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 License plate detection.

Chapter 1

Feed-Forward neural networks

To understand the technique used in this report, it is necessary to understand basic neural networks functioning. 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. Let's introduce the idea of 'perceptron'.
Perceptrons are the building blocks of neural networks, and the best way to get stated is with an example. Assume at the university's admission office the students are evaluated with two pieces of information, the results of a test and their grades in school. Let's take a look at some sample students, see fig. 1.1.

Figure 1.1: *test vs grades*

The data on the figure can be nicely 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 is going to be our model.
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 and add no new information to our model. Now, the natural question that arises: how do we find the line ?

We start by labeling the axis $\mathbf{x} = \{x_1, x_2\}$. The boundary line separating the students has a linear equation specifically: $2x_1 + x_2 - 18 = 0$. Plotting the grades in the equation

Figure 1.2: *seperating line*

gives rise to a score, if the score is positive –the student gets plotted in above the line–, the student gets accepted with otherwise not. 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 that the label will be, namely:

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

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

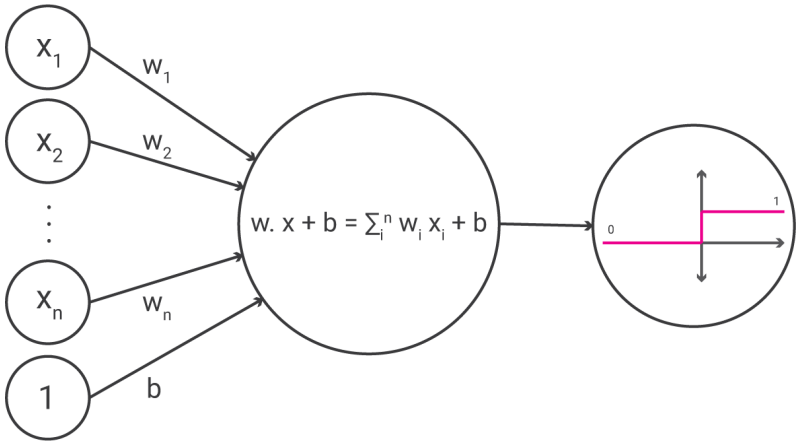


Figure 1.3: *perceptron's graph representation*

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

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

This function, becomes large when our network 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 2.2.3.

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 the 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*. 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:

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

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 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 solve 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

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

It is a well known result that, given a point $x \in \mathbb{R}^n$, the gradient at that point 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) see .

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. And 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

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

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. Here the output consists of j outputs rather that a single one. See fig. 1.5 to illustrate the output layer.

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 eq. (1.4) now become important. In fact, we are only going to

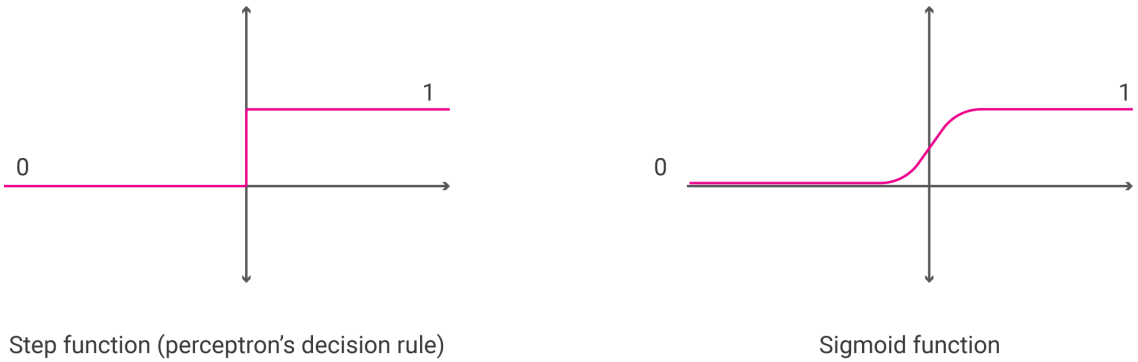


Figure 1.4: *sigmoid vs step function. The two plots clearly show cast the continuity of the sigmoid.*

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'll first calculate $\frac{\partial}{\partial w_j}\hat{y}$, where $\hat{y} = \sigma(w \cdot x + b)$.

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

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:

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

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

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 **feedforward** neural nets or **multilayer perceptron** (MLPs) are as the name indicates are 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 **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: $f(x) = x$. The figure below depicts the architecture described above.

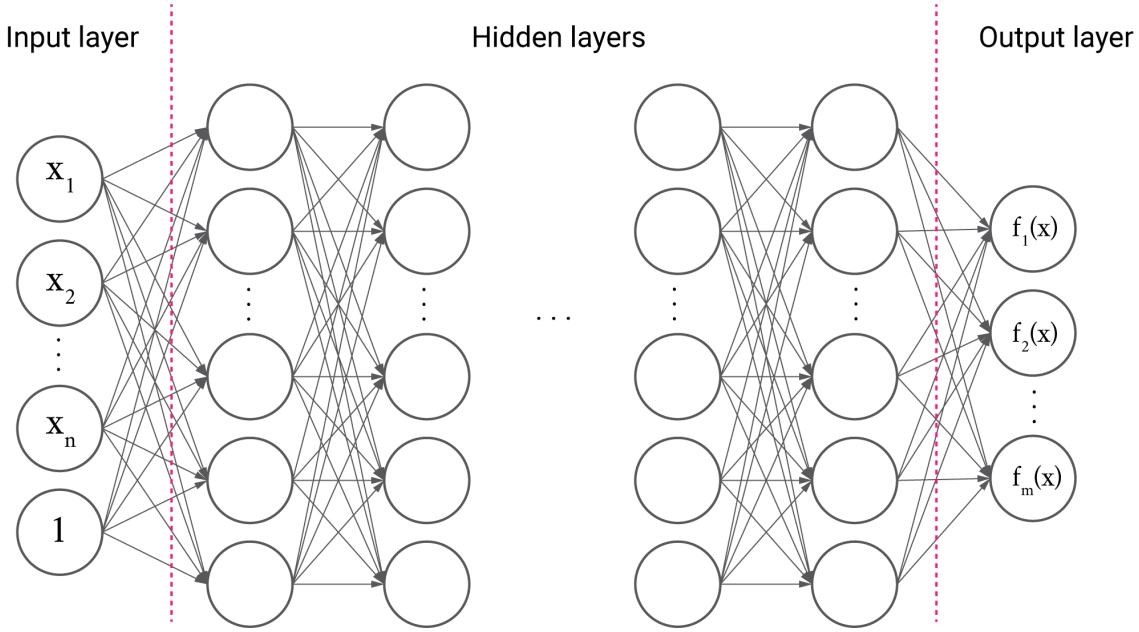


Figure 1.5: *A visual representation of a feed-forward network which approximates some function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ by computing the function $f^*(x) = (f_1^*(x), \ldots, f_m^*(x))$. In this approach, the network is shown as a directed weighted graph. Here $x = (x_1, \ldots, x_n)$*

For notation, set $w_{a,b}^n \in \mathbb{R}$ 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.

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

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

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

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 look likes

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

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".

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

$$z^l_j = \sum_k w^l_{j,k} a^{l-1}_k + b^l_j \qquad \text{so that} \qquad a^l_j = \sigma(z^l_j).$$

Additionally, we denote the following:

$$\delta^l_j = \frac{\partial}{\partial z^l_j} L \tag{1.14}$$

This value will be useful for propagating the algorithm backwards through the network and directly related to $\frac{\partial}{\partial w^l_{i,j}} L$ and $\frac{\partial}{\partial b^l_j} L$ by the chain rule. since

$$\frac{\partial L}{\partial w^l_{i,j}} = \frac{\partial L}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{i,j}} = \delta^l_j a^{l-1}_i \tag{1.15}$$
$$\frac{\partial L}{\partial b^l_j} = \frac{\partial L}{\partial z^l_j} \frac{\partial z^l_j}{\partial b^l_j} = \delta^l_j. \tag{1.16}$$

The value a^{l-1}_j has already been calculated through the forward pass. The only remaining term to calculate is δ^l_j 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, apply the chain rule once again

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

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 developed 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.

1.5 Problems related to neural nets

Neural networks are extremely powerful function approximators, but during the design of 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. This 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 [6].
Another issue in deep neural nets is the **vanishing gradients** problem. As we learned from backp-ropagation, 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 [6]. 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 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 on section 1.1, eq. (1.4). Therefore, instead 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. One downside though of SGD is, once it reaches close to the minimum value then 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.

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 [7]. 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 [7]. The update rule of the adam optimizer is as follow, first calculate the running average of the weights as follows (eq. (1.19)):

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

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

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)) [7]. 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 equation apply to the bias term b. The reason why adam is effective is because it is invariant to 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 [7]. 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 $\hat{v}_t = 0$ [1].

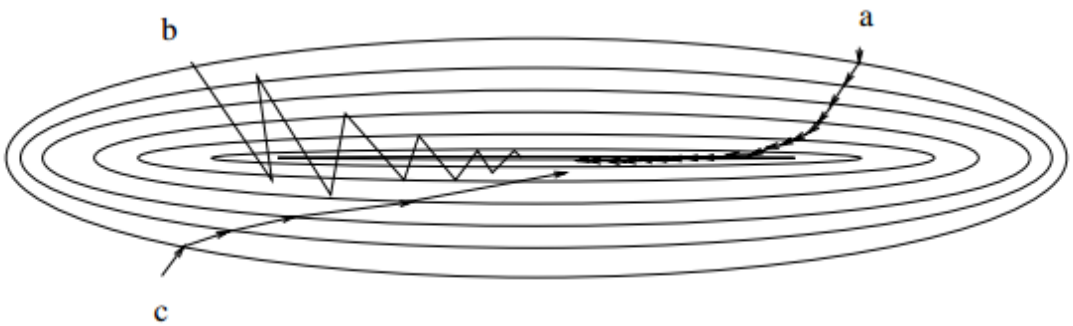


Figure 1.6: *The descent in weight space. The concentric ellipsis represents the counters of the cost function. a) SGD with a small learning rate, b) SGD with a large learning rate, c) Adam with a large learning rate.*

Chapter 2

Neural network Variant: Convolutional Neural Networks

Convolutional networks, also known as convolutional neural networks, or CNNs, 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. Convolutional networks 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. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [8].

2.1 The 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 convolution 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 (RGB images), and the kernel is usually a multidimensional array of parameters

that are adapted by the learning algorithm. These multidimensional arrays are referred 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:

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

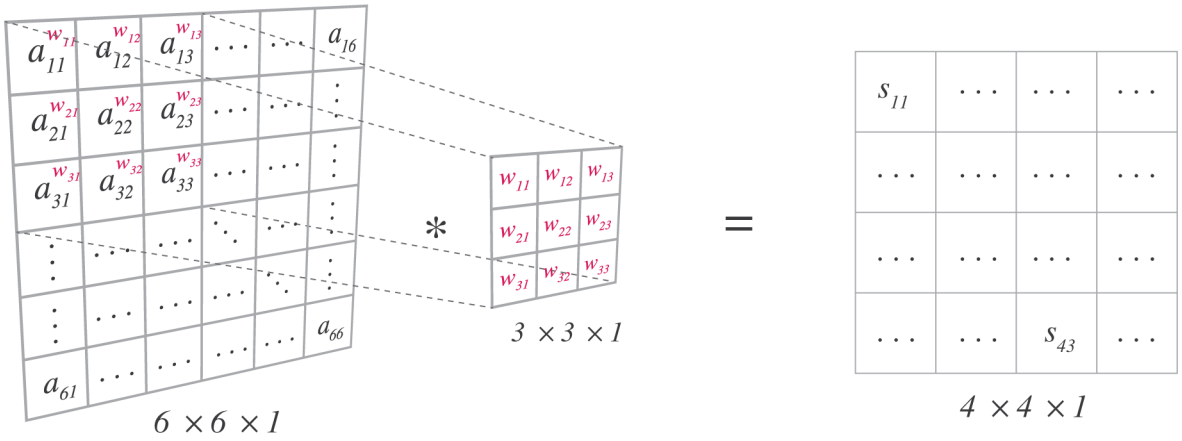
Convolution is commutative, meaning we can equivalently write:

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

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*

2.2 Convolution networks architecture

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. Convolutional networks, 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 [8]. 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 convolutional neural net, 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

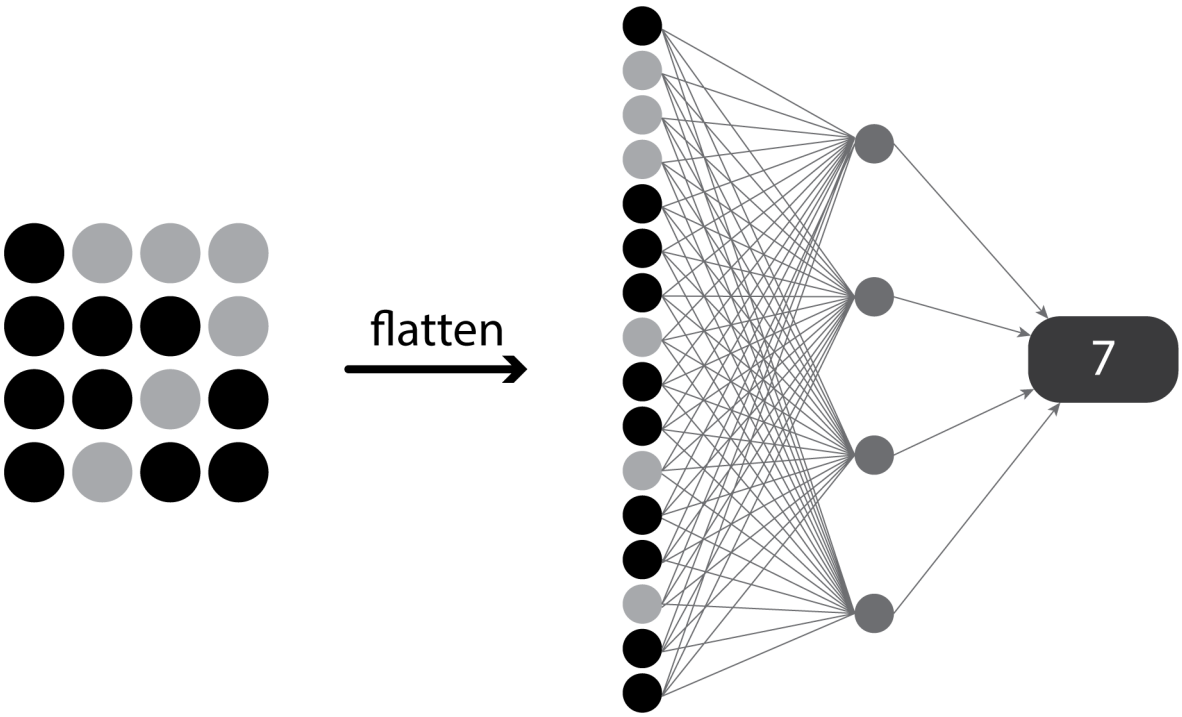


Figure 2.2: *Traditional neural network connections. The last layer has been replace by a black box for simplicity*

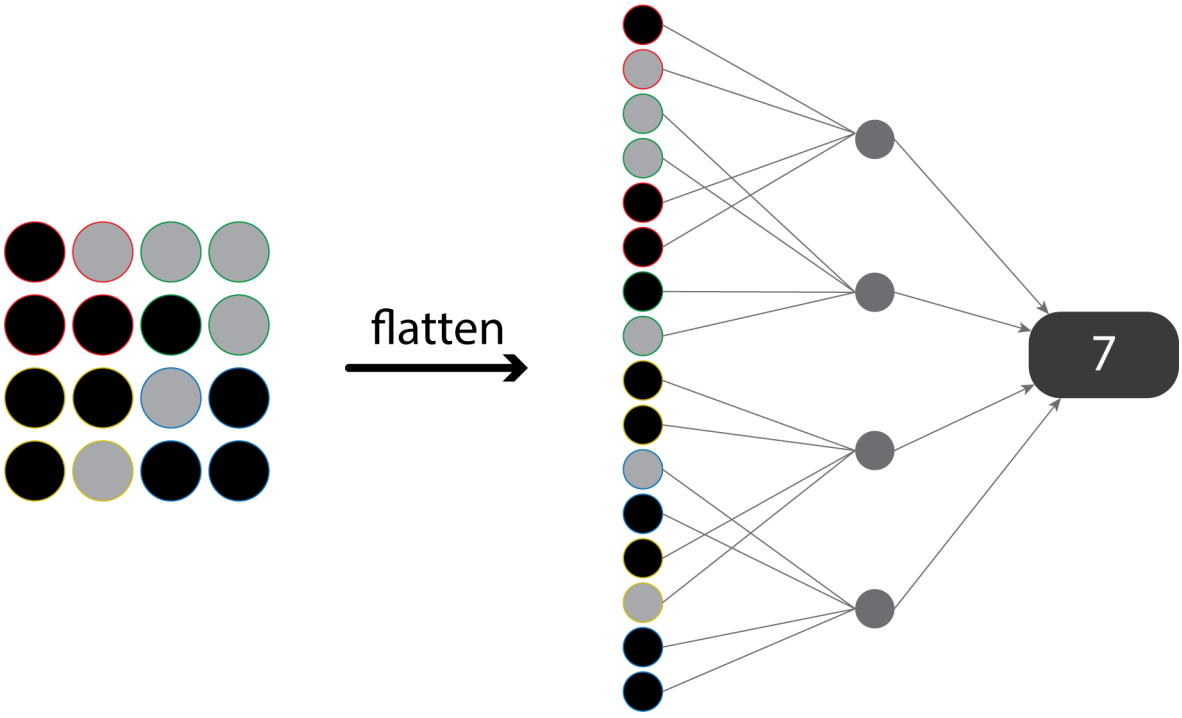


Figure 2.3: *Sparce connectivity*

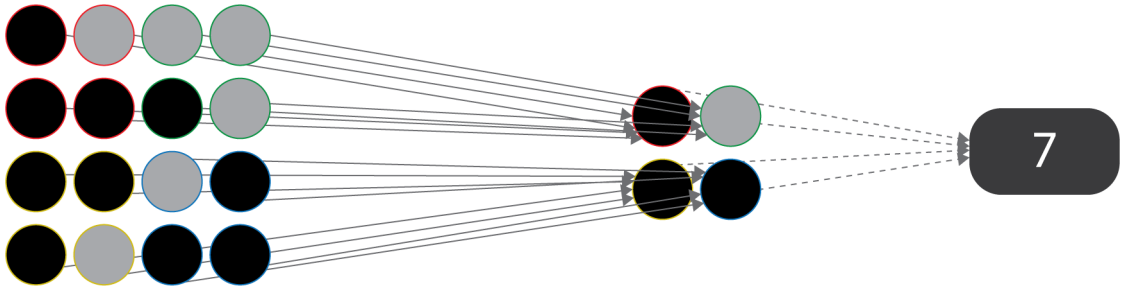


Figure 2.4: *Sparce connectivity after rearrangement*

image. This does not affect the runtime of forward propagation—it is still $O(k \times n)$—but it 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 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.

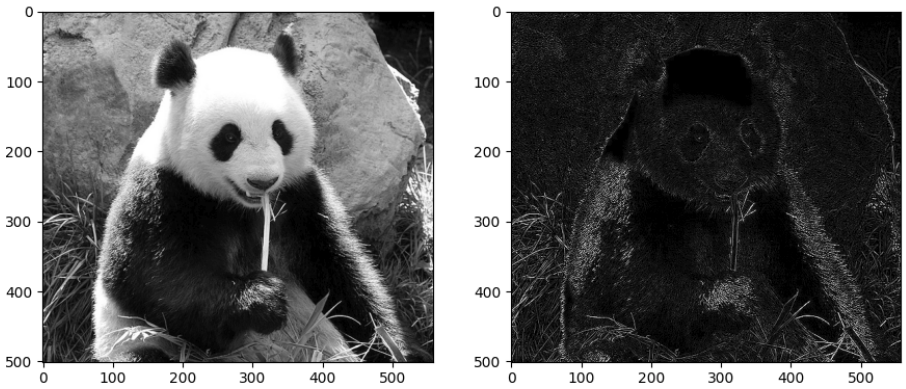


Figure 2.5: *2D convolution. Where there is no change or little change of intensity in the original picture, the high pass filter block those areas out and turn the pixels black. But in the areas where a pixel is way brighter than its immediate neighbors, the high pass filter enhance the change and create 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, 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 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

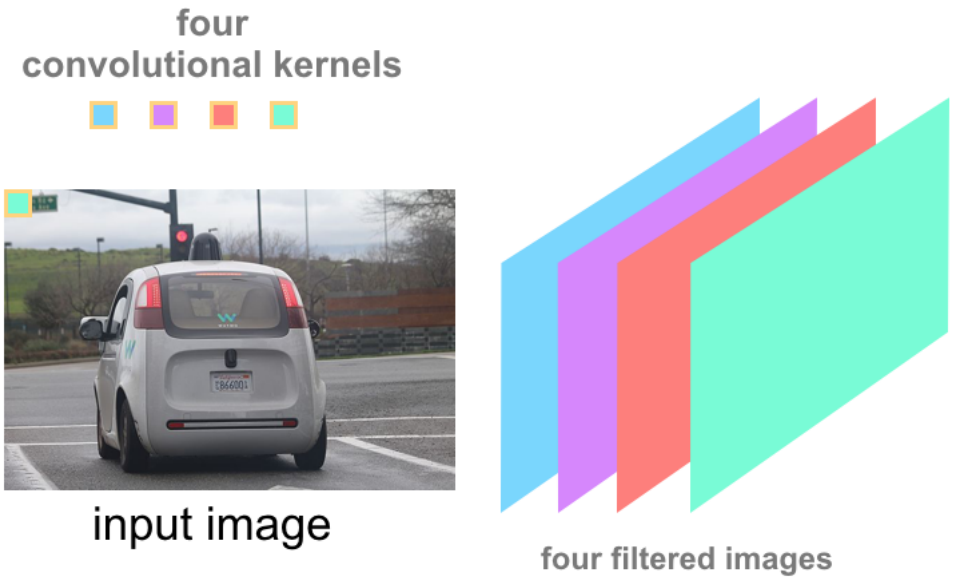


Figure 2.6: *Multiple filters for multiple pattern detection*

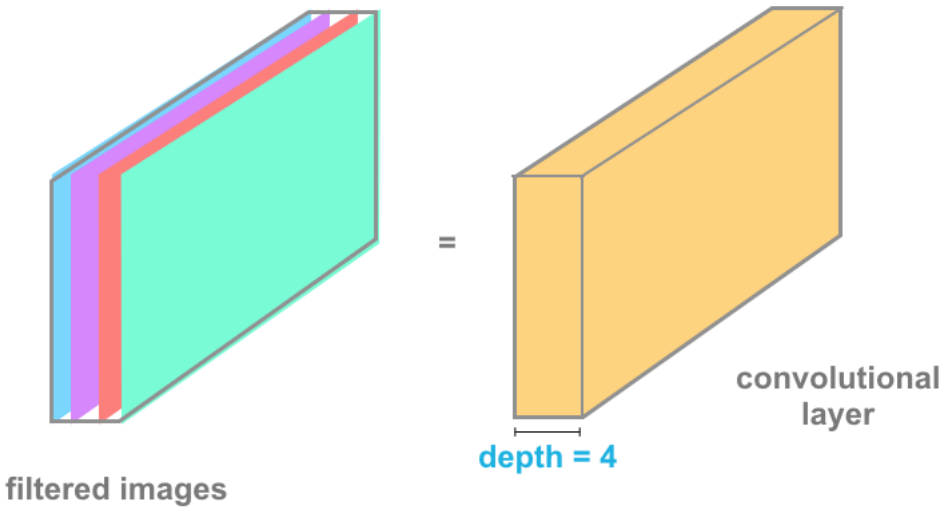


Figure 2.7: *A complete convolutional layer with 4 filters*

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 Fully connected 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(equ from chapter 1). 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 convolutional neural network 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 fig. 2.1, the stride was one. We move the convolution window horizontally and vertically across the image one pixel at a time [2]. The width and height of the output of the convolution is given by eq. (2.5), if the input image is $n \times n$, with a filter $f \times f$:

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

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 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.

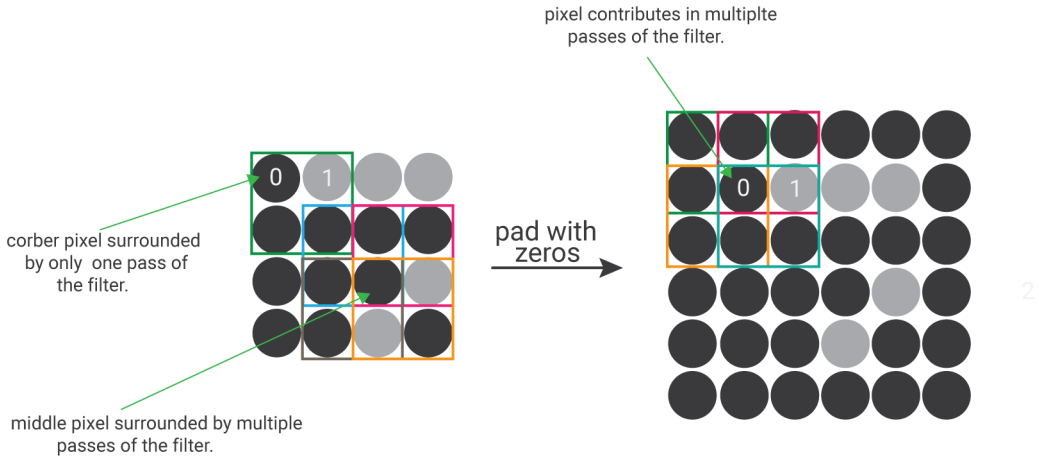


Figure 2.8: *Padding example.*

2.5 Pooling

Pooling function is the next type of layer in convolutional neural networks. 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 [8]. see figure on how to perform max pooling.

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, 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 [8].

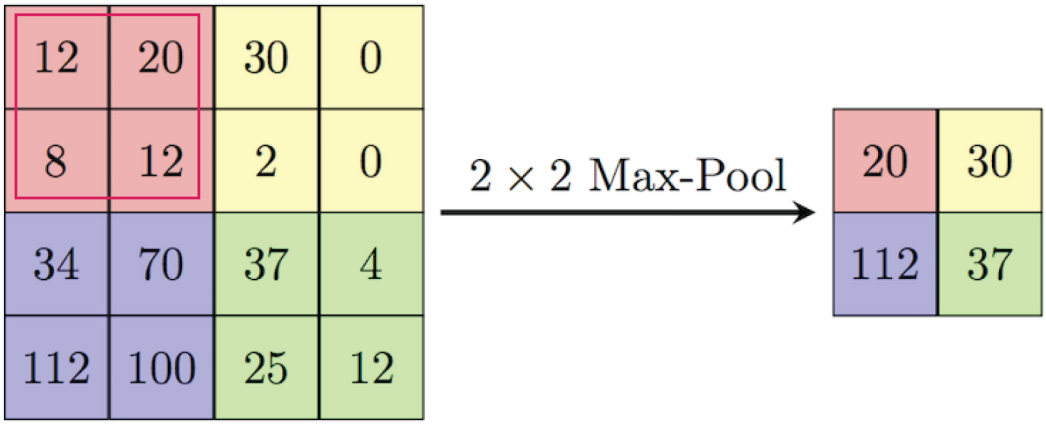


Figure 2.9: *Maxpooling example. As in the convolution operation, we slide a window across the image typically a* 2×2 *window. The value of the corresponding node in the max pooling layer is calculated by just taking the maximum of the pixels contained in the window. The pooling function is applied independently 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.*

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

So why look at 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 convolutional neural network 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, which is a data-set of over 14 million images belonging to 1000 classes. It was one of the famous model submitted to ILSVRC-2014. It makes the improvement over AlexNet 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 NVIDIA Titan Black GPU's. see fig. 2.10.

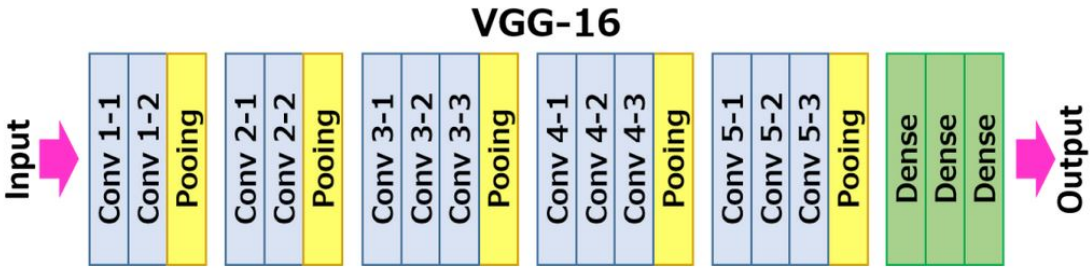


Figure 2.10: *vgg 16*

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 conv. and 3 FC layers) to 19 weight layers in the network E (16 conv. and 3 FC layers). The width of conv. 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.

2.6.2 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.

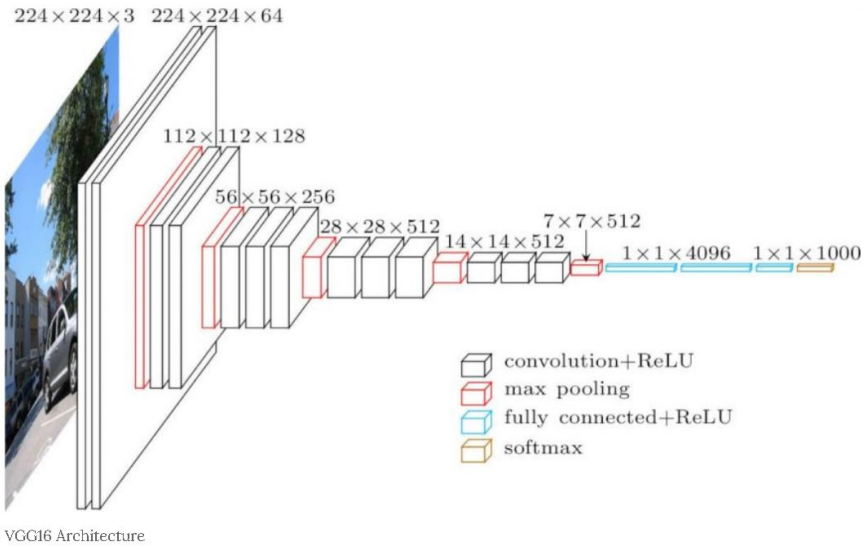


Figure 2.11: *vgg 16 config*

The fundamental breakthrough with ResNet was it allowed us to train extremely deep neural networks with 150+layers successfully. 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"

Skip Connection

ResNet first introduced the concept of skip connection. fig. 2.12. below illustrates skip connection. The figure on the left is stacking convolution layers together one after the other. On the right 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 function. The main two reasons why skip connections work are :

- They 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 2.12: *resnet1*

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.

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. 2.13. 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. And 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.

As a solution, 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”.

Figure 2.14 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. See fig. 2.15

Figure 2.13: *inception1*

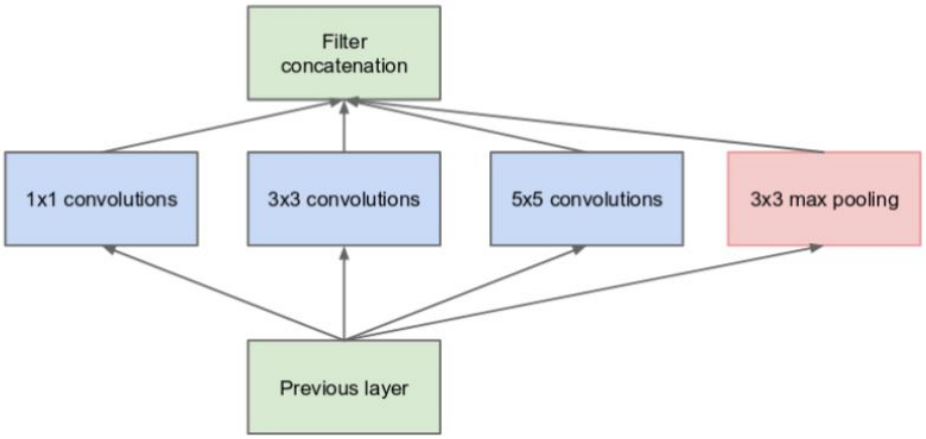


Figure 2.14: *inception2*

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. 2.16.

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.

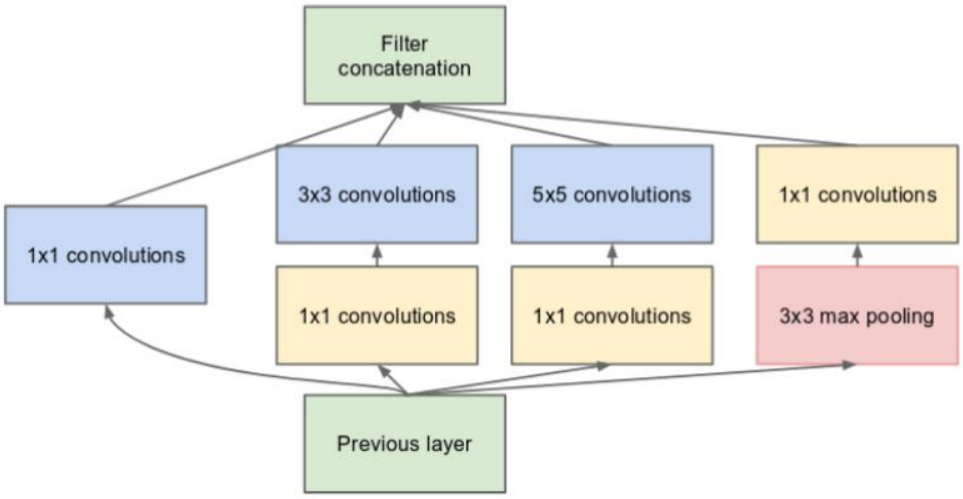


Figure 2.15: *inception3*

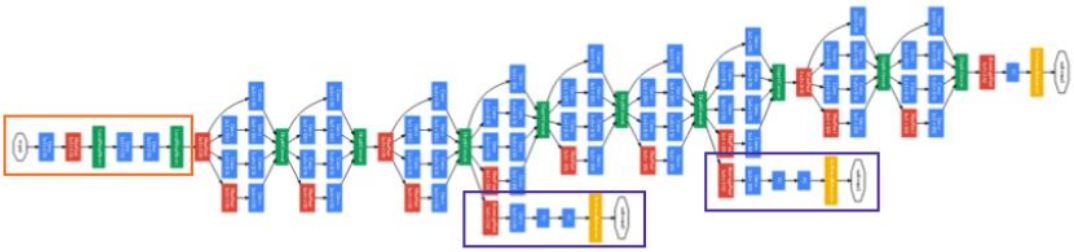


Figure 2.16: *inception4*

To prevent the middle part of the network from "dying out", the authors introduced two auxiliary classifiers (The purple boxes in fig. 2.16). 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. Needless to say, auxiliary loss is purely used for training purposes, and is ignored during inference.

$$total\ loss = real\ loss + 0.3\ aux\ loss_1 + 0.3\ aux\ loss_2$$

2.6.4 Mobilenet

MobileNet is a CNN architecture model for used for object detection and image Classification, generally used in mobile 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".

Depthwise Separable Convolution

Before we get into the definition of "Depthwise Separable Convolution" we need to go over some aspects of the convolution operation. Let's consider an input matrix of shape $D_f \times Df \times M$ as shown in fig. 2.17. If our input was an RGB image then M would be equal to 3. If we apply a convolution using a filter of shape $D_k \times Dk \times M$ we would obtain an output of size $D_G \times DG \times 1$ if we apply the same convolution using N filters of the same shape and concatenate the results we would obtain an output shape of $D_G \times DG \times N$, see fig. 2.18.

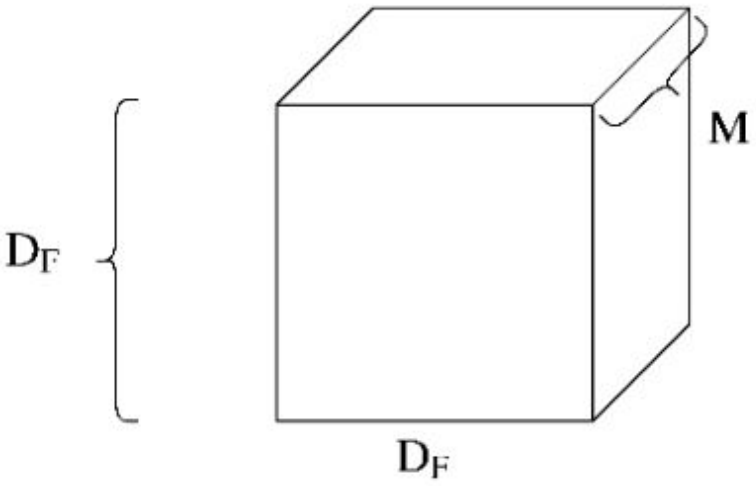


Figure 2.17: *Convolution input*

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

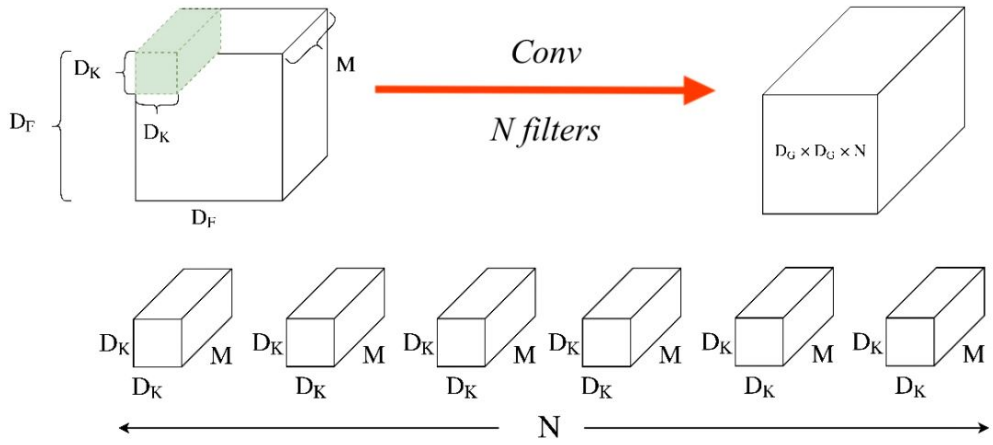


Figure 2.18: *Convolution operation*

$D_k \times D_k \times 1$ see fig. 2.19. 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. 2.20

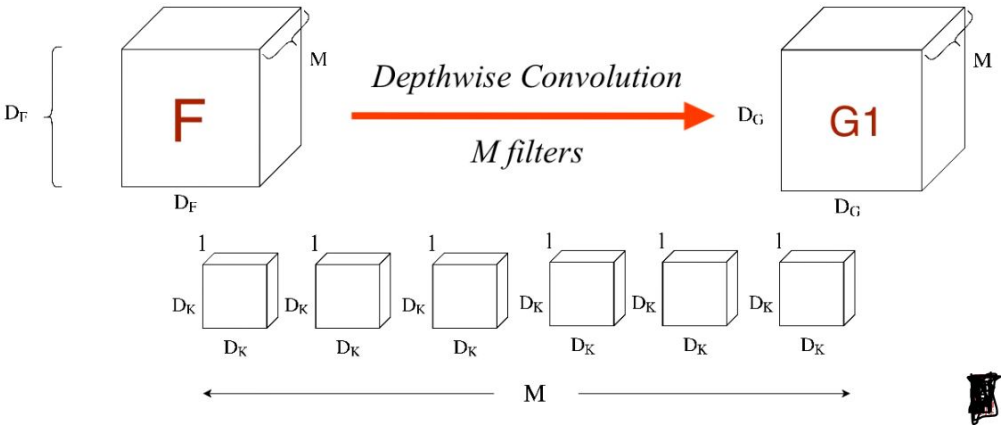


Figure 2.19: *Depth-wise Convolution*

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

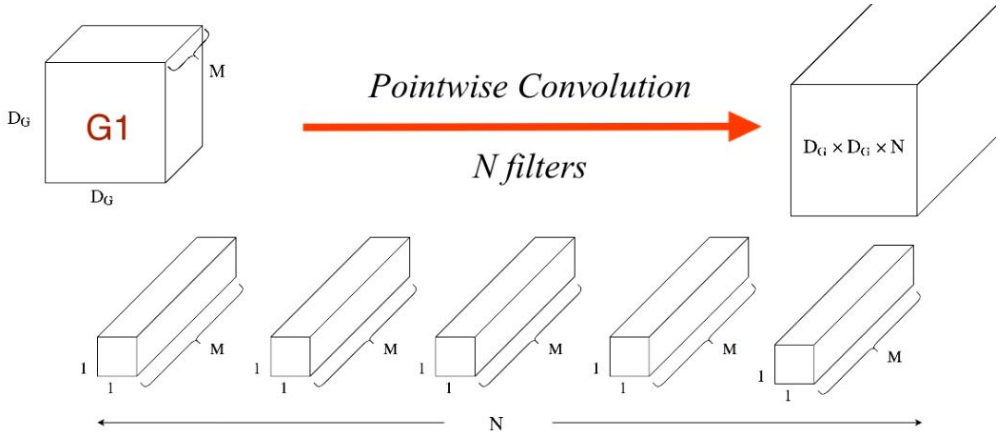


Figure 2.20: *Point-wise Convolution*

Mobilenet model

The Mobilenet model is composed of convolutional and Max Pool layers where the full structure is demonstrated in the table seen in fig. 2.21

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 2.21: *mobilenet table*

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 [9]. 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 [9] 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 reinitialize the weights (but some new layers might be added or changed depending on the task at hand). The existing weight are said to be *fine-tuned* to the new task at hand [10].

2.8 Data augmentation

Data augmentation is a strategy that enables practitioners to significantly 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 present 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 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.

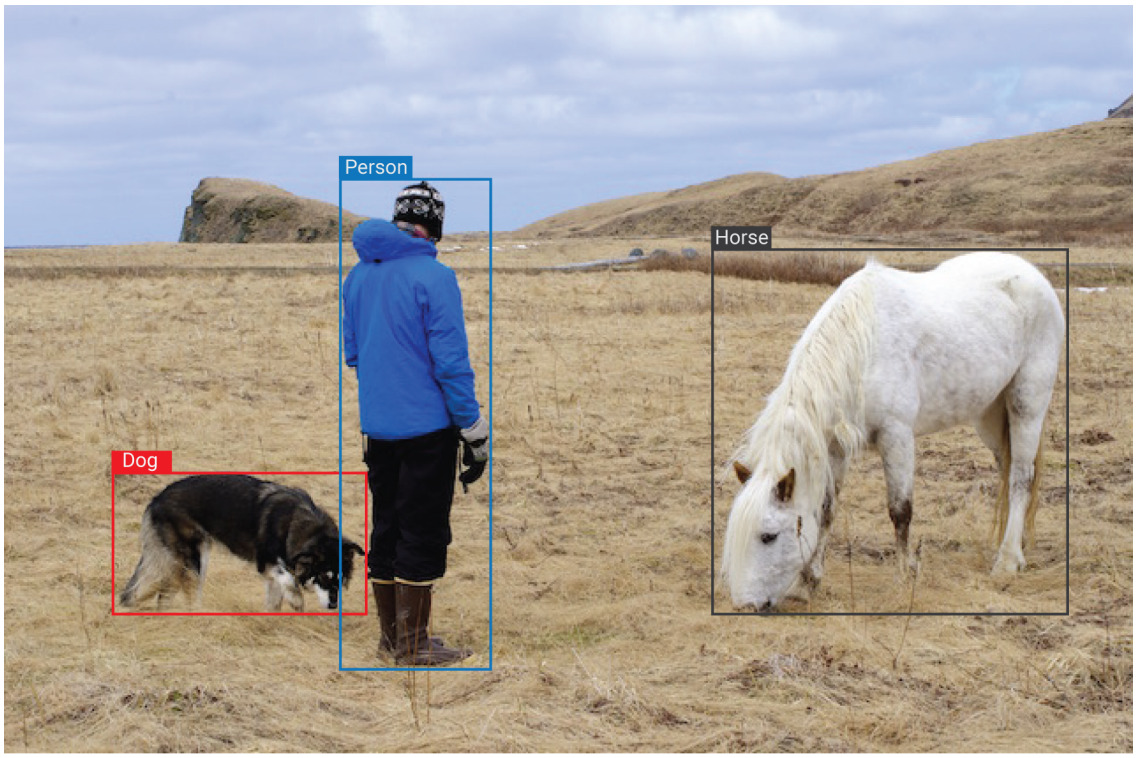


Figure 3.1: *Example of what an object detection system should accomplish*

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 it 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 which we already have a solid background of.

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 [13]. Each grid cell predicts B bounding boxes, using anchor boxes. Anchor boxes are predefined boxes of certain width and height. Those boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect. They are typically chosen based on object sizes in the training data [3], see . 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 wont 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. 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) [13]. This should be ideally 1 if there is an object otherwise 0 [12].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 [4]. 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. See fig. 3.2.

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

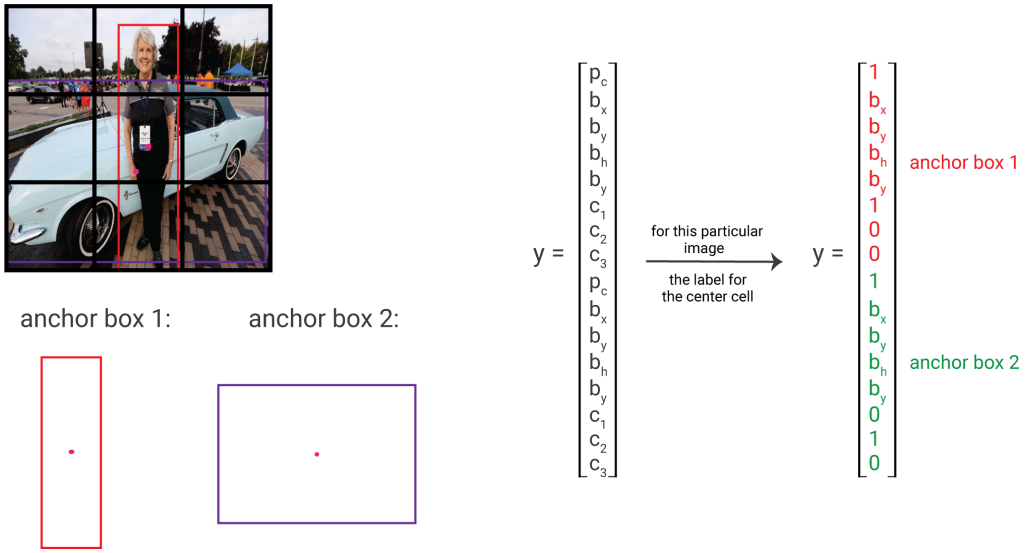


Figure 3.2: *Example of anchor boxes. As we can see on the figure, the anchor boxes capture the scale and aspect ratio of cars and pedestrians. Indeed, most cars and humans will have approximately the same scale and aspect ratio. The vector y is composed of the objectness score as well as the bounding 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.*

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 appendix. 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 [11]. so by using an input image of 416 we get an output feature map of 13×13. The second issue anchor boxes address is the training instability [11]. 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 $pw,\ p_h$, then the predictions correspond to [11]:

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

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

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

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

Since we constrain the location prediction the parametrization is easier to learn, making the network more stable [11], see fig. 3.4. The question that naturally rises is: How, at the beginning, do we get $p_w,\ p_h$? 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 (IoU) to proceed.

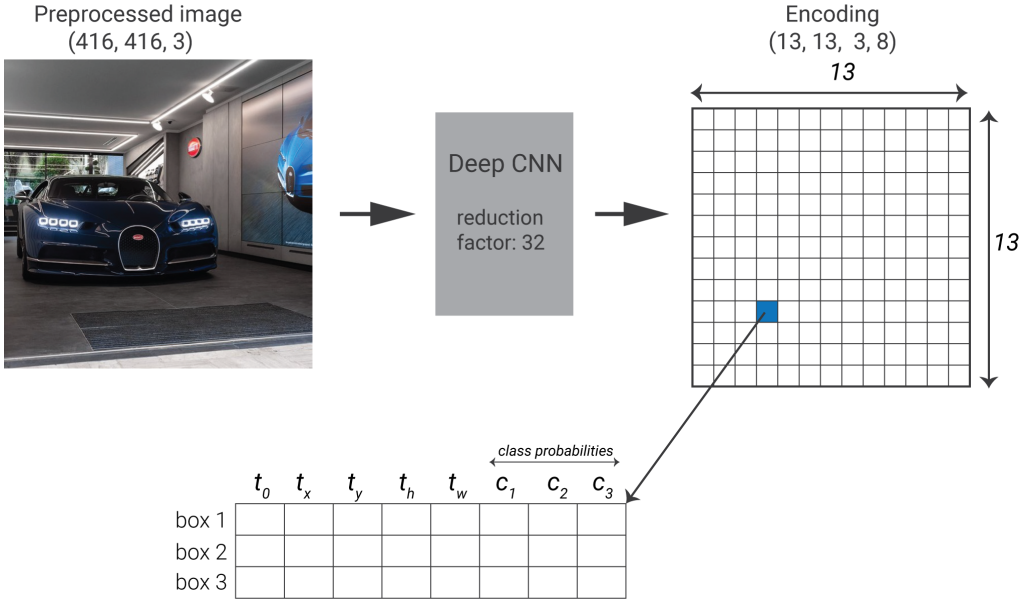


Figure 3.3: *The true output of YOLOv3 after introducing the training instability issue. The net works output a* $13 \times 13 \times (8 \times 3)$ *in this case, or simply put* $13 \times 13 \times 24$ *output volume. Each grid cell outputs three bounding boxes.*

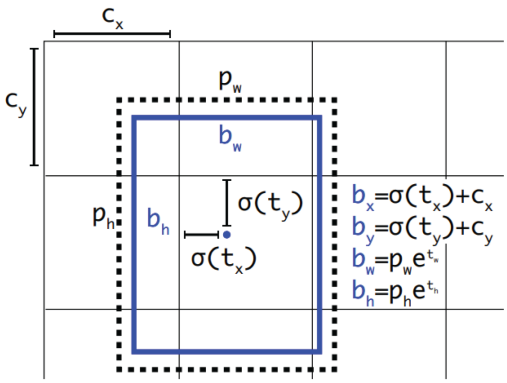


Figure 3.4: *Bounding box calculation. We predict the width and height of the box as offsets from manually chosen anchor boxes. We predict the center coordinates of the box using a sigmoid function.*

During training we optimize the following, multi-part loss function. As we can see 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 appendix.

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

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.

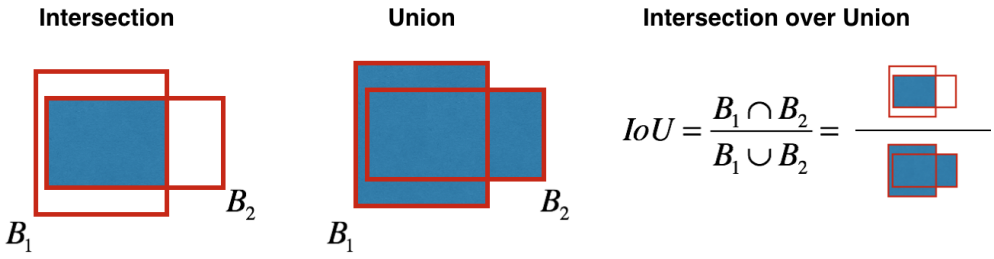


Figure 3.5: *Intersection over union.*

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 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 [5].

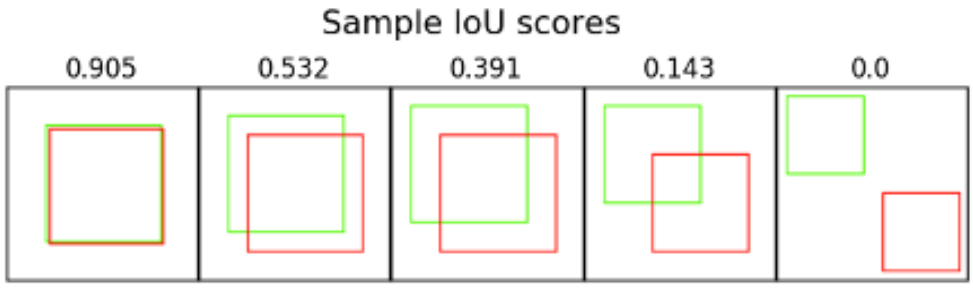


Figure 3.6: *Sample IoU scores.*

Back to our original question, how non-max suppression works. First, all the boxes having an $objectness \times the\ class\ probability$ less or equal than some threshold are discarded (typical value used 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 an 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 Intersection-over-Union (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. The main different here with its' predecessor Fast R-CNN is that the later uses an algorithm called "selective search" to generate region proposals. 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. The architecture is as shown in fig. 3.7.

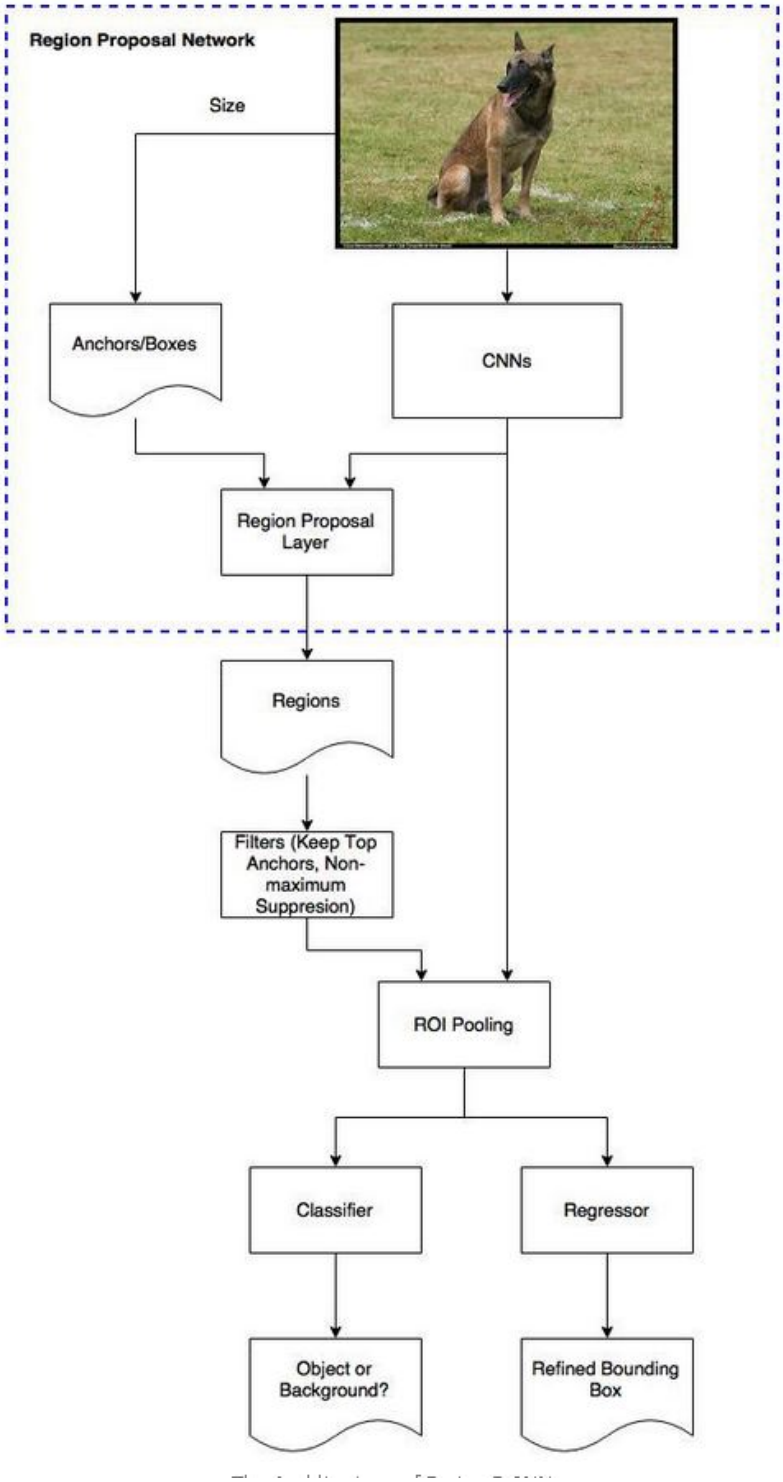


Figure 3.7: *rcnn1*

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: $128 \times 128,\ 256 \times 256,\ 512 \times 512$. And in 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 number of boxes that will be considered by the classifier network.

These anchors work well for Pascal VOC data-set as well as the COCO data-set. However you 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.

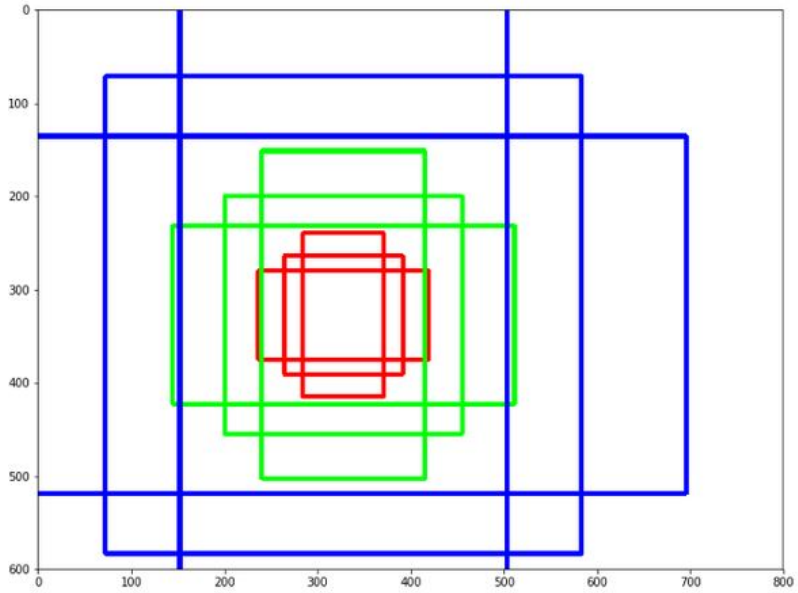


Figure 3.8: *rcnn2*

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. 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. Since the RPN performs a 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.

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.

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.

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 :

- A 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 pre- defined 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 doesn't 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. What's the benefit of RoI pooling? One of them 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. The fig. 3.9 below shows the working of ROI pooling.

This 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 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 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).

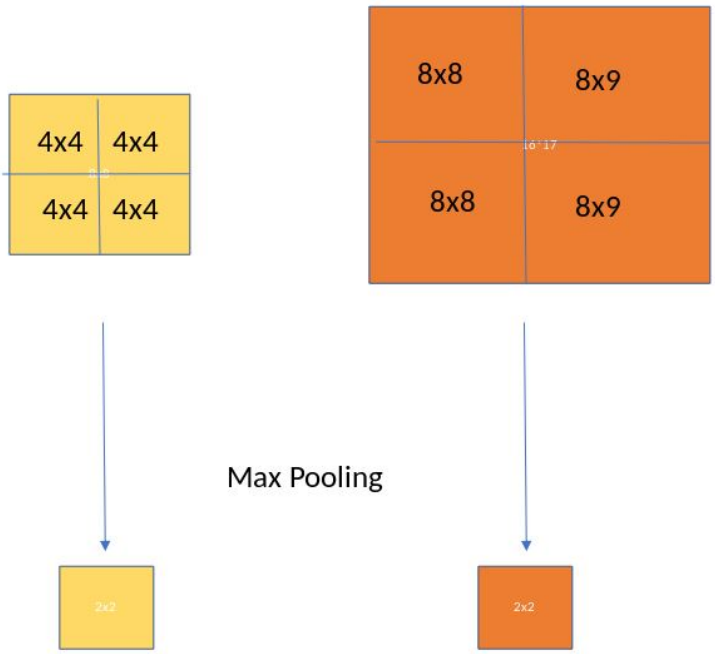


Figure 3.9: *rcnn3*

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

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

Chapter 4

Method and results

4.1 Method description

For building the application of a license plate detector, we used a part-by-part approach rather than and end-to-end approach. The reason being the lack of labeled data sets designed for this specific application and 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, we call 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, we call "the digits network" receives the detected plates from the first module, detects and recognize the 10 digit classes from 0 to nine, 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 if 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

4.2 Data collection

This part includes the manual collection of images of Algerian license plates in different positions, angles, lighting, distance, size, color ... etc. The data set contained close to 1000 images set to be used for labeling. Figure 4.2 below demonstrate some examples.

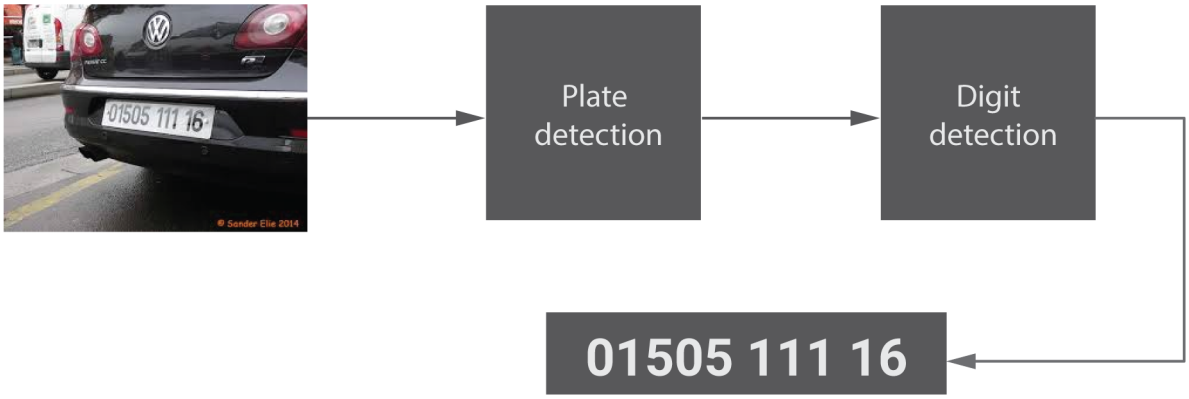


Figure 4.1: *The entire system flow. An input image is fed to the plate network, detects the plate crop it according to the output bounding box, fed it to the digit network. The digit network detects the different digits in the plate and output a prediction according to the position of the bounding boxes.*

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.

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 tumor.

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 by manually 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 to be completed 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.

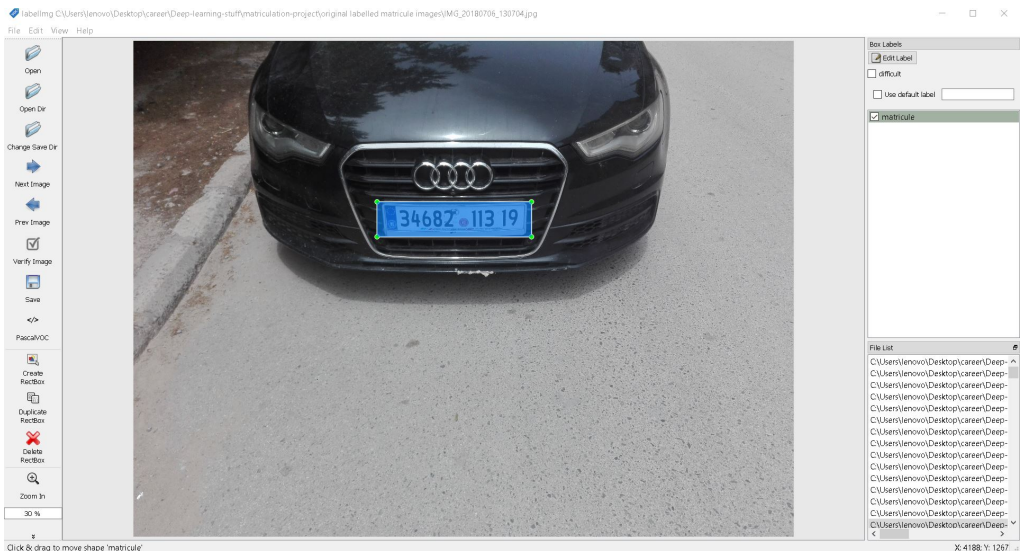


Figure 4.3: *img2*

4.4 Plate detection and localization

This part of the application requires training a set of models to detect and localize an Algerian license plate in any given image. The models trained belonged to two families of CNN models, Faster RCNN and YOLOv3.

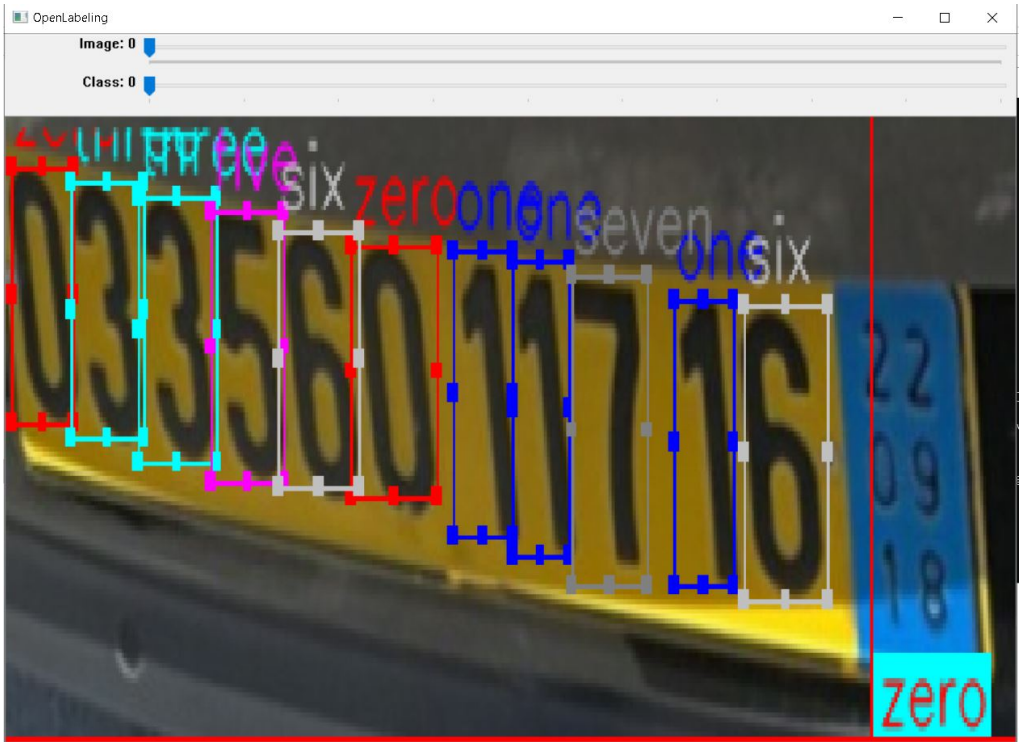


Figure 4.4: *img3*

4.4.1 Using Faster RCNN

For this family of models, there are three many parameters which can be adjusted in order to obtain optimal results. The following parameters were chosen for both the digit and plate networks:

- Anchor scales
- The backbone network
- The number of training epochs

This choice is justified by the fact that the other parameters as specified by the original papers 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 Adam for all instances of training. The anchor scales and ratios used were :

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

The backbone networks used were :

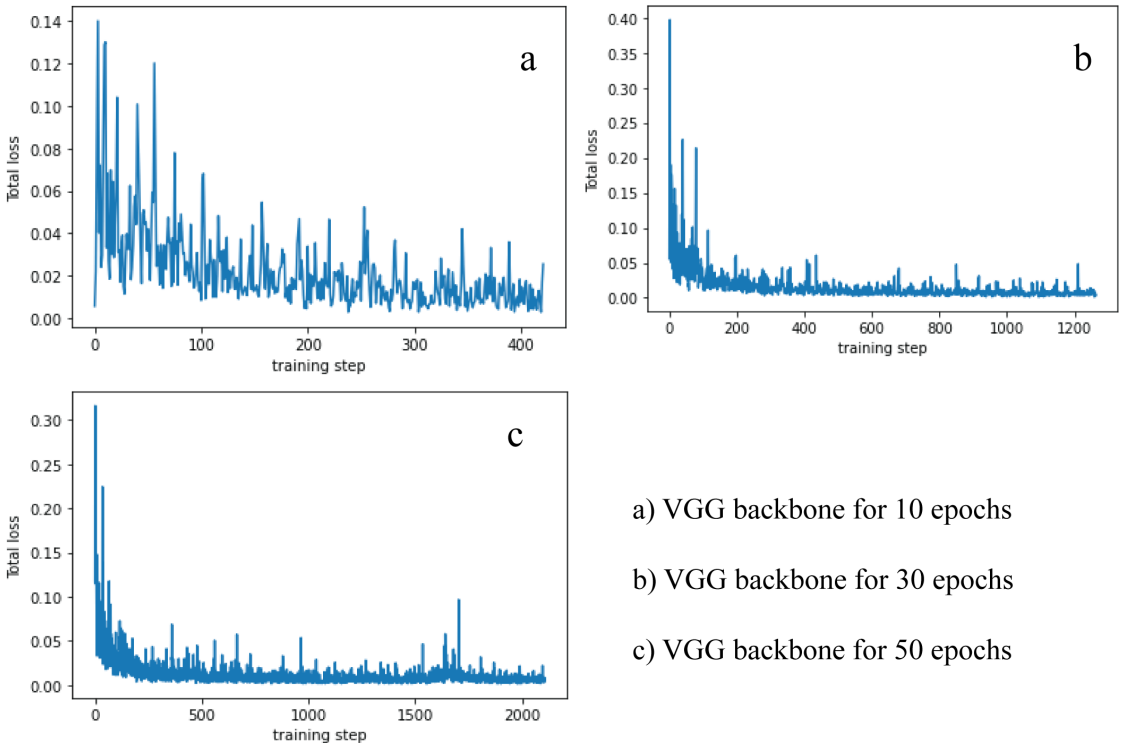
- VGG16
- Mobilenet
- Inception
- ResNet

The numbers of training epochs used were :

- 10 epochs
- 30 epochs
- 50 epochs

Plate network

This implies that the number of training processes launched is $2 \times 4 \times 3 = 24$. The value of the total cost for each training process was plotted with respect to training steps.



a) VGG backbone for 10 epochs

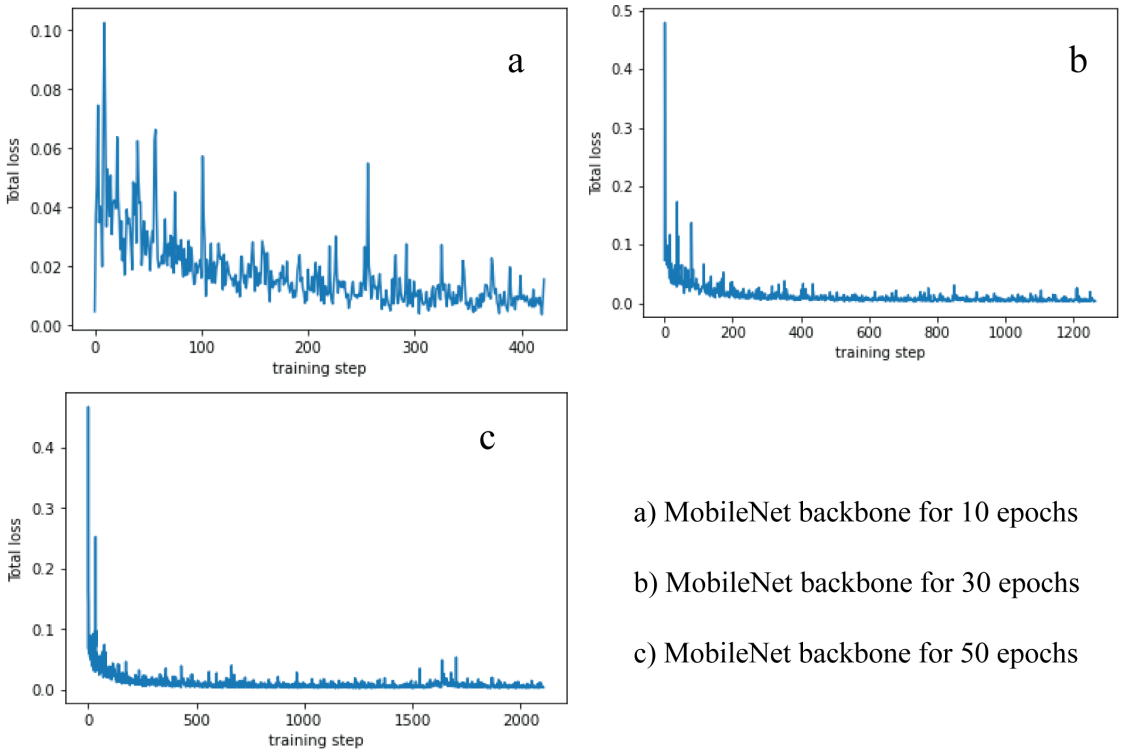
b) VGG backbone for 30 epochs

c) VGG backbone for 50 epochs

Figure 4.5: *VGG-16 for the first scales and aspect ratios*

After these models are trained, they need to be tested on both the training and testing set for analysis. The criterion chosen is the mAP(mean average percent) which is a performance evaluation formula which takes into account the accuracy of the classification as well as the precision of the localization of objects, see (reference publication). For each set of scales and aspect ratios the results of performance on the training and test set were tabulated in table 4.1, table 4.2, table 4.4, table 4.5.

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

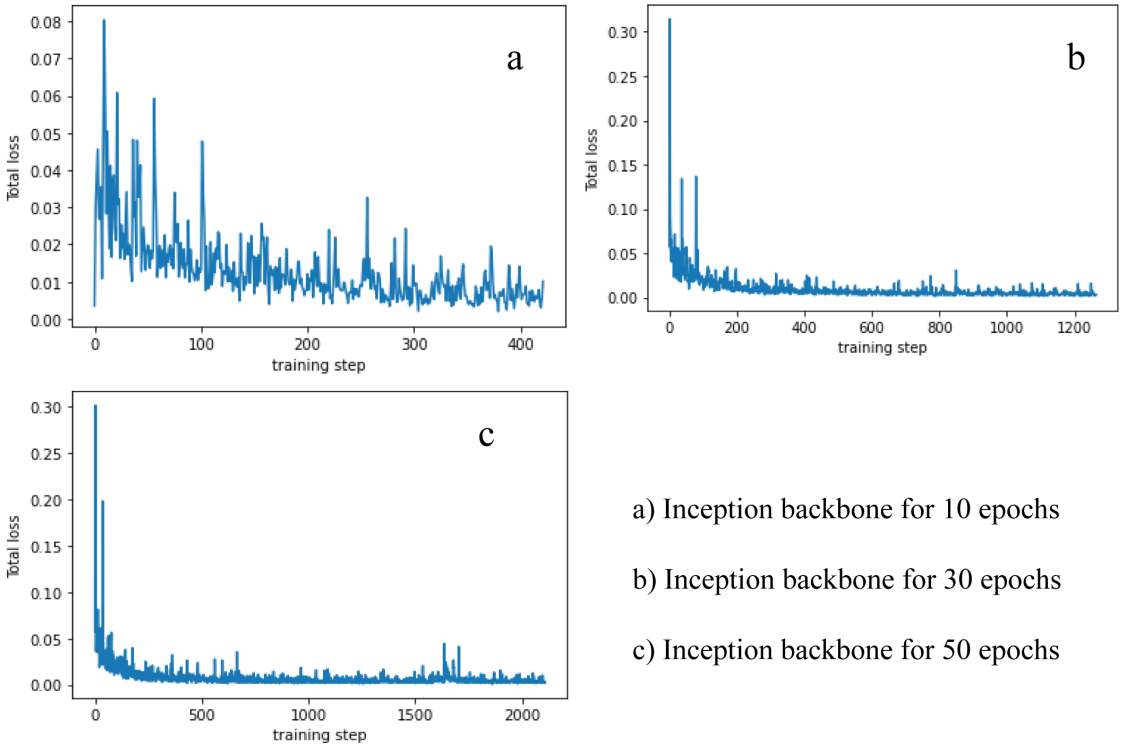


a) MobileNet backbone for 10 epochs

b) MobileNet backbone for 30 epochs

c) MobileNet backbone for 50 epochs

Figure 4.6: *mobilenet for the first scales and aspect ratios*

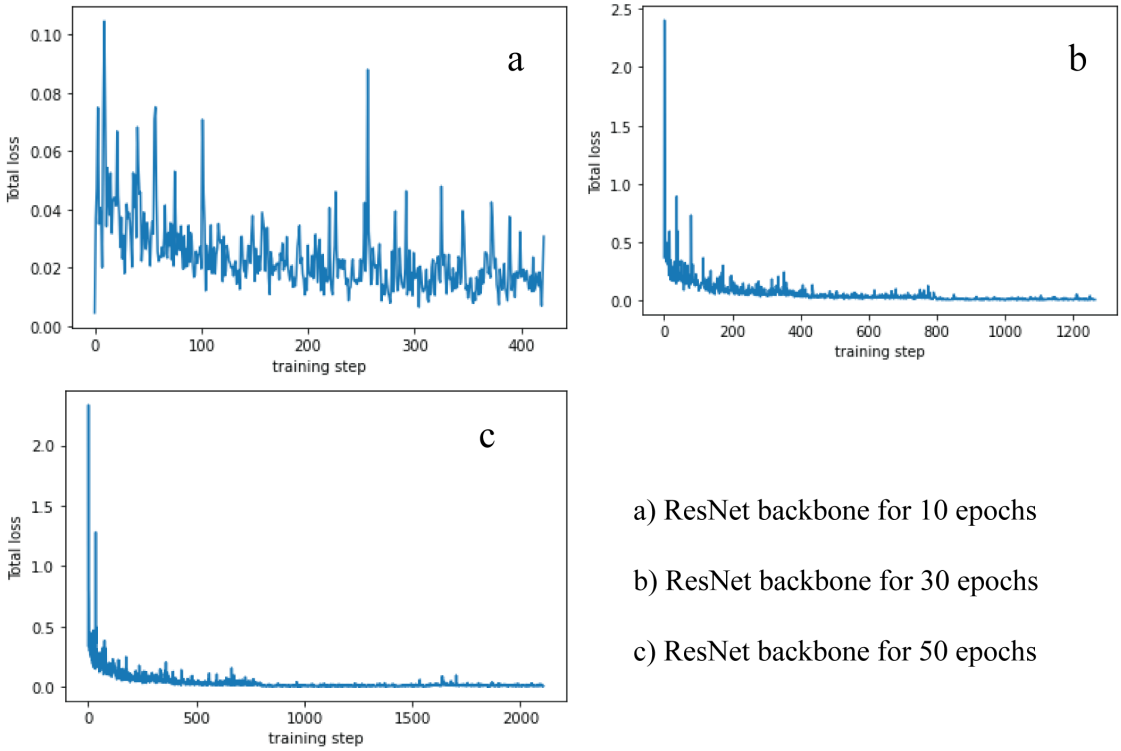


a) Inception backbone for 10 epochs

b) Inception backbone for 30 epochs

c) Inception backbone for 50 epochs

Figure 4.7: *inception for the first scales and aspect ratios*

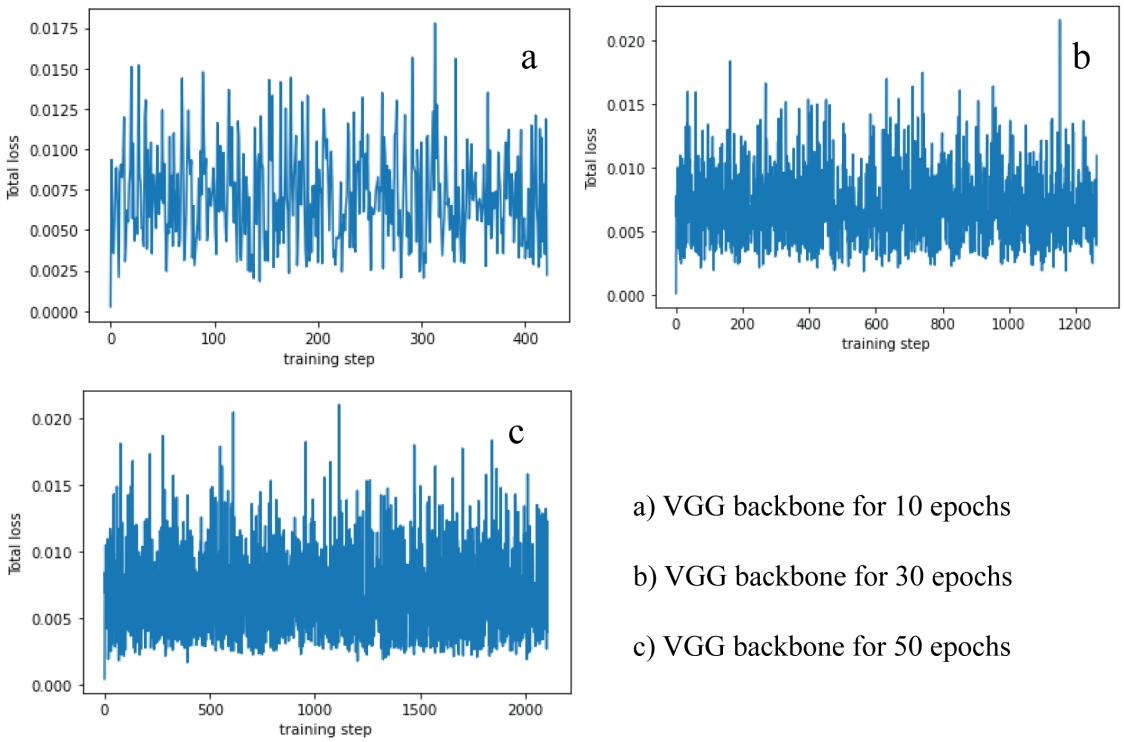


a) ResNet backbone for 10 epochs

b) ResNet backbone for 30 epochs

c) ResNet backbone for 50 epochs

Figure 4.8: *resnet for the first scales and aspect ratios*

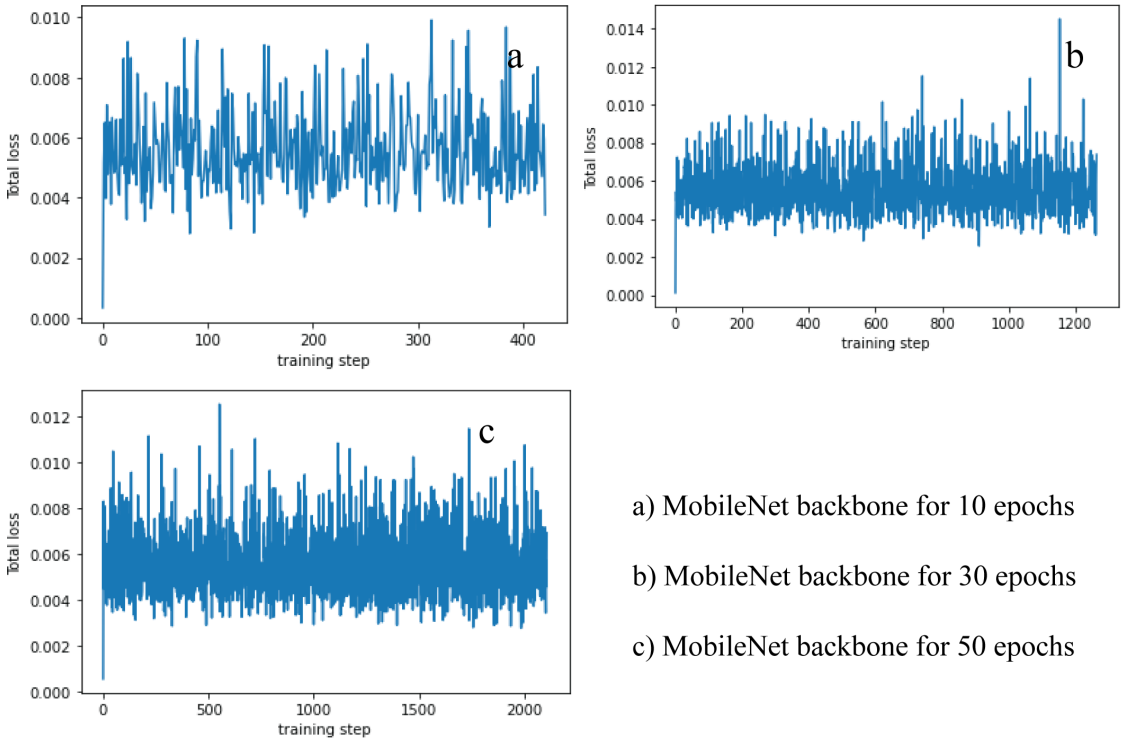


a) VGG backbone for 10 epochs

b) VGG backbone for 30 epochs

c) VGG backbone for 50 epochs

Figure 4.9: *VGG-16 for the second scales and aspect ratios*

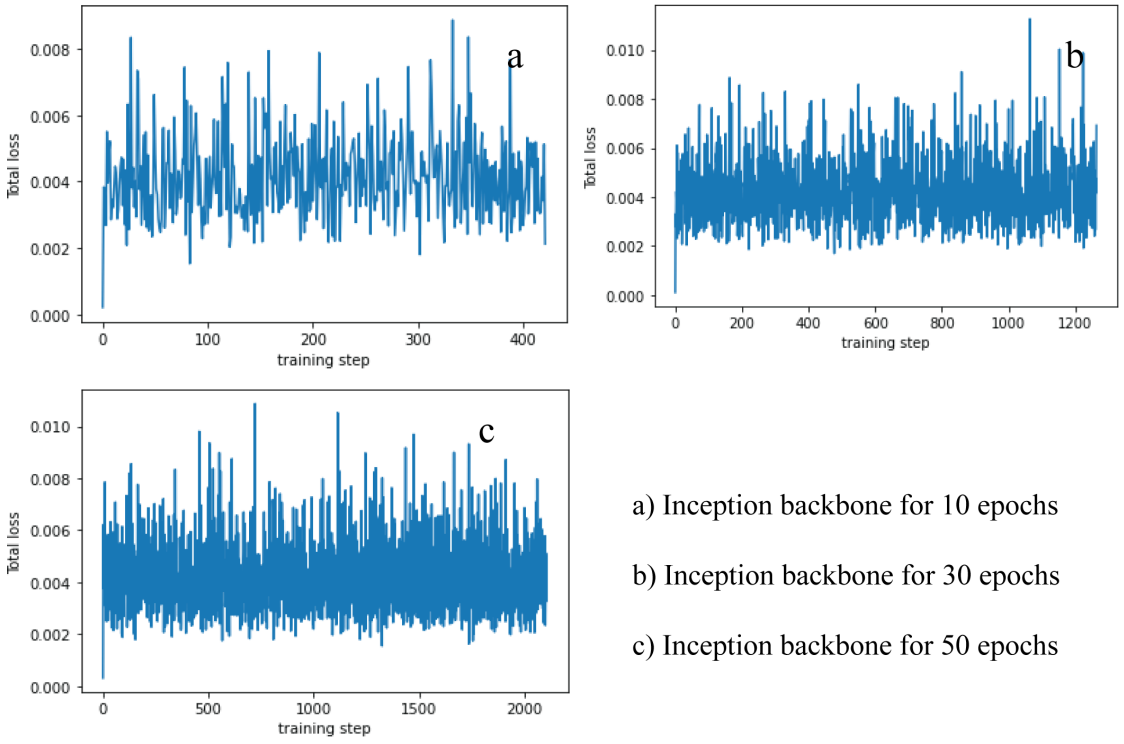


a) MobileNet backbone for 10 epochs

b) MobileNet backbone for 30 epochs

c) MobileNet backbone for 50 epochs

Figure 4.10: *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 4.11: *inception for the second scales and aspect ratios*

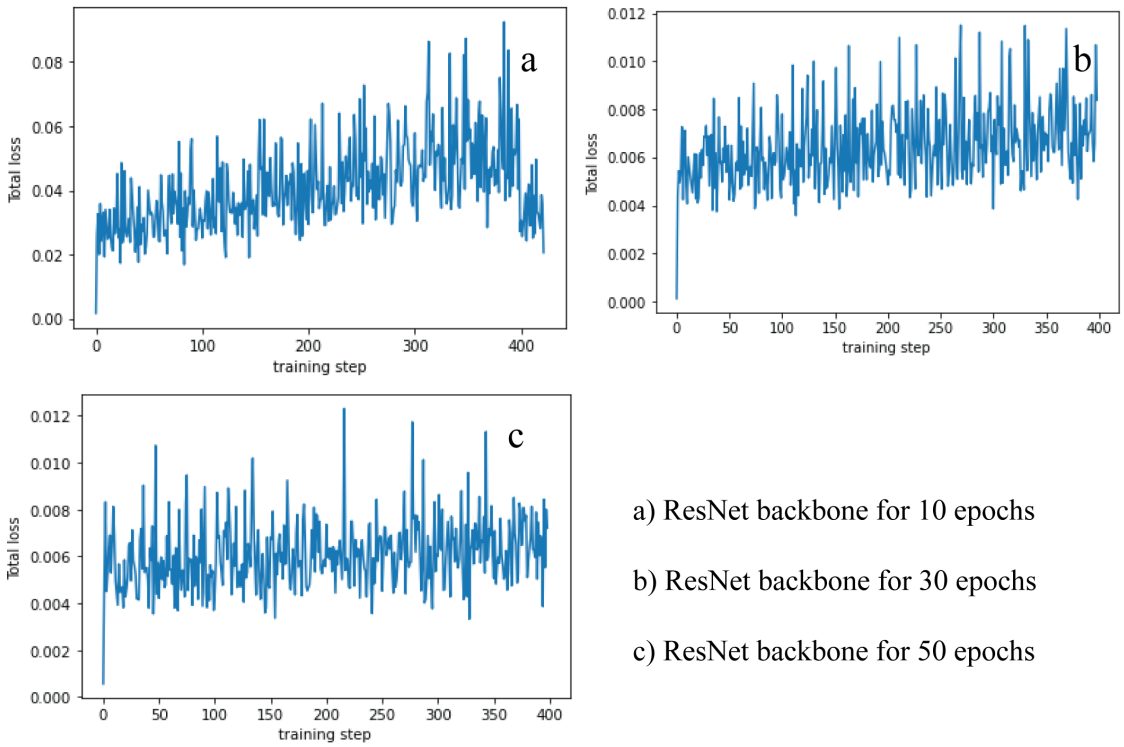


Figure 4.12: *resnet for the second scales and aspect ratios*

Table 4.1: *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 4.2: *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%

Table 4.3: *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

Digit network

For this task, 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, yielding the following tables.

Table 4.4: *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%

Table 4.5: *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 4.6: *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

The final application

The application is composed of a Python script that takes an image as input and passes it through a plate detection model then uses the output to crop the regions containing a license plate and passes them through a digit recognition model (see figure). Both models had an Inception network as a backbone because it has the best mAP. This algorithm is slightly more robust to errors that can be made by the plate detection model, because it checks the digit recognition models' output if it 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 application, the only relevant characteristics is either or not it reads the license plate number correctly. After testing on 100 test images containing 114 license plates, 96 license plates were read yielding, therefore an accuracy of **84.21%**.

4.4.2 Using YOLOv3

Module architecture

Both the plate and digits networks are based on the YOLOv3 architecture (see appendix) 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 [12]. Our networks, the plate and digit networks predicts one and ten possible classes respectively. As discussed in section, 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, will result in a large output volume which makes the training difficult and 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 [12]. This technique helps a lot when dealing with small objects.

Training configuration

The convolutional network Darknet-53 (see appendix) 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). Since the network is huge it is impractical to retrain; that would take ages to train due to low computational power available. The last fully connected layer of Darknet-53 is remove and replaced with additional convolutional layers. The exact architecture is presented in appendix. Both networks have been trained with 0.001 learning rate using the Adam optimizer with a momentum of 0.9. The plate net has been trained for 4500 iterations whereas the digits net 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 process of training. 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.

Training and testing results

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. 4.13) on the test set, 270 images where 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 97.8% (see fig. 4.14). 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 4.7 summarizes the above.

Table 4.7: *Plate network results using YOLOv3 model*

Number of Epochs	Loss	mAP
1400 epochs	0.123	97.0%
4000 epochs	0.052	97.8%

Figure 4.15 shows some examples of plate detection using our network.

Digit network

The digit network has been trained on 845 images containing 10,402 hand annotated digits. As the plate net, it has been trained for 4000 iterations at the beginning which brought the loss to 0.8205 with an mAP of 92.0% 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 93.8% 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 4.8 summarizes the above.

Figure 4.16 shows some examples of digit detection using our network.

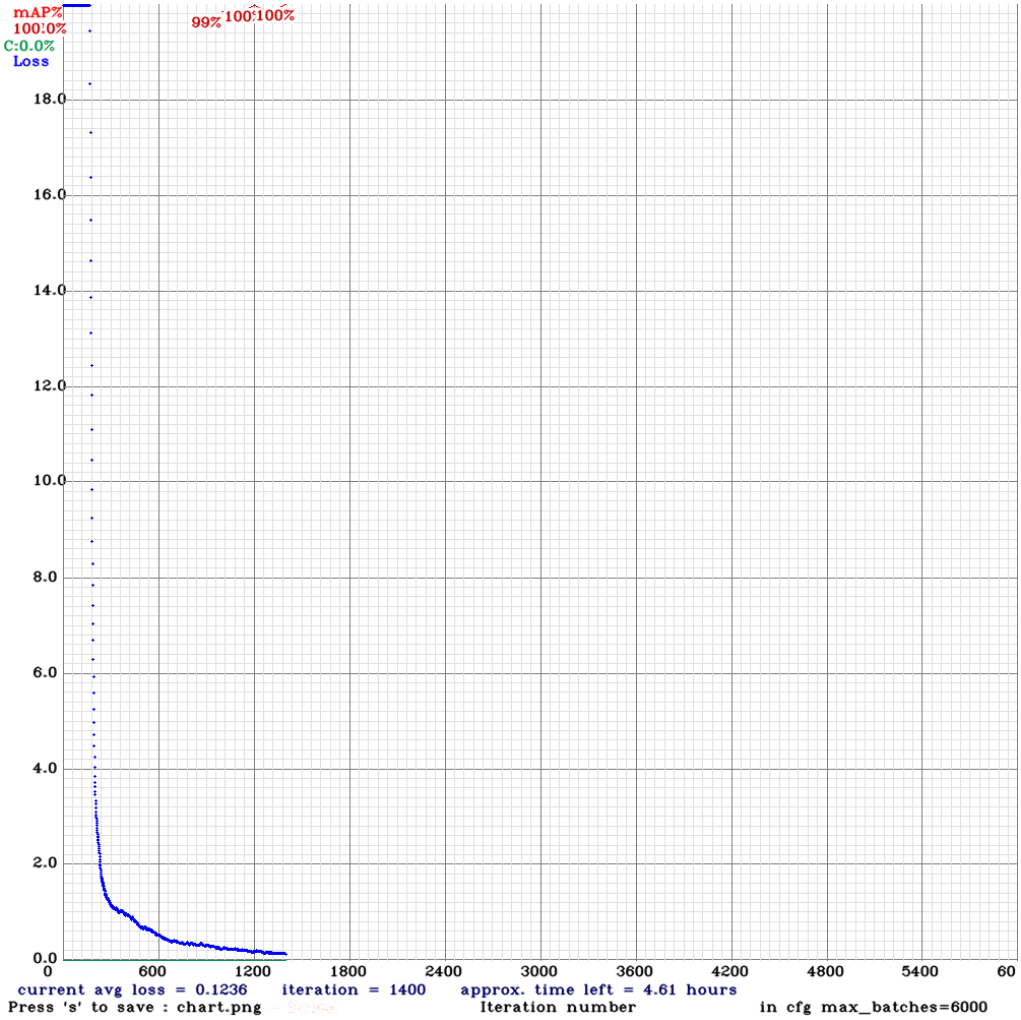


Figure 4.13: *Loss plot for the plate network after 1400 iterations*

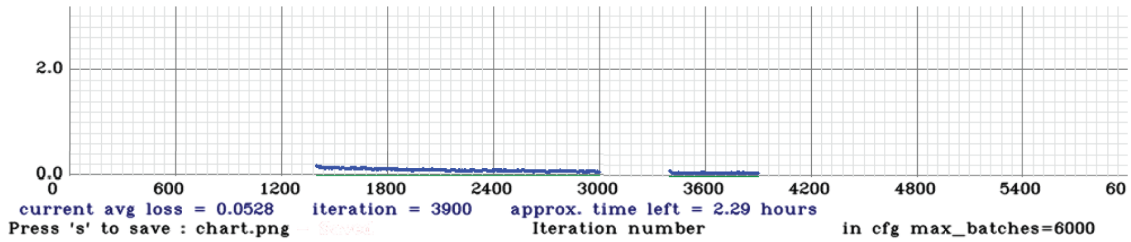


Figure 4.14: *Loss plot for the plate network after 4000 iterations. The erased blue portion was due to unstable internet connection*

Table 4.8: *Digit network results using YOLOv3 model*

Number of Epochs	Loss	mAP
4000 epochs	0.8205	92.0%
8000 epochs	0.567	93.8%

Figure 4.15: *Plate detection examples. a) Successful detection under challenging conditions b)Complete failure. This is probably due to the shape and the color of the plate which are not usual.*

Figure 4.16: *Digit detection examples. a) Complete failure. Our model struggles a lot with diagonal images due to the lack of examples. b) Successful detection in blurry images. c) Detection of a false negative in the top right corner.*

Speed testing the system

The networks have been tested locally on an i5-825u CPU. 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 by 416 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. Table below summarizes all everything discussed above.

Table 4.9: *Speed test summary*

Network	Average Time (s)
Plate network	0.41 s
Digit network	0.42 s
Entire system	0.76 s

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.

Figure 4.17: *Entire system flow example. 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).*

Chapter 5

Method and rsults

5.1 Result analysis

5.1.1 Total loss graphs

At first glance at the graphs in (cite section) 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 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, PASCAL, and IMG-NET.

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" 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.

5.1.2 mAP tabels

The mAP tabels in (section) show 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" 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.

5.1.3 Final application

All of the previous, in addition to the speed performance results, indicates that the model using Inception network as a backbone is optimal for the use in application. The accuracy of 84.21% obtained in the application testing is considered a good result for a first attempt. Although there are many ways to improve on it by working on the following :

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

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.

Appendix A

YOLOv3 network architecture

YOLOv3 predicts boxes at 3 different scales. From our base feature extractor (appendix A.1) we add several convolutional layers. The last of these predicts a 3-d tensor encoding bounding box, objectness, and class predictions. In our experiments with license plate data-set we predict 3 boxes at each scale so the tensor is $N \times N \times [3*(4+1+10)]$ for the 4 bounding box offsets, 1 objectness prediction, and 10 class predictions, representing the ten digits for zero to nine. Next we take the feature map from 2 layers previous and upsample it by $2\times$. We also take a feature map from earlier in the network and merge it with our 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. We then add a few more convolutional layers to process this combined feature map, and eventually predict a similar tensor, although now twice the size. We perform the same design 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 [12]. See fig. A.1

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.1

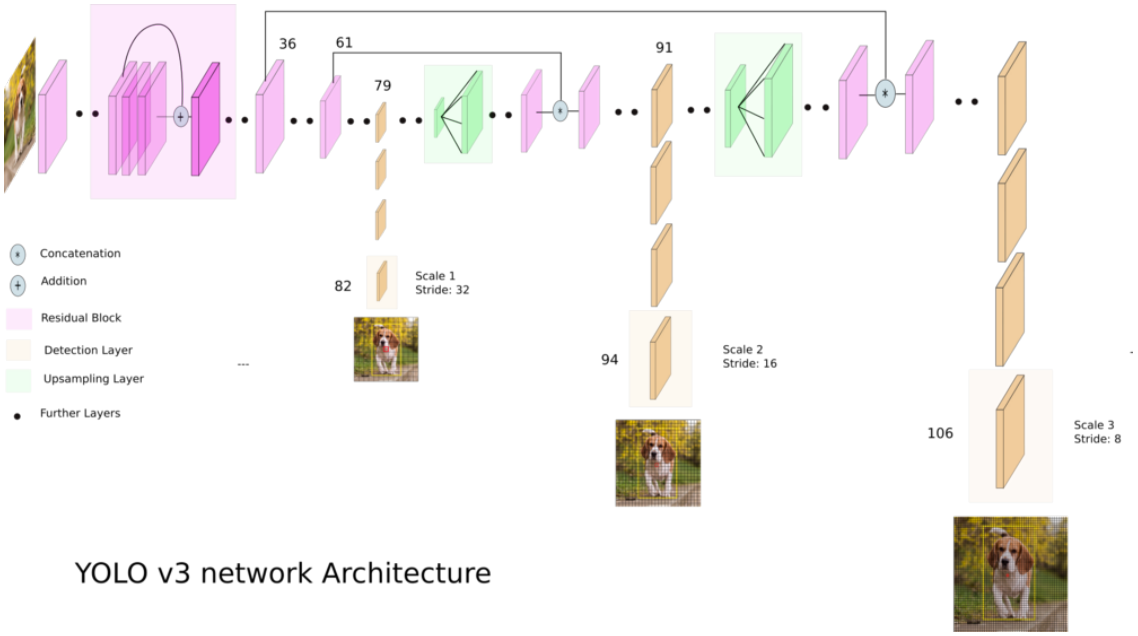


Figure A.1: *YOLOv3 network architecture*

Table A.1: *Darknet-53*

	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			

Table A.2: *Comparison of backbones. Accuracy, billions of operations, billion floating point operations per second, and FPS for various networks.*

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**

The above 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.2

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 [12].

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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