Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Object detection with YOLO v3

Tesi di laurea triennale

Supervisor
Prof. Alessandro Sperduti

Candidate
Francesca Lonedo

Academic year 2017-2018

To my parents, who gave me the opportunity to graduate twice.

Abstract

The following is a description of the work done by the candidate Francesca Lonedo during her three-hundred-and-twenty hours long curricular internship at THRON S.p.A.
Object detection is a computer vision related technology that deals with identifying instances of semantic class objects, and locating their position in digital images and videos. It has application in various domains, such as self-driving cars, video surveillance and image retrieval.
My project's goal was to develop a proof-of-concept prototype application to detect class instances and their location in image data. There are different approaches to solve such task, but I was asked to follow a *deep learning*[g]method that uses a *convolutional neural network*[g]called *yolo v3*[g]. Furthermore, it was required to use a production-ready framework for development, and I was given the freedom to experiment with both *Mxnet* and *Tensorflow* and to choose whichever I found more suitable for the project.

"There is nothing either good or bad, but thinking makes it so"

— William Shakespeare's Hamlet

Thanks

I want to thank my parents, for their endless love and support. They accepted my weird interests and gave me the opportunity to follow my dreams and graduate twice.

I want to thank my partner, who re-introduced me to computer science. He taught me a lot of things and always spoils me.

I want to thank my friends, old and new, who kept me company on this journey.

I want to thank my tutor and everyone at THRON for making my internship a wonderful learning experience.

Padova, December 2018 Francesca Lonedo

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduction

1.1 THRON

THRON S.p.A.[1] is an Italian company that develops a *marketing DAM*[g]software. Founded in 2000 as New Vision by CEO Nicola Meneghello and CTO Dario De Agostini, it was one of the pioneers at delivering web applications in Italy.
In 2004 the software house launched 4ME, a cloud based service for content management which was the precursor of the current company-homonym product.
Nowadays THRON has four premises around the globe and innovation remains one of its core values.
In the following I will refer to the company as THRON and to the product as Thron to distinguish them.

1.2 Birth of the project

THRON's homonymous product is a *marketing DAM*, a software application that can be seen as a centralized archive to manage multimedia content; any type of file can be stored, but contextually images are predominant. A DAM provides various features such as advanced item search, property management and tools to share content on social media and web platforms.
The core feature of a DAM is its ability to categorize the indexed media, making it easy for the user to find and edit objects with specific properties or tagged in a certain way. Thron makes this process even simpler thanks to its intelligence, a system that automatically elaborates the inserted media and suggests how to tag and categorize it.
Currently Thron's intelligence is powered by general purpose *artificial intelligence*[g]tools, hence the ability to recognize only a limited set of classes/features. Since most of Thron's users are big brand-name companies, teaching the system how to detect custom classes relevant for each client would improve greatly the software's relevance on the market.
As stated above, the vast majority of the stored media is in an image format; this both determines the problem and its solution: what is needed in Thron is an object detection system, and given the considerable database already available the most straightforward

[1] https://www.thron.com

way to create it is to follow a deep learning[2] method.

1.3 Content organization

Il secondo capitolo descrive ...

Il terzo capitolo approfondisce ...

Il quarto capitolo approfondisce ...

Il quinto capitolo approfondisce ...

Il sesto capitolo approfondisce ...

Nel settimo capitolo descrive ...

This works adheres to the following typographic conventions:

* acronyms, abbreviations, ambiguous and field-specific terms are defined in the glossary at the end of the document;
* the first occurrence of glossary terms is emphasized as follows : *glossary term*[g];
* technical terms are emphasized as follows: *technical term*.

[2] I will exhaustively explain what deep learning is and why it's the best solution for this problem in chapter two.

Capitolo 2

Deep learning

In this chapter I will dive deep (no pun intended) in the reasons behind the choice of a deep learning approach to detect custom-class objects in image data.

2.1 Overview

Deep learning is a part of the broader family of *machine learning*[g]methods that focuses on learning features of interest by experiencing them thought data samples. In contrast to *machine learning*, which still needs the problem's layout to be described by a rather complex human-coded algorithm, a *deep learning* model only needs a dataset to autonomously learn from. Thanks to the ever-growing availability of computational power and labeled data, and the relative simplicity in which a network can be trained, the deep learning approach to solve artificial intelligence problems is flourishing, bringing artificial intelligence to everyday life.
At the core of every *deep learning* model there is an artificial neural network, whose architecture is strongly inspired by the structure and functioning of a biological brain. Basically, an artificial neural network is composed by a variable number of neuron layers densely connected and stacked upon each other, hence the name *deep neural networks*. The information is elaborated by traveling from layer to layer until the top is reached.
As stated, each layer is composed of computational units that work like *neurons*: each *neuron* receives an input, and using an activation function it produces an output that propagates to activate a *neuron* in the following layer. Actually each *neuron*'s output is influenced by weights and thresholds that need to be adjust to achieve the best performance. When learning, the network indeed adjusts the weights and thresholds that activate each *neuron*'s function, so that it can get closer to the wanted result in an iterative manner. After each passage through the network, a *loss function*[g]is calculated to determine how well it is performing; then a *gradient descent function*[g]derives how this performance can be improved by decreasing the loss and accordingly *back-propagation*[g]is performed to adjusts the weights used by the neurons in their activation functions.

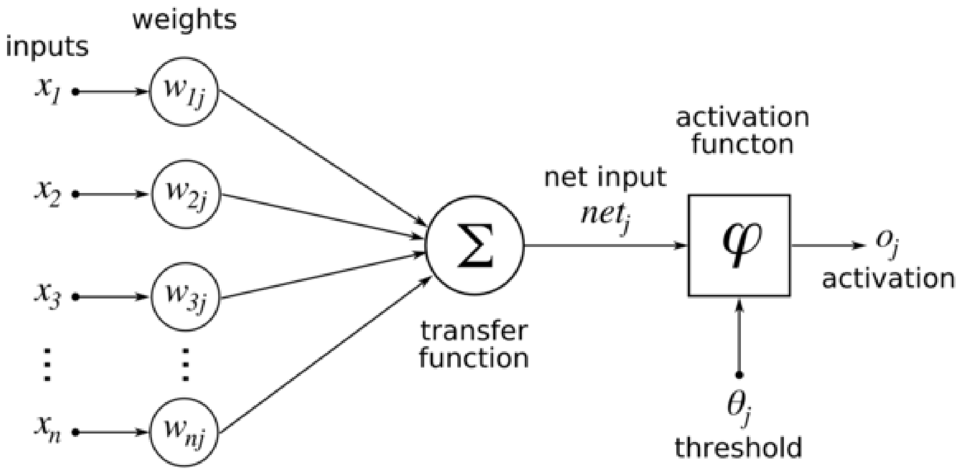


Figura 2.1: Visual representation of the activation function that determines a neuron's output and the parameters that influence it.

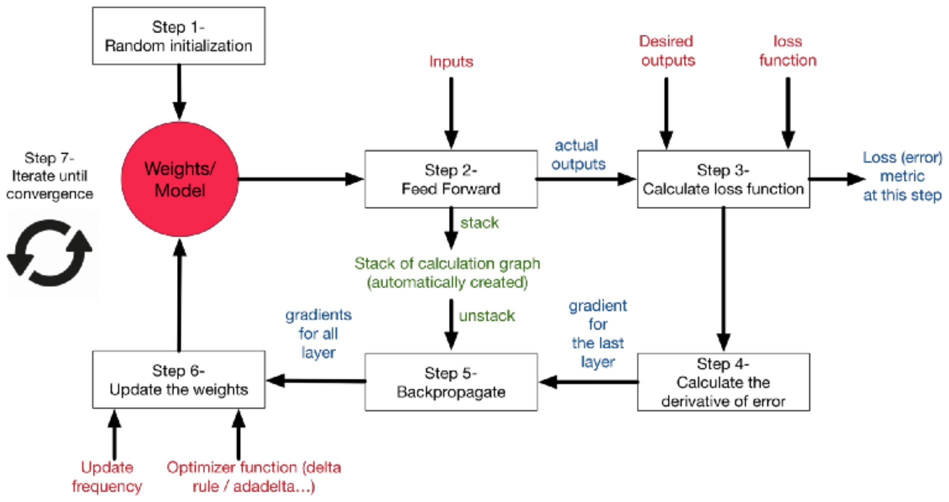

Step 1- Random initialization
Weights/ Model
Step 7- Iterate until convergence
Inputs
Step 2- Feed Forward
actual outputs
Desired outputs
loss function
Step 3- Calculate loss function
Loss (error) metric at this step
stack
Stack of calculation graph (automatically created)
unstack
gradient for the last layer
gradients for all layer
Step 4- Calculate the derivative of error
Step 5- Backpropagate
Step 6- Update the weights
Update frequency
Optimizer function (delta rule / adadelta...)

Figura 2.2: Schematic representation of how a network updates the weight to decrease the loss and achieve better accuracy.

2.2 Deep neural networks

As mentioned above, the *deep learning* approach to solve a problem is to train a *deep neural network* to recognize the wanted features so it can later make predictions about them. Basically when you train a *deep neural network* you feed it thousands of data samples, so it can learn from the wanted data representation by "seeing" it, just like you would show a child pictures of kittens to teach him what a kitten is. There are two different ways to train a network, each one with different goals and contexts of use:

∗ supervised training: the network is given a labeled *dataset*[g]to learn from; for each data sample the ground truth is provided, so the network knows what it is learning. This is the preferred approach whenever labeled data is available.

∗ unsupervised training: the network is given an unlabeled *dataset* to learn from; no ground truth is given, so the network has to identify patterns and divide them into different categories on its own. This approach is preferred when the goal is to group data in categories or when data is too complex to be labeled.

There are different network architectures, each one dedicated to a specific type of problems. When solving *computer vision*[g]task the best performances are held by a class of networks called *convolutional neural networks.*

2.2.1 Convolutional neural networks

convolutional neural networks (CNNs) are *deep neural networks* that consist of an input layer, an output layer, and a variable number of hidden layers with different purposes. What names this family of networks is in fact on particular type of layer, called *convolutional layer*; a single network typically contains various *convolutional layers* of different size, and each perform a *convolution* operation that filters its input stimuli before passing them the to next layer in order to reduce the parameters number, thus allowing the network to be deeper.
CNNs, among others, are mainly used to perform computer vision tasks; in particular they achieve good results in object detection tasks.

2.2.2 Object detection

Object detection is a computer vision problem that concerns the identification of class instance objects in an image (or video), and locating the actual position of said object in the picture. Commonly the spacial orientation of the detected object is framed by a rectangular *bounding box* that determines its height and width. Thus, object detection is far more powerful than mere image classification, not only because it "draws" a box where the object is located, but also because it can identify multiple object instances in a single image, while classification models have the limit of labeling only the one predominant object in the scene. The capability of labeling and locating multiple instances opens object detection models to a new set of application, such as video surveillance (e.g. moving subjects can be tracked) and instance counting for industrial purposes (e.g. counting boxes in a warehouse). Furthermore, object detection has proven to be more reliable than classification to scan images where the subject of interests occupies only a small part of the picture (e.g. a street sign in the corner).
Commonly an *object detection* model is actually built on top a classifier that works

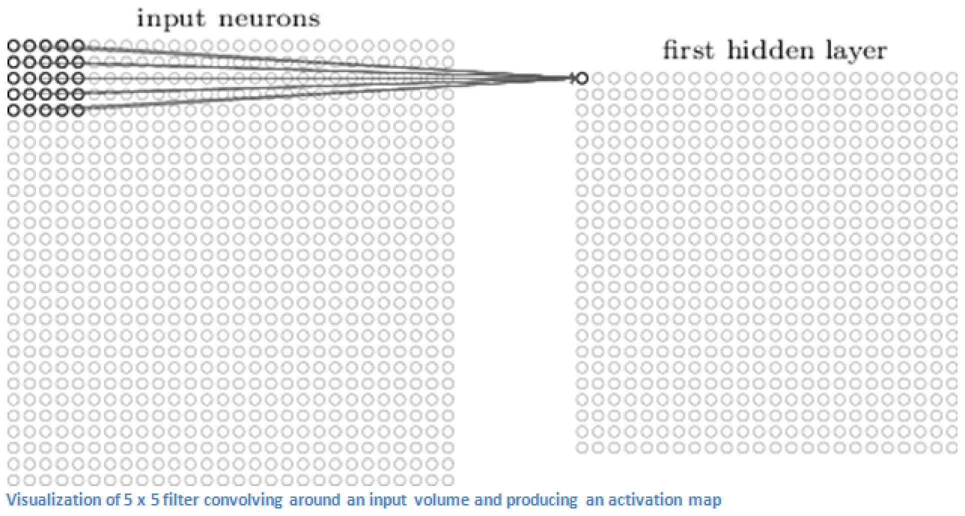

input neurons
first hidden layer
Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

Figura 2.3: Visual representation of a convolutional layer.

as a *feature extractor*, but this exudes the topics of my project and I am introducing this notion only because it is noticeable in the naming convention of model networks, which are indeed important in my work.
There are three popular network architectures for object detection:

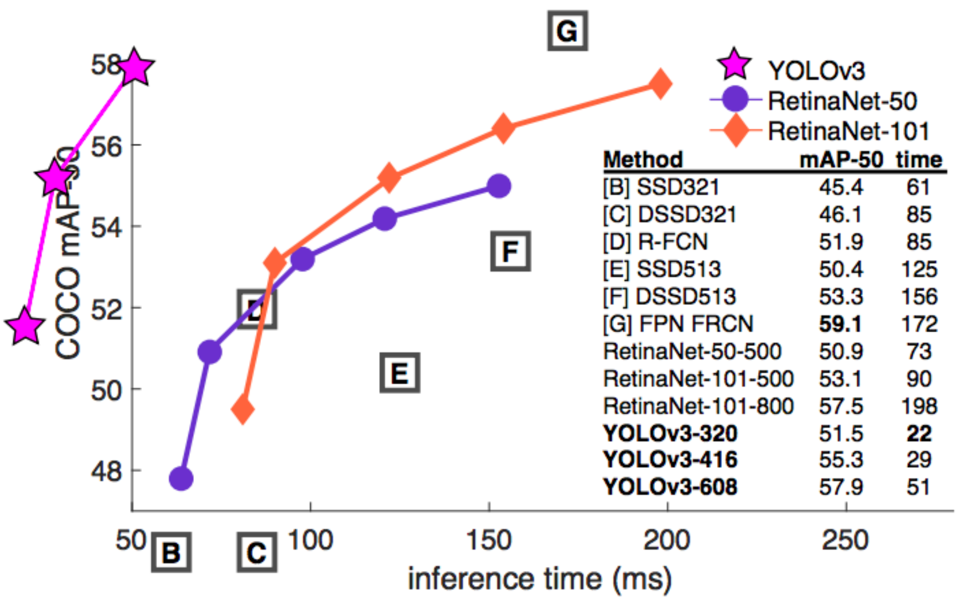
* SSD (Single Shot Detection);
* R-CNN (Region-based *convolutional neural network*, and its upgrades Fast R-CNN and Faster R-CNN);
* YOLO (You Only Look Once).

While both *SSD* and *YOLO* follow the same approach of computing the image data through a single network, *R-CNN* splits the image in various regions trying to guess where objects of interest might be and processes each area separately, thus requiring more computational power. State-of-the-art models share comparable performances in accuracy, but due to the time overhead of processing a picture multiple times, *R-CNN* is a bit slower than *SSD* and *YOLO*.

2.2.3 YOLO v3

YOLO[1] is a *convolutional neural network* built on top of *Darknet* classifier, now on its third version *YOLO v3*. Both *YOLO* and *Darknet* are developed by Joseph Redmon on his C-written open source neural network framework Darknet[2].
What makes *YOLO* stand out compared to its competitor is its speed: at par of accuracy YOLO can perform inference in less than half the time, achieving the ability to track moving subjects in real time in 30fps videos using a gaming-tier GPU[3].
For my project I was asked to used *YOLO* in its newest version, *YOLO v3*. As I

[1] https://pjreddie.com/darknet/yolo/
[2] To avoid misunderstandings I will call Darknet the framework and *Darknet* the classifier.
[3] Real time inference is possible on Nvdia GeForce Titan X GPU.



Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

Figura 2.4: Comparison on object detection performances on Nvidia Titan X GPU. A higher mAP value means a better accuracy.

already said, *YOLO* was firstly implemented in Joseph Redmon's Darknet framework, which is written in C so it is fast, but unfortunately it is suitable only for research purposes. Since I was asked to develop a production-ready application, I had to choose another framework to work on, and I will discuss about it in later chapters.

Structure

YOLO v3 is an *convolutional neural network* for *object detection* built on top of *Darknet53*, an image classifier that uses 53 convolutional layers. With *Darknet53* as its feature extractor, *YOLO* processes every image once, looking at it as a whole, hence encoding extra information about the context each class lives in. To do so it models detection as a regression problem, dividing the image into an S $\times$ S grid, and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities; when a class-object's center falls inside a grid cell, that grid cell is responsible of detecting that object.

2.3 Training a network

Training a *convolutional neural network* is a computation-heavy task, that involves the processing of thousands of pieces of data, which in the context of computer vision is in the format of images. Therefore best way to process thousands of images is to use a GPU; using a GPU (or even clusters of GPUs) it is possible to process the images in parallel among the *cores*, speeding up the process greatly. As a comparison, while training on CPU would take days even on a small dataset, on a GPU all the work would be done in just few hours.

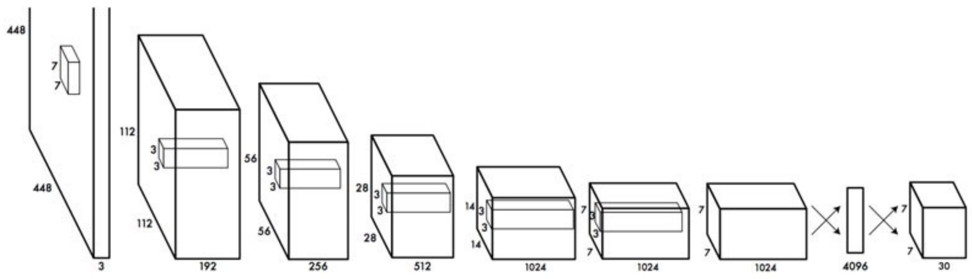


Figura 2.5: Visual representation of *YOLO* network layers in its first version.

2.3.1 Dataset

The first and most important step to train a network is to create the *dataset* it will learn from. Note that the quality of the dataset itself will have great influence on the final accuracy of the network. For instance you will want to feed your network examples of your objects of interest from every perspective and immerse in their context; also, when training a network to perform an *object detection* task, you want your *bounding boxes* to be as accurate and tight around the figures as possible.
Clearly, creating a custom dataset is a demanding job, since piece of data must be manually labeled by a human. For my project I was given a custom dataset created by one of my colleagues, containing classes relevant for one of THRON's clients. Creating a *dataset* exudes the goals of my project, since on production it will be possibly performed by clients[4].
When creating an *object detection dataset* there are there are two main qualities to determine its goodness:

∗ *Cardinality*, which is the means of the number of labels per image;

∗ *Density*, which is the cardinality divided by the number of classes.

A density too low has negative influence on the final accuracy of the network, because basically you are trying to teach it to recognize a high number of classes, but you're giving it only a few examples for each. On a lesser degree, a cardinality too high has a negative influence on the final accuracy as well because the examples you are giving are too cluttered with information. Furthermore, when creating your own *dataset*, you might want to insert a similar amount image samples/labels for each class.
After the full dataset is created, it is common practice to divide it into two separate *datasets*: a *training dataset* and a *validation dataset*, with the suggested ratio of 70% training and 30% validation. Note that these two *datasets* contain different samples to avoid *overfitting*[g].

[4]As explained in the introduction, THRON's product is a DAM, which in this context can basically seen as a database of categorized images; with possibly minor changes to data labeling (e.g. introduction of bounding boxes for the objects of interest) and database export, image data can be easily turned into a dataset for computer vision.

2.3.2 Training

Once the dataset is ready and formatted in the network's preferred format[5], the training process can begin.
The training process is conceptually simple; as previously stated, teaching a machine to recognize kittens can be compared to showing pictures to a child and pointing at the kitten in them to make him understand what a kitten is. When doing so with a child you only need a few examples, when with a machine you'll probably need a few hundreds. The next step is to check whether the machine truly learned what a kitten is; again this can be roughly compared to checking whether the child truly learned what a kitten is by giving him a picture and asking him to point at the kitten, if there is one.
Now that we talked about kittens and children we can explore in detail how a machine learns the way a human would. The training process is a loop in which the network is "shown" the *dataset* it should learn from; since a *dataset* is composed by thousands of images, it can't be entirely loaded into memory and must be split in smaller units for processing, called *batches*; in practice maximum *batch* size is determined by your hardware specifics. Your network will be fed all of the training *dataset batch* after *batch*. Each loop of computation on the whole training dataset is called an *epoch*[g]. To train your network you will want to perform at least a couple of hundreds epochs; this means that to learn the *dataset* your network will go through all of it hundreds of times.

2.3.3 Validation

Every fixed epoch number you will want to check your network's learning progress; to do so you will perform a validation loop, which means that you will feed the network your validation *dataset* and compare the network's prediction with the ground truth. There are various metrics to calculate a network's accuracy, and the one I used on my project is called *mean average precision*[g](mAP). This evaluation metric uses an *intersection over union*[g](IoU) threshold to determine how well the detected positives overlap with the ground truth; setting the IoU threshold higher or lower influences the wanted precision at locating objects[6], and I set it to a value of 0.5.

2.3.4 Transfer learning

Training a network from scratch is a quite demanding operation in terms of computational time and power, because the weights for each class need to be calculated from a random initialization. Luckily, it is possible to train a network to recognize new custom classes starting from pre-trained weights for other (general) classes, and this process is called *transfer learning*.
Basically when you apply transfer learning on a pre-trained network[7], you teach it to recognize new classes by adjusting its weights (which have a meaning and are not random anymore) and the structure of the two layers that determine the final output. To benefit from *transfer learning* you want to choose a base network with weights for

[5]Depending on the framework, the preferred format changes. To work with YOLO v3 on MXNet data can provided in both RecordIO format or LST + Jpeg format.
[6]Note that ground truth bounding boxes are hand-drawn by a human so they already come with an error, thus setting the IoU threshold too high could prove to be useless and even counterproductive.
[7]All the major deep learning frameworks usually provide networks and weights pre-trained on popular datasets such as COCO, Pascal VOC or ImageNet.

classes that share some features with your custom ones for example if you want to be able to detect different species of animals, a good choice would be to *fine-tune* a network trained on a datasets that already contains cats and dogs.

2.3.5 Hyperparameters

The accuracy of a *convolutional neural network* is influenced by various paramenters, called hyperparameters, that tweak how the weights are updated and how the results are evaluated. This values come in the form of *"magic numbers"*, meaning that for every dataset each network has its own values to optimize the final performance and that they can be found only in a trial and error fashion[8]. Since this is rather time consuming, especially when training on CPU, I couldn't tune the hyperparameters for my project. However I will introduce briefly what the main hyperparameters do on a *YOLO v3* architecture network.

* Learning rate. Speed at which the network learns. It is used when updating weights and a learning rate too high might cause overfitting.
* Learning rate deacay rate. As the network learns, its learning rate decreases; this value is important to avoid overfitting.
* Momentum. Stochastic gradient descent momentum, used by the gradient descent function to calculate how to reduce the loss.

2.4 Object detection for THRON

As mentioned in Chapter 1, THRON already uses generic classification services in its *DAM*. Therefore, to provide a better service, it comes natural that the Company has interest in developing a system able to detect custom classes.
Since THRON already has a vast database of labeled data, following a deep learning approach is the easiest option, even more so now that framework providing almost out-of-the-box are increasing prominence.
At first it was still unclear whether the Company needed just a classifier, but with my proof-of-concept the benefits of an *object detection* model became clear. We particularly chose *YOLO v3* under the suggestion of a data scientist working in my team, since he already developed a research-purpose prototype application using *YOLO v2*. Due to the technology used, his application couldn't be released in a production environment, so it was my job to develop a production-ready application using *YOLO*.

[8]To determinate the best values it is common practice to re-train the network various times and compare the results.

Capitolo 3

Internship description

My project at THRON S.p.A. was to develop a production-ready application for object detection *using* YOLO v3. *I was given the responsibility to choose the framework to use, and I coded both the training and inference parts of the application.*

3.1 Project introduction

The goal of my project was to develop a production-ready *object detection* application that could be integrated in Thron in the future. Since THRON strongly relies on web services like those provided by *AWS*[g], so it was my job to keep compatibility.

3.2 Risk analysis

Every project comes with its risks. Since deep learning is a relatively new field, my project was at high risk from the beginning.

Decription	**Solution**	**Occurrence**
Stable release The technology I used is fairly new, so it is not surprising that the main toolkit I used was in version 0.3.0 (version 0.4.0 was released later on my internship). This lead to some inconveniences such as bugs and inconsistencies between documentation and source code.	No real solution was identified. When necessary I submitted issues to the code developers and patched bugs myself when I could.	High
AWS support Since the technology I used was fairly new, it relied on the newest version of my framework of choice. During my AWS provided out-of-the-box support only to the previous version of my chosen framework, which isn't compatible with the core toolkit I used.	No real solution was identified. I tried other AWS services, but they are not as straightforward as the one I was supposed to use and proved to be rather troublesome.	High
Hardware Deep learning is a demanding technology that requires a lot of computational power. Specifically, it is suggested to execute training on a GPU since it is much faster, but I didn't have one. Training on CPU, on the other hand, is indeed possible, but it requires fairly new hardware, which, during my first week, I didn't have.	Before I was assigned a newer CPU I got to work on a remote AWS machine via SSH. This machine provided a GPU so I could test example algorithms. When given a newer CPU I had to train my network on that, which required a couple of days and thus prevented me from tuning the hyperparameters.	High
Python Deep learning frameworks are written almost exclusively in Python. With no prior experience, writing a Python application could prove to be troublesome.	None of my coworkers was a Python expert, but they always helped me to find good libraries and frameworks to use.	Medium

3.3 Requisiti e obiettivi

3.4 Pianificazione

Capitolo 4

Analisi dei requisiti

Breve introduzione al capitolo

4.1 Casi d'uso

Per lo studio dei casi di utilizzo del prodotto sono stati creati dei diagrammi. I diagrammi dei casi d'uso (in inglese *Use Case Diagram*) sono diagrammi di tipo Unified Modeling Language (UML) dedicati alla descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso. Essendo il progetto finalizzato alla creazione di un tool per l'automazione di un processo, le interazioni da parte dell'utilizzatore devono essere ovviamente ridotte allo stretto necessario. Per questo motivo i diagrammi d'uso risultano semplici e in numero ridotto.

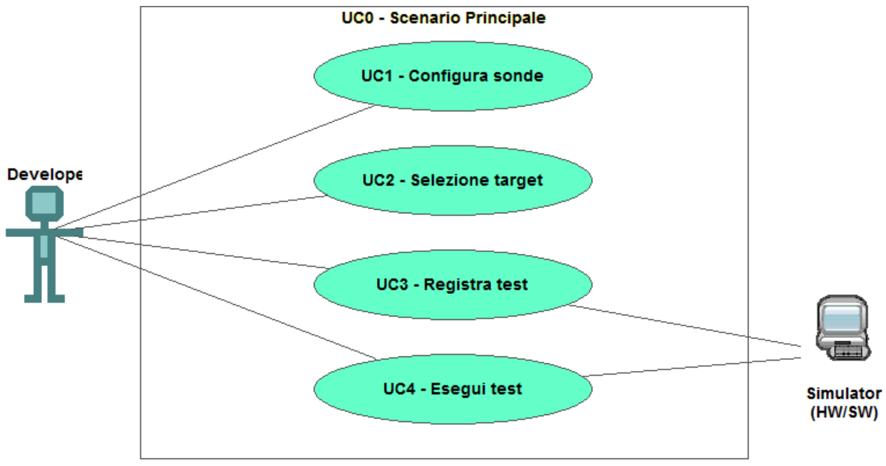


Figura 4.1: Use Case - UC0: Scenario principale

UC0: Scenario principale

Attori Principali: Sviluppatore applicativi.

Precondizioni: Lo sviluppatore è entrato nel plug-in di simulazione all'interno dell'IDE.

Descrizione: La finestra di simulazione mette a disposizione i comandi per configurare, registrare o eseguire un test.

Postcondizioni: Il sistema è pronto per permettere una nuova interazione.

4.2 Tracciamento dei requisiti

Da un'attenta analisi dei requisiti e degli use case effettuata sul progetto è stata stilata la tabella che traccia i requisiti in rapporto agli use case.
Sono stati individuati diversi tipi di requisiti e si è quindi fatto utilizzo di un codice identificativo per distinguerli.
Il codice dei requisiti è così strutturato R(F/Q/V)(N/D/O) dove:

R	=	requisito
F	=	funzionale
Q	=	qualitativo
V	=	di vincolo
N	=	obbligatorio (necessario)
D	=	desiderabile
Z	=	opzionale

Nelle tabelle 4.1, 4.2 e 4.3 sono riassunti i requisiti e il loro tracciamento con gli use case delineati in fase di analisi.

Tabella 4.1: Tabella del tracciamento dei requisti funzionali

Requisito	Descrizione	Use Case
RFN-1	L'interfaccia permette di configurare il tipo di sonde del test	UC1

Tabella 4.2: Tabella del tracciamento dei requisiti qualitativi

Requisito	Descrizione	Use Case
RQD-1	Le prestazioni del simulatore hardware deve garantire la giusta esecuzione dei test e non la generazione di falsi negativi	-

Tabella 4.3: Tabella del tracciamento dei requisiti di vincolo

Requisito	Descrizione	Use Case
RVO-1	La libreria per l'esecuzione dei test automatici deve essere riutilizzabile	-

Capitolo 5

Progettazione e codifica

Breve introduzione al capitolo

5.1 Tecnologie e strumenti

Di seguito viene data una panoramica delle tecnologie e strumenti utilizzati.

MXNet

Descrizione Tecnologia 1.

TensorFlow

Descrizione Tecnologia 2

Docker

Descrizione Tecnologia 2

OpenAPI + Swagger

Descrizione Tecnologia 2

Flask + Connexion

Descrizione Tecnologia 2

AWS

Descrizione Tecnologia 2

5.2 Ciclo di vita del software

5.3 Progettazione

Namespace 1

Descrizione namespace 1.

Classe 1: Descrizione classe 1

Classe 2: Descrizione classe 2

5.4 Design Pattern utilizzati

5.5 Codifica

Capitolo 6

Verifica e validazione

Capitolo 7

Conclusioni

7.1 Consuntivo finale

7.2 Raggiungimento degli obiettivi

7.3 Conoscenze acquisite

7.4 Valutazione personale

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia