s

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*$_{\mathrm{g}}$ method that uses a *convolutional neural network*$_{\mathrm{g}}$ called *YOLO v3*$_{\mathrm{g}}$. Furthermore, it was required to use a production-ready framework for development, and I was given the freedom to experiment with both *MXNet*$_{\mathrm{g}}$ and *TensorFlow*$_{\mathrm{g}}$ 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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 THRON

Figure 1.1: THRON logo.

THRON S.p.A.[1] is an Italian company that develops a *marketing DAM*$_{\text{g}}$ 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]https://www.thron.com

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*$_{\text{g}}$ 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 way to create it is to follow a deep learning[2] method.

1.3 Content organization

The second chapter briefly covers the theory behind *deep learning* and explains why it proved to be the best approach to build a custom *object detection* application for Thron.

The third chapter covers the goals of my project, showing its requirements and the work schedule.

The fourth chapter briefly showcases the technologies I used and explains the reasons why I chose them. Furthermore, the chapter it covers my main design choices and application architecture.

The fifth chapter explains my choices on the matter of tests and validation.

The sixth chapter compares my initial goals to what was actually achievable and explains the problems that occurred during development.

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*$_{\text{g}}$;
* 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.

Chapter 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*$_{\mathrm{g}}$ 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 into 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*$_{\mathrm{g}}$ is calculated to determine how well it is performing; then a *gradient descent function*$_{\mathrm{g}}$ derives how this performance can be improved by decreasing the loss and accordingly *back-propagation*$_{\mathrm{g}}$ is performed to adjusts the weights used by the neurons in their activation functions.

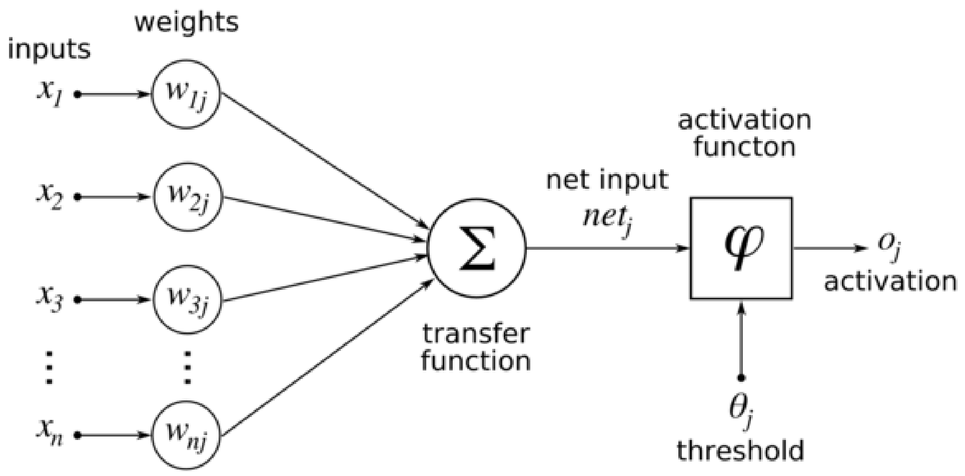


Figure 2.1: Visual representation of the activation function that determines a neuron's output and the parameters that influence it.

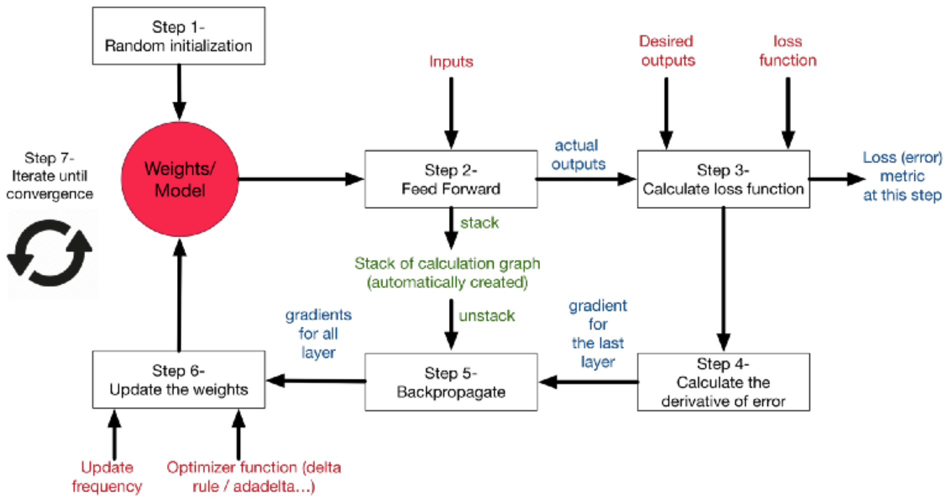


Figure 2.2: Schematic representation of how a network updates the weight to decrease the loss and achieve better accuracy.

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*$_{\mathrm{g}}$ 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*$_{\mathrm{g}}$ 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

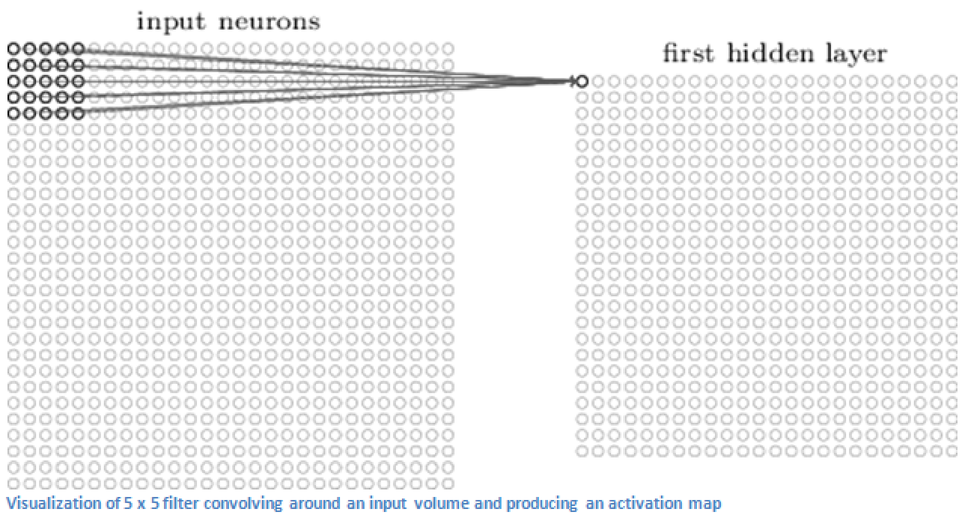

input neurons
first hidden layer
Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

Figure 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:

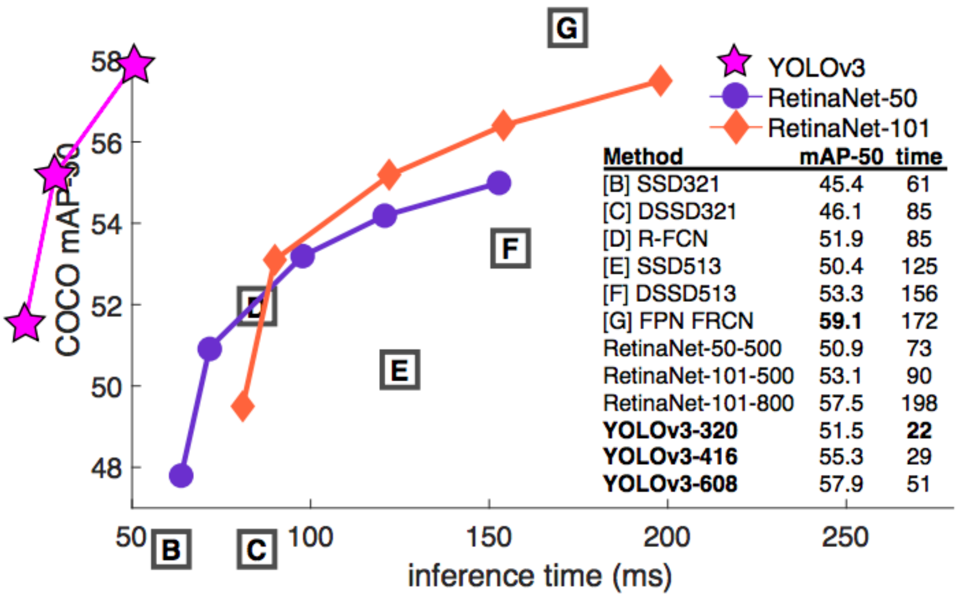
- SSD (Single Shot Detection);
- R-CNN (Region-based *convolutional neural network*, and its upgrades Fast R-CNN and Faster R-CNN);
- YOLO (You Only Look Once).

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

Figure 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 × 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.

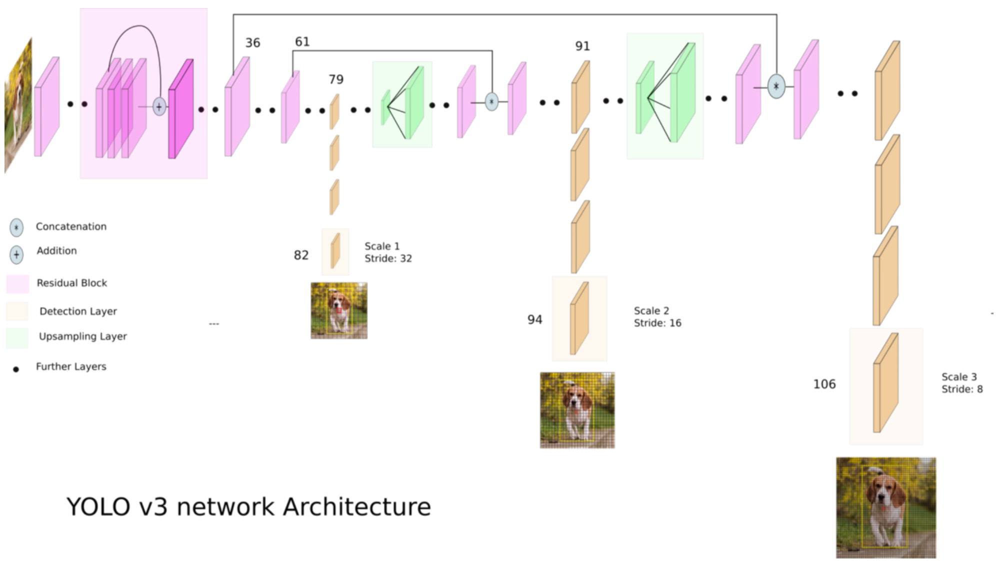


Figure 2.5: Workflow of how a *YOLO v3* network processes an image, making it pass through its layers.

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

[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.

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*$_{\text{g}}$.

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*$_{\text{g}}$. 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*$_{\text{g}}$ (mAP). This evaluation metric uses an *intersection over union*$_{\text{g}}$ (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

[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.

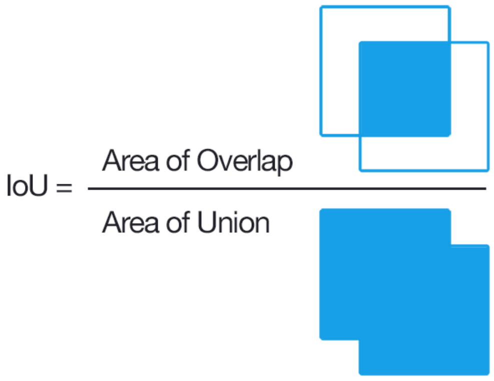


Figure 2.6: Graphic representation of *intersection over union.*

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 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.

[7]All the major deep learning frameworks usually provide networks and weights pre-trained on popular datasets such as COCO, VOC or ImageNet.

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

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*.

Chapter 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*$_{\mathrm{g}}$, 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, the probability my project would incur into some issues during development was high.

3.3 Goals and requirements

Requirements are categorized with the following criteria:

* MUST, the functionality is absolutely required;
* SHOULD, the functionality is recommended;
* MAY, the functionality is optional.

Due to the nature of the project, whose purpose was to explore a new technology and determine the possibility of deploying an application in a production environment, requirements and goals changed along the way. In fact, some requirements proved impossible to be satisfied because of the unavailability of tools and services supporting them yet.

* Required:

Table 3.1: Risk analysis.

Decription	**Solution**	**Occurrence**
Stable releases unavailable The technology I used is fairly new, so it is not surprising that the main toolkit I used, *Gluon CV*$_g$ was actually in pre-release version 0.3.0. This lead to some inconveniences such as bugs and inconsistencies between build and documented 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 *MXNet* 1.3.0. During my internship, AWS provided out-of-the-box support only to *MXNet* 1.2.1 and prior, which isn't compatible with *Gluon CV* 0.3.0.	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 *CUDA*$_g$ 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 to run at all, 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

Table 3.2: Activities time schedule.

Assigned hours	**Activity**
8	Functional requirements analysis
8	Study of existing documentation
16	Training on the current image feature extraction system
16	Architecture design
16	UI design
160	Back end development
80	Front end development
16	Documentation

- MUST 01: development of a training function;
- MUST 02: integration with Thron APIs;

* Recommended:
  - SHOULD 01: development of a prototype user interface;
* optional:
  - MAY 01: development of unit tests;
  - MAY 02: extraction of metrics to evaluate the algorithm's performance;
  - MAY 03: training process automation;
  - MAY 04: integration of the prototype to product.

3.4 Schedule

Schedule is organized to follow the initial requirements; however, as already explained, these changed along the way influencing the schedule as well.

3.5 Future development

At the end of my internship, my co-workers will continue the project of developing a custom *object detection* application to integrate in Thron. Both my successes and failures will help in the following phase, as in such an innovative field a process experimentation through trial-and-error is at some point inevitable.

Chapter 4

Design and development

In this chapter I will explain my technology choices and how these influenced the development of the final application.

4.1 Technology and tools

In the following I will introduce the technologies I used and the reasons behind their choice.

Deep learning framework

There are many popular deep learning frameworks out there, but I was asked to choose between *MXNet* and *TensorFlow* after experimenting with both for bit. Keep in mind that I was required to use *YOLO v3* as my *convolutional neural network* and it influenced greatly my final choice. Furthermore, due to Python being predominant in deep learning, I had little freedom in choosing a programming language[1]

MXNet and Gluon CV

MXNet is an open source deep learning framework developed by Apache. It supports various languages, but the most extensive toolkit is written in Python; in particular MXNet Python provides *Gluon*, a high-level API that allows to easily build and train *deep neural networks*. *Gluon* is also used by *Gluon CV*, a *computer vision* toolkit that provides out-of-the-box implementation of several *neural networks*, pre-trained models and even complete *training scripts*. In particular *YOLO v3* is well supported and many useful tutorials are available.
Both *MXNet* and *Gluon CV* can be installed from *PyPi*$_{\mathrm{g}}$ with *Pip*$_{\mathrm{g}}$.
There are minor compatibility issues since *Gluon CV* 0.3.0 requires *MXNet* 1.3.0 and later.

[1]Actually I did try Scala APIs on MXNet, but I had little success with them due to the restricted set of functionality provided when compared to Python.

TensorFlow

TensorFlow is an open source machine learning framework developed by Google Brain. It supports various language, but as common in this field the most extensive one is written in Python. *TensorFlow* is compatible with *Keras*, a high-level API to build and train *deep neural networks*; sadly *Keras* does not provide any network model for *computer vision*.
When looking to *YOLO v3* support I explored *TensorFlow*'s *GitHub* repository, where they host some research projects developed and maintained by independent researchers and not officially supported or distributed in release branches. Among the *computer vision* models on there I did not find *YOLO v3*, but I took my time to experiment with a *SSD* model just to see if *TensorFlow* was worth anyway.
TensorFlow can be installed from *PyPI* with *Pip*, but the research toolkit cannot.

My choice

Since *MXNet*'s *Gluon CV* provides a comprehensive toolkit with models, scripts and utilities to train and infer *YOLO v3 convolutional neural networks*, it easily became my framework of choice.
As I said I tried *TensorFlow* with *SSD*, but it was a low-level implementation that relied a lot on a *Jupyter notebook*$_{\text{g}}$ and it was not suitable for the goals of my project.

IntelliJ IDEA

I used *IntelliJ IDEA*$_{\text{g}}$ IDE to write my code and I installed *PEP Python* linter plugin to check my code style.

GitLab

I was given access to a *GitLab*$_{\text{g}}$ repository to version control my work and I used *IntelliJ IDEA*'s built-in *Git* interface to access it.

Docker

Focker$_{\text{g}}$ is a software application to create containers, virtual environments with their own operative system and tools installed. Containers can communicate with each other and to the outer world through ports, and can mount hard drives when consistent memory is needed.
I used it in various steps of my project's development; at first I created a *Linux Docker* container to set up a *Python 2.7* and *MXNet* environment to run my code. I also created a *MongoDB* instance on another container.
At some point I even dockerized my application to run it on *AWS*.

OpenAPI + Swagger

Thron works with a service architecture, so I was asked to follow it when developing my inference web application. I wrote a *YAML*$_{\text{g}}$ *OpenAPIs*$_{\text{g}}$ using *Swagger*$_{\text{g}}$ to declare my application's interface, which also served as documentation.

Flask + Connexion

My server application for inference is written in *Python*, so I set up a *Flask*$_g$ server to host it. The *Flask* needed to adhere to the operations and responses I declared on the *OpenAPI*; matching everything manually would be a tedious job, but thankfully *Zalando*[2] developed a framework called *Connexion*[3] that, provided the *OpenAPI* automatically handles the HTTPs requests and maps them to your Python functions.

Postman

I used *Postman*$_g$ to send *HTTP* request to my web server and check its behavior.

MongoDB + Robo 3T

I set up a *MongoDB*$_g$ instance in a *Docker* container to model a queue I needed in my inference application, and I used the *Pymongo*[4] toolkit to work with it from my server application.
To easily access and manage my database, I used *Robo 3T*$_g$ client.

AWS SageMaker

I experimented with *AWS SageMaker*$_g$ to test whether a *YOLO v3* training script could easily take advantage of *AWS*'s computational power in a future production environment.

4.2 Design

In this section I will cover the design choices I made while developing my application.

4.2.1 Overview

As stated before, THRON'S *DAM* categorizes images inserted by the clients to make their access easier. Details about how these images are stored in a non-relational database are irrelevant for my project, and I actually do not know them. However, how a database export is structured and what it contains do is important and had major influence in some implementation choices.
A database export is a *JSON lines*$_g$ where each line represent a valid JSON file containing information about an image in the database; the covered fields are the following:

* ID: image unique identifier;
* Client ID: client identifier[5]; the goal is to construct a custom dataset for detection for each client's needs so this parameter is vital;
* Url: location where the image is stored, likely an *Amazon S3*[6] cloud storage location.

[2]E-commerce platform.
[3]Zalando Connexion repository: `https://github.com/zalando/connexion`
[4]Pymongo website: `https://api.mongodb.com/python/current/`
[5]Thron has several clients and often uses client ID based sharding techniques to run services in order to guarantee various levels of fairness.
[6]AWS cloud storage service; Amazon S3 website: `https://aws.amazon.com/s3`

∗ Data: image properties relevant in the dataset context such as *bounding boxes* coordinates and classes.

4.2.2 Dataset creation

Format choice

To train the network model for my prototype I was given a dataset created by one of my co-workers; his format of choice[7], however, it was not compatible with my framework so I had to convert it with a script.
Gluon CV provides modules to load a custom dataset from two different formats:

∗ Record file format; this format is composed by two different files, a **.rec* binary file, containing the encoded images and their labels, and a **.idx* file containing the indexes to provide random access to the binary;

∗ Pure text file format; this format is composed by a **.lst* file[8]. Each line of this file contains the path to an image on disk and the labels describing its bounding boxes.

The record file format allows faster training, but I chose to use the **.lst* file format instead because it allows you to work directly from raw images without prior computation. There are two main reason behind my choice:

1. The production training process should run on *AWS SageMaker*, which reads input data from an *Amazon S3*; Thron already uses *S3* so uploading the training on a dedicated bucket on there would be the easiest approach;

2. In production environment the datasets should be created from *JSON lines file* database exports, so simply converting the *JSON lines* content describing the bounding boxes for each image to an **.lst* would be the easiest and fastest approach.

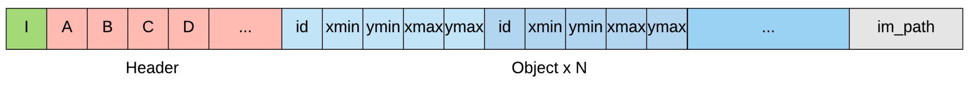
Dataset creation approach

The dataset-creation module I designed takes a *JSON lines* database export file as its input and creates a **.lst* for each *client ID* found in the *JSON lines.* The module also provides a function to split the dataset into *training dataset* and *validation dataset* after creation and saves an additional text file containing statistics about the dataset[9].
I chose to keep the creation of the full dataset and splitting it separated, as it is common practice to split the dataset right before training.
To track the class names with their numerical encoding used in the **.lst* dataset a **.txt* calles *synset.txt* is used. This file must be created manually.

4.2.3 Training

Model training for my *YOLO v3* network can be executed as a stand-alone *Python* script, which is heavily based on the suggested *training* script provided by the *Gluon CV* toolkit. This script receives as an input the *training* and *validation* datasets in

[7]Each raw image in the dataset was labeled by an XML file containing the image size, bounding boxed and classes.
[8]It is a text file readable as a tab-separated **.csv file.*
[9]These statistics include number of images, number of classes, cardinality and density.



I	Integer Index of this sample
A	Length of header, at least 2
B	Length of label for each object, usually 5
C D ...	Optional, extra information can be stored here
id	Integer class ID [0, C - 1], where C is number of total foreground classes
xmin	Left border of object, normalized by width to range [0, 1]
ymin	Top border of object, normalized by height to range [0, 1]
xmax	Right border of object, normalized by width to range [0, 1]
ymax	Bottom border of object, normalized by height to range [0, 1]
im_path	Relative path to the same root folder

Figure 4.1: Structure of a **.lst* file containing dataset information.

**.lst* format and the *synset.txt* file.
My script actually performs *transfer learning* instead of training from scratch due to the better performances achievable this way; the base network used is a YOLO v3 Darknet53[10] pre-trained on the *VOC* dataset; the *VOC* weights are provided by the *Gluon CV* toolkit and should cover classes general enough to prove useful in most cases.
The training process is composed by both *training* and *validation* loops, and I chose to perform *validation* after every epoch; this behavior can be easily changed by setting the provided input arguments. As for evaluation metric, I used the *mAP* metric in its version used for the *Pascal VOC challenge*[11].

Hyperparameters

My scripts accepts arguments to tune the training *hyperparameters*. Follows the hyperparameters list and their description:

- Data shape: input data shape; the shortest side of each image will be resized to match this value. Accepted values are 320, 416 and 608 (32 multiples) and it defaults to 416;
- Batch size: number of images in each computing batch. It must be tuned accordingly to your hardware as a batch size to big would trigger a bus error. It defaults to 8 (which works for CPU), but with GPU training you might be able to use a bigger size such as 32;
- Workers: number of workers used for training. You must tune it accordingly to your hardware as a number too high might result in a thread error;
- Gpus: numbers of the GPUs to use as a comma separated list of integers; to train on CPU leave this argument empty;
- Epochs: number of epochs to perform;
- Learning rate: speed at which your model learns the dataset content. A high learning rate makes you model learn quicker, but might result in overfitting;
- Learning rate decay rate: speed at which your learning rate decays;
- Learning rate decay epoch: epoch interval where your learning rate should decrease;
- Momentum: stochastic gradient descent momentum;
- Syncbn: argument to synchronize devices when training in parallel.

Model export

There are two ways to export a trained (or partially trained) network:

- Saving weights: it is possible to set the network weights as a **.PARAMS* binary file every set epoch; this type of file is mostly used as a training checkpoint to resume training from later.

[10]YOLO v3 convolutional neural network built on top of a Darket classifier composed by 53 layers
[11]Object detection challenge on the *VOC* dataset; it used a *IoU* threshold of 0.5 for evaluation.

* Network export: it is possible to set an export epoch where both the network architecture with adjusted top layers and its weights are exported. The network itself is saved as a **.JSON* file describing its layers while the weights are saved as a **.PARAMS* similar to the checkpoint file. These files are coupled and must be used together to load the trained network; this export approach is used when you want to save a model to use for inference, as it can be loaded from different languages other that *Python*.

Resume training

Since training demands a lot of computational power, it is useful to be able to stop a training process in order to resume it later. The following arguments allows you to load weights from a training checkpoint to resume training:

* Resume: path to the **.PARAMS* binary file containing the training checkpoint;
* Start epoch: epoch where training should be resumed.

It is important to load a checkpoint **.PARAMS* binary file to resume training as the model export **.PARAMS* binary file cannot be loaded without its paired **.JSON*.

AWS SageMaker

AWS provides a service called *SageMaker* for production deployment of *deep learning* applications. There are three ways to deploy on *AWS*:

1. You can use one the pre-made inference services, which is the easiest approach but has evident limitations;
2. If your algorithm runs on a supported framework, such as *MXNet*, *TensorFlow*, *Pytorch* etc. you can submit it with minor changes and *AWS* will run it inside a *Docker* container; this approach proved to be troublesome when a specific version of the frameworks and additional dependencies are required;
3. You can submit your own *Docker* container with your algorithm for *AWS* to run; this approach proved to be troublesome as the *Docker* container requirements to be run on *AWS* aren't clearly explained.

My initial goal was to run my *training* script with the second approach, but I couldn't do I because of the reasons I will explain in chapter seven. Thus, I tried the third approach, writing my own *Docker image* containing my code and loading it to *SageMaker*, but all of my attempts to run training this way failed; I will explain the reasons for this in chapter seven as well.

4.2.4 Inference service

While training usually requires GPU power to speed up the process, inference can be reasonably done on CPU, so I decided to keep my training script and my inference script separated.

RESTful Architecture

My inference request server uses a*RESTful API*$_g$ architecture that I designed using the API first approach; specifically, I wrote my *OpenAPIs* using the *Swagger*. To

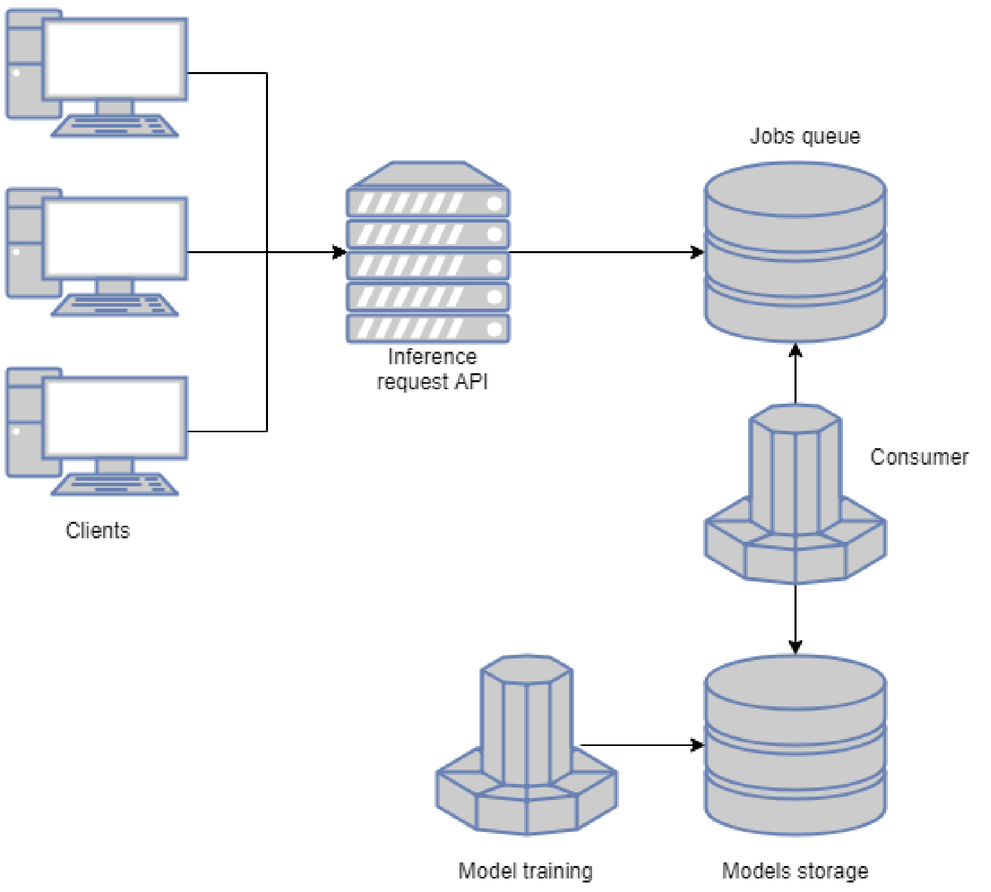


Figure 4.2: Schematic representation of my application architecture. *Clients* send HTTP requests to the *inference request API*, which inserts them in a *jobs queue* to be processed asynchronously. The *consumer* picks up batches of jobs sharded by client ID and loads the correspondent model from the model storage. The *training*, who potentially resides on AWS, periodically updates the models.

implement my APIs I took advantage of *Zalando*'s *Connexion*, a framework based on *Flask* that tracks the operations declared in the OpenAPI to the application, validates the HTTP request exchange by comparing the actual JSON content to what is declared in the API and manages error messages[12].
To increase decoupling I kept my server interface separated from its implementation; in fact operations are declared in a *JobOpsMongo* module called by the server interface via a *JobService* module. I followed this decoupling approach for both *MongoDB* related operations and *MXNet* inference related operations, but the latter aren't directly called by the inference request server but are used by a queue consumer I did not implement and that potentially resides on AWS.

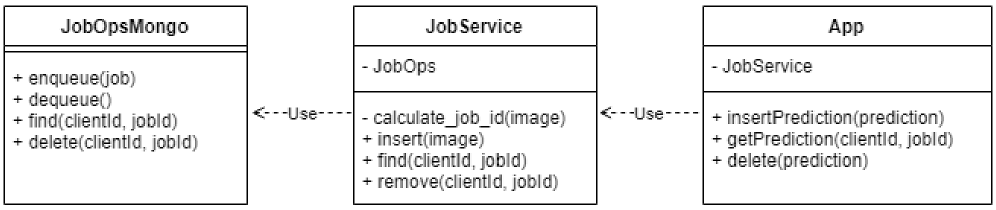


Figure 4.3: UML describing the relationship between the server application (App class) and the implementation of its methods.

Basically the server's job is to accept and manage inference requests by inserting them into a *MongoDB* queue; from there a consumer will dequeue them and run the actutal inference.
HTTP requests work with *JSON* objects which, despite *Connexion*'s automatic conversion into *Python* dictionaries, still require tedious syntax to access their fields. On the other hand *MongoDM* indexes its content in *JSON* format. Since working with different formats of data would unnecessarily increase the application's complexity, I found it useful to model *Python* class objects to reflect what was declared in the *OpenAPI* and use them to work within the logic modules. My approach while designing these classes was to roughly follow the structure I declared in the OpenAPI, adding further information that could be useful inside the application, and to provide object creation, serialization[13] and deserialization [14] methods to quickly convert between *Python* objects, *Python* dictionaries and *JSON*s as needed.

Asynchronous computation

While inference isn't to heavy on the computational side of the task, performing a prediction requires the trained network model to be loaded into memory and, since each model occupies over 250MBs of space, it is obvious that constantly switching from model to model would make the performance fall. The easiest solution to this problem is to perform inference by sharding the requests by client ID, so each batch of images runs on the same network model. A further advantage of client ID sharding is that it allows guarantee different levels of fairness accordingly to each client's service contract. Since computation is obviously performed asynchronously, a client needs a way to

[12]Since I used *Connexion* instead of a pure *Flask* server application I couldn't override the standard HTTP error messages
[13]From *Python* object to *JSON*
[14]From *JSON* to *Python* objects.

retrieve its inference once it's completed. There are two main options to manage this: *callbacks*$_g$ and *polling*$_g$. I chose the latter because it comes more natural when working in a *RESTful* environment, as after having inserted the new job in the queue the server can simply send a HTTP 202 Accepted response containing the address to check for the requested resource.

Queue

I modeled the queue on *MongoDB*, as it provides an easy way to sort the elements in its collections by insertion time. Furthermore, contents in *MongoDB* are represented as *JSON* objects, so it proved to be convenient to work with the objects I declared in the *OpenAPI*.
The jobs queue is accessed by both my inference request server and a consumer, but I did not implement the latter.

Inference work flow

When a client needs an image to be inferred, it sends the server a HTTP Post request containing a *JSON* stating the URL to the image and a client ID; upon receiving the request the server creates a new inference request job and inserts it in a *MongoDB* queue. If the insertion was successful, the server sends the client a HTTP 202 Accepted response, containing the URL where to check for the requested prediction; since computation runs asynchronously, the client might have to check several times for its requested inference result before it's actually available.
When the prediction is ready, the client can retrieve it from the given URL; inference contains the detected object classes, their confidence score[15] and their *bounding boxes*, expressed in percentage on the image's size.
Predictions are executed by I consumer I did not implement.

4.2.5 Standalone inference script

Before designing and developing the inference request server application, I wrote a *Python* standalone script to perform inference on an image. This script takes the image path and the network model to use as arguments, and then run inference. The result is composed by the classes of the detected objects, their *bounding boxes* and their confidence score; furthermore the inference script saves a copy of the input image where it draws[16] the *bounding boxes* with their class and confidence score.
I used this inference script mainly when testing whether the network training was successful; since I used unlabeled data[17] for testing, visually checking whether the detection was accurate proved to be faster that automatizing the process, as classes and their *bounding boxes* must be manually determined by a human anyway.

[15]How much the network is confident the detected object belongs to the stated class; a higher score means higher confidence.
[16]To edit images I used Python PILLOW library. PILLOW website: `https://pillow.readthedocs.io/en/5.3.x/`.
[17]Using dataset images for testing could give false performance, especially if overfitting has occurred.

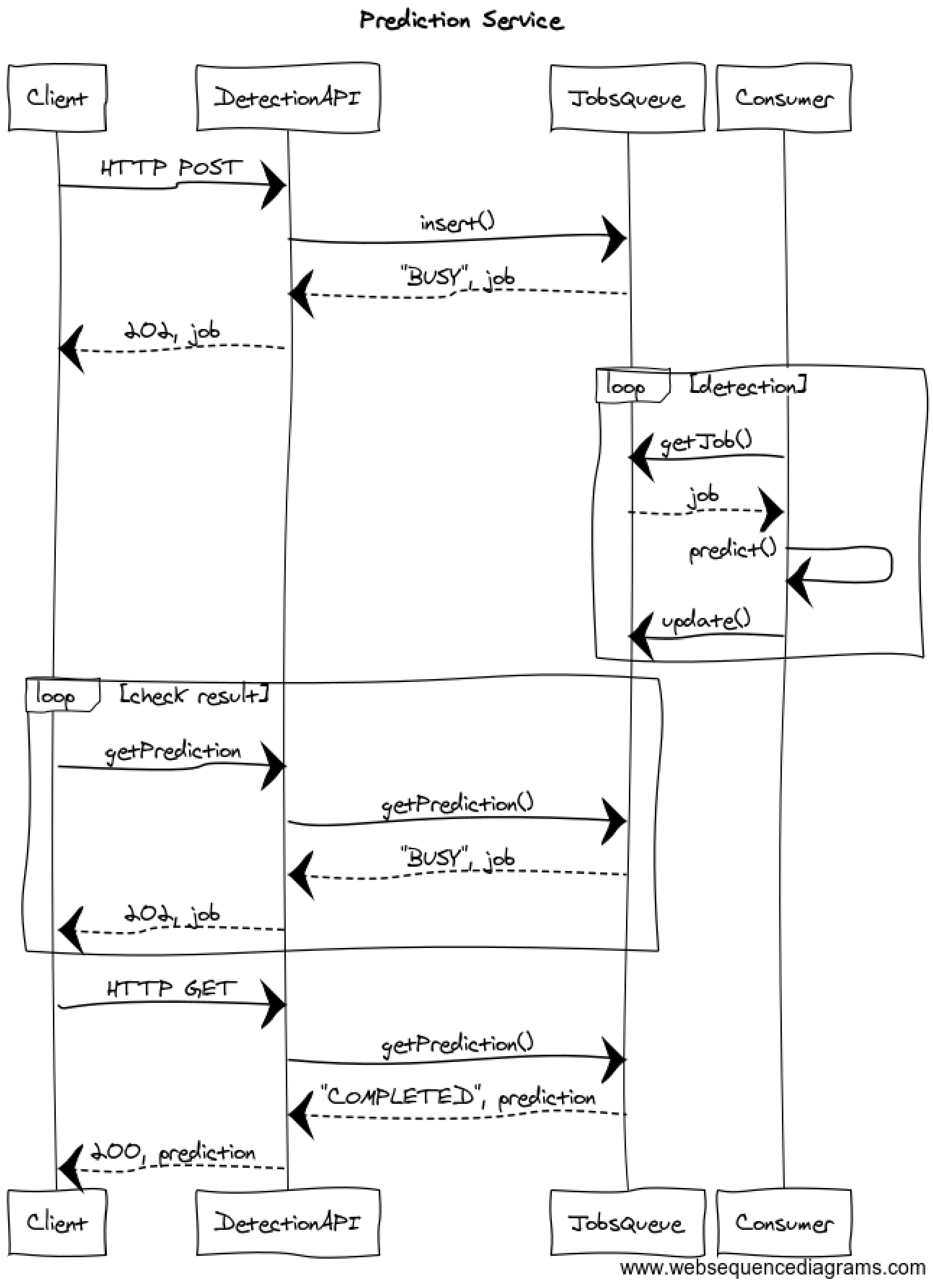


Figure 4.4: UML flow chart displaying the HTTP message exchange in the application.

Chapter 5

Tests and validation

5.1 Unit tests

Due to the nature of the project it wasn't always possible to write useful *unit tests* to check my code; however I did write tests whenever it was possible.

5.1.1 Training script

Due to its nature, it wasn't possible to write useful units tests to check my *training* script worked properly. Apart from the fact that while running on CPU the script required a couple of days to terminate, its output was a trained network model and its weight in binary format, whose correctness can't be tested aside from performing inference from them and confront the results to the ground truth. As I previously explained, trained models must be tested with new data, which isn't contained in the *training/validation* dataset; this means that the test images are unlabeled data, thus the problem to manually label them. I found printing the detected classes and their *bounding boxes* on a copy of the input image to be the easiest and fastest way to check whether the predictions where correct, as ground truth for *computer vision* tasks must still be determined by a human.
To manually test whether my trained model produced correct detection, I wrote a standalone *Python* script, heavily based on the script provided by *Gluon CV*.
Furthermore, since both my *training* and *inference* scripts heavily relied on *Gluon CV* code with minor changes, I could assume that to a certain degree they were correct.

5.1.2 Inference service

I wrote unit tests that cover both the methods provided by the *JobOpsMongo* module and the *PredictionOpsMxnet* modules.
While the *JobOpsMongo* module was actually used by the application and I tried its functionality by sending HTTP requests via *Postman* as well, the *PredictionOpsMxnet* module methods have only been called by its unit tests. In fact, the *PredictionOpsMxnet* module provides the inference methods that the consumer should use, but I did not implement it because it should potentially run on *AWS* and therefore requires specific coding.
When writing unit tests my approach was to define a *spec* module for the code I wanted to test; then, in the *spec* module I wrote functions that called the method they were

supposed to test and checked whether the expected outcome would occur.

5.2 Keynote

On my last day of internship I was asked to perform a short lecture to explain what my project was about. With the aid of slides, after a brief introduction on *object detection* and its applications, I showcased the technologies I used, pointing out their pros and cons.

Chapter 6

Conclusion

6.1 Final balance

As initially decided, my internship lasted for 320 in total. The work schedule, however, did suffer from major changes as the implementation of some requirements needed more time that estimated.

6.2 Achieved goals

In this section I will track the project requirements, stating whether they were implemented and the occurred issues.

6.2.1 MUST 01: development of a training function

Implemented. I coded a standalone *Python* training script on *MXNet* framework with *Gluon CV*. The network architecture I used is *YOLO v3* as required at the beginning of my internship.
During development I incurred in the following issues:

* **Hardware requirements (1):** *MXNet* relies on newer CPU instructions, available only on fairly recent hardware. Despite it being a quite average machine,

Table 6.1: Final balance.

Required hours	Activity
10	Functional requirements analysis
34	Study of existing documentation
26	Training on the current image feature extraction system
16	Architecture design
0	UI design
210	Back end development
0	Front end development
16	Documentation
8	Keynote

I could not execute *MXNet* on the PC I was assigned at the beginning of my internship. Before I was given a newer one I worked on a remote machine on *AWS* via SSH.

∗ **Hardware requirements (2):** training a network requires a lot of computational power, as it consists of thousands of matrix operations. Since these operations can be executed in parallel, GPUs, which are indeed optimized to execute matrix operations on their thousands of cores, are the best way to speed up training. Unfortunately my machine didn't have a GPU, so I had to work with the long times and downsides of CPU training;

∗ **Framework compatibility:** *YOLO v3* is implemented in *Gluon CV* version 3.x.x and later, which requires *MXNet* version 1.3.x and above. While I started my internship on 3rd September, the stable release of *MXNet* version 1.3.0 came out on 14th September, while *GluonCV* version 0.3.0 came out on 16th October. Thus I could not deploy my training script on *AWS SageMaker* as the framework version I needed wasn't supported yet;

∗ **Nightly builds and bugs:** both the framework and the toolkit are extremely new, with *Gluon CV* still being on 0-version, and I often had to work with nightly builds. The major downside of this is that the online documentation and provided source code didn't always match with the content of the build, so sometimes the functionality I needed didn't actually exist yet; when working with *Gluon CV* I was actually forced to implement some functionality myself[1]. Furthermore, the toolkit code contained a few bugs, one of which was contained on release version *Gluon CV* 3.0.0; I sent an issue to fix it, but after it was fixed I was limited on using nightly builds as release 0.4.0 wasn't out yet;

∗ **Documentation:** documentation provided by both the framework and the toolkit I used wasn't always clear in regards of the input and output of functions (e.g. image size, matrix dimension); hence, I had to find for myself the right formats through a trial and error process while developing.

6.2.2 Integration with Thron APIs

Implemented. I coded a Python server to manage the inference requests. I did not implement the consumer, which would basically be an infinite loop that executes prediction of batches of images loaded from a queue, as it would finally reside on some *AWS* service and therefore would not be interesting to have locally; I did however provide the method it should use to perform inference.
I wrote my inference application with the API-first approach and following the *REST* standard. As of currently Thron isn't fully RESTful, but I was asked to strictly adhere to the standard. Furthermore, my project is structured following the standard used in Thron APIs. During development I incurred in the following issues:

∗ **Framework compatibility:** see above development of a training function issues;

∗ **Nightly builds and bugs:** see above development of a training function issues;

∗ **Documentation:** see above development of a training function issues.

[1]What I actually did was to copy-paste the provided source code into a module I declared.

6.2.3 SHOULD 01: development of a prototype user interface

Not implemented. I did not develop a prototype user interface; we discussed about it during the development and we decided that focusing on the back end application would be more meaningful given the project nature.

6.2.4 MAY 01: development of unit tests

Implemented. I wrote unit tests for both the jobs queue management operations and the inference methods in my server application.
No particular issue occurred.

6.2.5 MAY 02: extraction of metrics to evaluate the algorithm's performance

Not implemented.

6.2.6 MAY 03: training process automation

Not implemented.[2] Thron often relies on *AWS* services to perform computation, so I was asked to try *SageMaker*. As I introduced in chapter five, there are two ways to run custom code on *SageMaker*. The easiest and preferred approach was to run my *training* script on one of *Amazon SageMaker*'s pre-made *Docker* container, but I needed one providing *MXNet* framework version 1.3.0, which wasn't available yet.
I built a custom *Docker* container containing my script, but I couldn't make it run on *SageMaker*. I incurred in the following issues that prevented me from implementing this requirement:

* **MXNet supported version:** At the time of my internship *SageMaker* provided *MXNet* up to version 1.2.1, but I needed version 1.3.0;

* **Dependencies**: *SageMaker* doesn't provide a way to install dependencies on their *MXNet* containers[3], so I couldn't even install *Gluon CV*;

* **Documentation:** Documentation about both pre-made containers and custom containers isn't always clear; in fact, the required structure for custom containers isn't stated at all, so the service would refuse to run *Docker* containers that work fine locally, making debugging a frustrating trial-and-error experience.

Because of these issues and my Tutor's busy schedule, I didn't manage to fulfill training automation.

6.2.7 MAY 04: integration of the prototype to product

Not implemented.

[2] I actually did work on this requirement, but with scarce success.
[3] On *TensorFlow* containers you can install dependencies, so hopefully the feature will be implemented for *MXNet* as well.

6.3 Acquired skills and knowledge

My internship at THRON was both my first working experience and my first approach to *deep learning*, so I learned a lot of things. First of all I finally know what *deep learning* is about, and I'm actually impressed that thanks to frameworks like *MXNet* I could train a *deep neural network* without prior knowledge on the topic.
Most of the technologies I used in my project were completely new to me, so I also had the opportunity to acquire new practical skills; I learned *Python*, which is the predominant language in the *machine learning* field, I wrote my first *OpenAPIs* and I created my first *MongoDB* database. I used *Docker* for the first time and now I can't even imagine working without it, and finally I wrote my first *Markdown* documentation to help my co-workers who will continue the development of the *object detection* application.

6.4 Comments

I am satisfied with what I learned during my internship. I had the opportunity to experiment in a fast-growing field, and accordingly to the available tools I was able to create the core of a working product.
People at THRON are really nice and I made some new friends.

Appendix A

Training a YOLO v3 network

A.1 Overview

In the following I will explain step by step how a training script works.
The training script can be run as a standalone that takes input arguments; with the aid of some auxiliary functions it takes care of initialization, loads the data and calls the training function.

```
if __name__ == '__main__':
    args = parse_args()
    # fix seed for mxnet, numpy and python builtin random generator.
    gutils.random.seed(args.seed)
      # training execution context
    if args.gpus == '':
        ctx = mx.cpu()
    else:
        ctx = [mx.gpu(int(i)) for i in args.gpus.split(',') if i.strip()]
    # loading the network
    net_name = '_'.join(('yolo3', args.network, args.dataset))
    args.save_prefix += net_name
    # use sync bn if specified
    num_sync_bn_devices = len(ctx) if args.syncbn else -1
    if num_sync_bn_devices > 1:
        classes = get_dataset_classes(args.synset)
        net = get_model(net_name, classes=classes, transfer='voc',
            pretrained_base=False, num_sync_bn_devices=num_sync_bn_devices)
        async_net = get_model(net_name, classes=classes,
            pretrained_base=False, transfer='voc') # used by cpu worker
    else:
        classes = get_dataset_classes(args.synset)
        net = get_model(net_name, classes=classes, transfer='voc',
            pretrained_base=False) #, num_sync_bn_devices=num_sync_bn_devices)
        async_net = net
    if args.resume.strip():
        net.load_parameters(args.resume.strip())
        async_net.load_parameters(args.resume.strip())
    else:
        with warnings.catch_warnings(record=True) as w:
            warnings.simplefilter("always")
            net.initialize()
            async_net.initialize()
    # loading the training and validation data and evaluation metrics
    train_dataset, val_dataset, eval_metric = get_dataset(args.train_dataset,
        args.valid_dataset, classes, args)
    train_data, val_data = get_dataloader(
        async_net, train_dataset, val_dataset, args.data_shape,
            args.batch_size, args.num_workers)

    # training
    train(net, train_data, val_data, eval_metric, ctx, args)
```

A.2 Training

The training function itself is unsurprisingly the most complex part of the training script, so I will only show the actual training loop omitting initialization and parameters save.
For each epoch the training loop goes through all of the training dataset. It computes a batch at a time, calculating the loss and updating the weights. After every epoch it executes validation, updates a log fine and saves the training checkpoint.
When all the epochs are done the training function exports the trained model.

```
def train(net, train_data, val_data, eval_metric, ctx, args):
    """Training pipeline"""
      #
      # Some initialization
      #
    for epoch in range(args.start_epoch, args.epochs):
        tic = time.time()
        btic = time.time()
        mx.nd.waitall()
        net.hybridize()
        for i, batch in enumerate(train_data):
            batch_size = batch[0].shape[0]
            data = gluon.utils.split_and_load(batch[0], ctx_list=[ctx],
                batch_axis=0)
            fixed_targets = [gluon.utils.split_and_load(batch[it],
                ctx_list=[ctx], batch_axis=0) for it in range(1, 6)]
            gt_boxes = gluon.utils.split_and_load(batch[6], ctx_list=[ctx],
                batch_axis=0)
            sum_losses = []
            obj_losses = []
            center_losses = []
            scale_losses = []
            cls_losses = []
            with autograd.record():
                for ix, x in enumerate(data):
                    obj_loss, center_loss, scale_loss, cls_loss = net(x,
                        gt_boxes[ix], *[ft[ix] for ft in fixed_targets])
                    sum_losses.append(obj_loss + center_loss + scale_loss +
                        cls_loss)
                    obj_losses.append(obj_loss)
                    center_losses.append(center_loss)
                    scale_losses.append(scale_loss)
                    cls_losses.append(cls_loss)
                autograd.backward(sum_losses)
            lr_scheduler.update(i, epoch)
            trainer.step(1) #trainer.step(batch_size)
            obj_metrics.update(0, obj_losses)
            center_metrics.update(0, center_losses)
            scale_metrics.update(0, scale_losses)
            cls_metrics.update(0, cls_losses)
        #
        # validation and parameter save
        #
```

Glossary

Artificial Intelligence Umbrella term for theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages. Machine learning and deep learning are sub-fields of artificial intelligence. . 2, 39

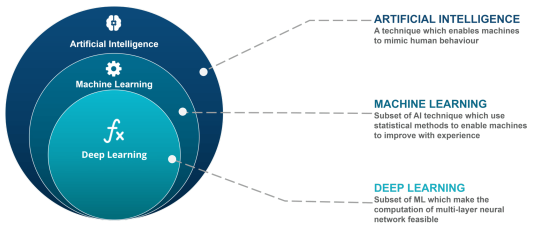

Artificial Intelligence
Machine Learning
fx
Deep Learning
ARTIFICIAL INTELLIGENCE
A technique which enables machines to mimic human behaviour
MACHINE LEARNING
Subset of AI technique which use statistical methods to enable machines to improve with experience
DEEP LEARNING
Subset of ML which make the computation of multi-layer neural network feasible

Figure A.1: Relationship between artificial intelligence, machine learning and deep learning.

Amazon Web Services Acronym of Amazon Web Services. A set of various services including, among others, cloud storage, computational power and artificial intelligence specific services.. 13, 39

Back-propagation Propagation of the calculated values to improve the network performance; these values are used to update the weights used by the neurons functions.. 3, 39

Callback In an asynchronous context, process where the server notifies the client when the requested resource is ready.. 26, 39

Computer Vision Interdisciplinary field that deals with how computers can be made to gain high-level understanding from digital images or videos. Computer vision technology is used to automate tasks that the human vision can do, such as instance counting, object and face recognition and video surveillance.. 5, 39

Convolutional Neural Network In deep learning, class of deep, feed-forward artificial neural networks, most commonly applied to analyzing visual imagery. Deep neural networks for object detection are SSD, YOLO and R-CNN.. v, 39

CUDA Compute Unified Device Architecture. Parallel computing platform and programming model invented by NVIDIA; it is vastly used in deep learning frameworks and applications.. 14, 39

Dataset Collection of data, commonly with statistic properties. Popular open source datasets used in computer vision are MNIST, COCO, ImageNet and VOC.. 5, 40

Deep Learning Part of a broader family of machine learning methods based on learning data representation, as opposed to task-specific algorithms. Deep learning models are inspired by the way biological brains compute data, and as such learn features by repeatedly experiencing them in a training process.. v, 40

Docker Computer program that performs operating-system-level virtualization, also known as "containerization". A Docker container can be compared to a virtual machine, but much lighter. Docker website: `https://www.docker.com/`. 18, 40

Epoch Epoch is when the entire dataset goes through the network forth (and back) once. When training you a network you commonly perform hundreds of epochs.. 9, 40

Flask Python server framework. Flask website: `http://flask.pocoo.org/`. 19, 40

GitLab Web based Git repository. GitLab website: `https://about.gitlab.com/`. 18, 40

Gluon CV High level toolkit for computer vision application on MXNet Python framework. Gluon CV website: `https://gluon-cv.mxnet.io/`. 14, 40

Gradient descent function Function based on differentiation that calculates how the weights should be changed to reduce the loss function.. 3, 40

IntelliJ IDEA Java Integrated development Environment developed by JetBrains. It provides plugins for several languages such as Python and Scala. IntelliJ IDEA website: `https://www.jetbrains.com/idea/`. 18, 40

Intersection Over Union Also known as IoU, measure to determine how well a detected bounding box overlaps with the ground truth.. 9, 40

JSON lines file A Json lines file is a *.json file that contains a valid Json on each line.. 19, 40

Jupyter notebook Web-based interactive computational environment for creating Jupyter notebooks documents.. 18, 40

Loss function Function to calculate the distance of the predicted value from the ground truth label. Commonly used loss functions are *Quadratic Loss Function* and *Cross Entropy Loss Function*.. 3, 40

Machine Learning Field of artificial intelligence that uses statistical techniques to give computer systems the ability to "learn" (e.g., progressively improve performance on a specific task) from data, without being explicitly programmed. Differently from deep learning methods, machine learning applications still need the problem's boundaries to be defined in the code by a human.. 3, 40

Marketing DAM Acronym of Digital Asset Management; a DAM is a software application to store, categorize and access multimedia content.. 1, 40

Mean Average Precision Accuracy evaluation metric for object detection models, also known with the acronym mAP. To calculate how well your network performs, mAP considers both the number of positives and actually how well these overlap with the ground truth bounding box. This second task is calculated using an IoU threshold.. 9, 41

MongoDB Open source, non-relational, document-oriented database program. MongoDB website: `https://www.mongodb.com/`. 19, 41

MXNET Open source framework for deep learning applications developed by Apache. MXNet website: `https://mxnet.apache.org/`. . v, 41

Figure A.2: MXNet logo.

OpenAPI Open Application Programming Interface. Publicly available interface to interact with web services; it often serves as documentation and guideline for service implementation (API first).. 18, 41

Overfitting Overfitting is a problem occurring in training a network when, due too long training or poor data samples, the models learns patterns that aren't actually relevant and therefore fails during inference. An overfitted model would predict exactly the content of sample data and fail on everything else because it adjusted too tightly to the examples.. 9, 41

Pip Pip Installs Packages. Package management system used to install and manage packages written in Python. It is mostly used for packages on PypI.. 17, 41

Polling In an asynchronous context, process where a client actively checks whether the requested resource is available by sending requests (e.g. client sending a HTTP Get requests to the server).. 26, 41

Postman API development tool; it features functionality to send HTTP request and edit their content. Postman website: `https://www.getpostman.com/`. 19, 41

Pypi Python Package Index. Official third party repository for Python packages. PyPI website: `https://pypi.org/`. 17, 41

RESTful API Representational State Transfer Application Program Interface. Web services that uses HTTP requests to perform transactions.. 23, 41

Robo 3T MongoDB client. It allows to do queries and manage the database content. Robo 3T website: `https://robomongo.org/`. 19, 41

SageMaker Web platform provided by Amazon to build, train and deploy machine learning models. It both supports popular frameworks and custom docker containers. Amazon SageMaker website: `https://aws.amazon.com/sagemaker/`. 19, 42

Swagger Online tool to declare OpenAPIs. Swagger website: `https://swagger.io/`. 18, 42

Tensorflow Open source framework mainly used for machine learning applications developed by Google. TensorFlow website: `https://www.tensorflow.org/`. .

Figure A.3: TensorFlow logo.

v, 42

YAML YAML Ain't Markup Language. Human-readable serialization language, commonly used in configuration files.. 18, 42

YOLO v3 Acronym of You Only Look Once; YOLO v3 is a convolutional neural network developed by Joseph Redmon. YOLO website: `https://pjreddie.com/darknet/yolo/`. v, 42

Bibliografia