NOVA IMS Information Management School

MDSAA

Mestrado em Métodos Analíticos Avançados

Master Program in Data Science and Advanced Analytics

Threat Detection with Computer Vision

Internship at Everis UK

Gabriel Azenha Cardoso

Internship report presented as a requirement to be awarded the master's degree in Data Science and Advanced Analytics

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

Threat Detection with Computer Vision

by

Gabriel Azenha Cardoso

Internship report presented as a requirement to be awarded the master's degree in Data Science and Advanced Analytics

Advisor: *Professor Doutor* Mauro Castelli

February 2023

ACKNOWLEDGEMENTS

I must express my gratitude to my family, especially my mother who encouraged me to take this master course and gave the support and conditions I needed to fulfill this goal.

I am also thankful to my friends from back home and all the new friendships I made along the way by helping me going through some hard moments that only a few could understand.

I also must thank Everis UK for the opportunity, and in particular to the Innovation team for receiving me with arms wide open and being extremely patient to answer all my questions and appeals for assistance.

A special thanks to Mauricio for guiding me through my first professional experience in information technology and being always kind and supportive and Wen for teaching me everything she knew on computer vision and deep learning.

My final acknowledgement goes to Professor Mauro Castelli, for all the advice and suggestions to this paper.

ABSTRACT

This document describes the work conducted during an internship experience at the AI Innovation Department of Everis UK (now NTT Data). It reports what was done, learned, and developed with the sole objective of having a commercial product solution for the company's clients.

The primary goal was to implement a solution in retail stores, to help assist the security team with threat detection. To do so, the solution consists in deploying trained deep learning models into hardware connected to the CCTV security cameras and detecting in that live feed any potential threats.

By the time I started working on this project, was at an advanced stage so I had to study all the work previously done to understand what was needed and properly integrate the team fully. My contribution was focused on the model training process, where I had to create and structure a dataset and train a model capable of detecting the targeted classes quickly and accurately.

KEYWORDS

computer vision; deep learning; inference; security

INDEX

LIST OF FIGURES

LIST OF TABLES

LIST OF ABREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
API	Application Programming Interface
CCTV	Closed-Circuit Television
CLI	Command Line Interface
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
CSI	Camera Serial Interface
CUDA	Compute Unified Device Architecture
CVAT	Computer Vision Annotation Tool
DLA	Deep Learning Accelerator
GPU	Graphics Processing Unit
HPC	High Performance Computing
HR	Human Resources
IT	Information Technology
IVA	Intelligent Virtual Agent
ML	Machine Learning
MVP	Minimum Viable Product
NGC	NVIDIA GPU Cloud
NN	Neural Networks
NVDEC	NVIDIA Video Decoding
NVENC	NVIDIA Video Encoding
OEM	Original Equipment Manufacturer
RTSP	Real Time Stream Protocol
SASL	Simple Authentication and Security Layer
SDK	Software Development Kit
SVM	Support-vector machine
TLS	Transport Layer Security
UI	User Interface
UK	United Kingdom
USB	Universal Serial Bus
VGG	Visual Geometry Group

VIC	Video Image Compositor
VM	Virtual Machine

1. INTRODUCTION

"*Everis (an NTT Data Company) is a multinational consulting firm that offers business and strategic solutions, development, maintenance of technological applications, and outsourcing services. The company operates in the telecommunications, financial, industrial, utilities, energy, public administration, and health sectors. It currently employs professionals at its offices and high-performance centers in 17 countries*" [1]. At Everis I integrated the Innovation department. This department is a key leverage for dealing with volatility, uncertainty, complexity and for creating adaptative strategies that help break down resistance and seize all possible opportunities. Here the process is to transform ideas into something tangible and beneficial for customers and society by creating a vision for a sustainable future through technology. The company works hard to foster a culture of innovation, both within its organization and its customers. Innovation takes many forms, and it can help to improve more traditional sectors of activity and business models while accompanying companies in more ambitious and disruptive initiatives to boost innovation and reach new markets and customers.

I was assigned to a project that the team had already started to develop, and my inclusion had the purpose of supporting the current team and speeding up the final delivery for testing and implementation at the client's premises.

The client is a retail multinational, and this project aims to develop a solution for the store's security teams. The solution consists in connecting and running an AI model with the live camera feed to identify potentially threatening objects and wearables in real-time. This happens by raising an alert that will be followed up by security and lead them to act swiftly on the potential issue. As previously stated, the customer for this solution is from the retail sector and internally they have a Computer Vision Platform initiative where they listed an Automated Weapon Detection System as one of the top 4 priorities to tackle and solve with this technology. In the UK alone there have been 815 assaults and robbery incidents reported where a weapon or a face-covering wearable was used.

Incident obs.	**Occurrences**
Weapon detected	558
Face covering worn	257

Table 1 – UK Stores Incident Registration

1.1. IDEA DESCRIPTION

This computer vision business use was already requested by our client's client, so at this point, NTT Data was a 3rd party in their CV Platform initiative. Another positive outcome from this project is that the technologies, tools, and methodologies for the solution can and will be re-used to create future computer vision use cases, where the changing requirement will be to build a new machine learning model, taking obvious advantage of the same architecture. The solution's architecture will be detailed in this report since the main challenge at first was its design and implementation, but mainly, the initiative aims to have an end-to-end solution able to use a live camera feed, process the videos using a deep learning model, get detections and send it through an API to a dashboard that will register those occurrences with details such as the hour, the object detected and the detection camera's id.

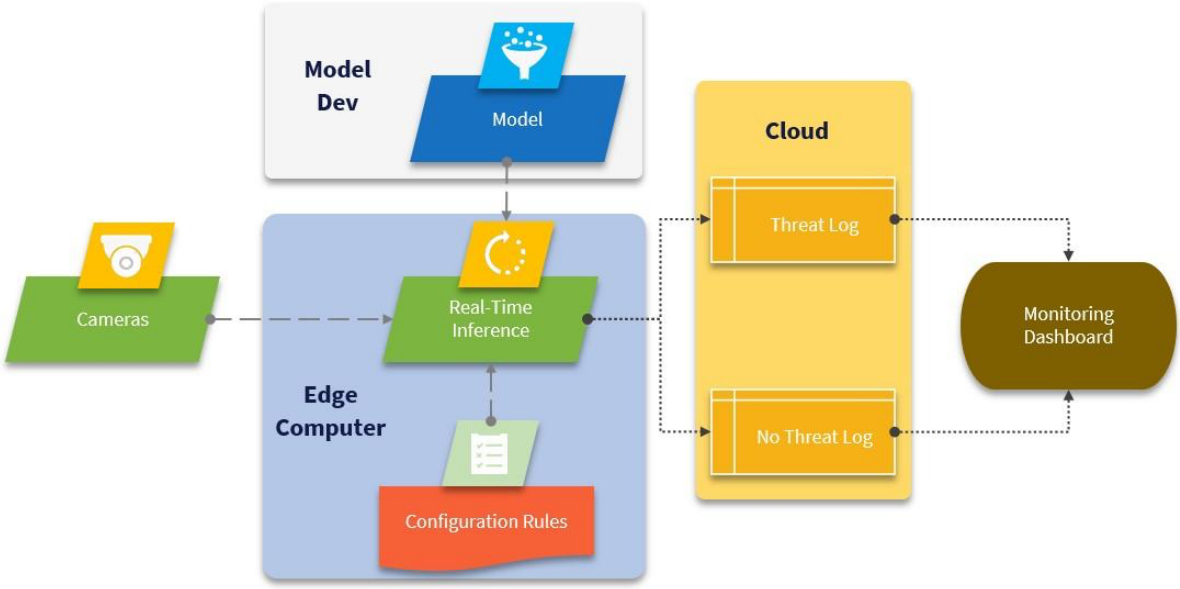

Figure 1 – Main Architecture Components

The *Model Dev* component is where we have the model training workflow represented below.

Figure 2 – Model Development Components

The *Edge Computer* is where the camera feed will be connected, and the trained model deployed to run a real-time inference process that will detect the intended objects. Those detections will be sent

to the *Cloud* component, where an API will filter the no threats and register the threats log in monitoring dashboards where real-time alerts will be raised per new entry.

1.2. BUSINESS OBJECTIVES

Create an end-to-end MVP ready for implementation and testing. The initial strategy was to implement a trial in 3 large stores by Q4 and if the results prove to be positive it will be followed by a trial in 10 other stores (8 large and 2 small).

The success criteria of it will be set by having the end-to-end MVP successfully detect the intended objects according to the configuration rules imputed.

1.2. SITUATION ASSESSMENT

1.2.1. Resources

There are 3 different resource types necessary to complete this project, hardware, software, and HR. For hardware necessities, we have the Nvidia Jetson Nano, Azure VM (which later was replaced by a HP local server). For software we need Nvidia SDK, TensorFlow (though at first, we used Pytorch), Python, Docker, git and CVAT. Regarding HR, the team should be composed by one Tech Lead, one Data Scientist and one ML Engineer.

1.2.2. Challenges and Benefits

Some of the main challenges identified were, how to get a proper dataset to train the model, the need to create realistic scenarios to train this model with and to use CCTV cameras as a source, and the cloud integration for real-time purposes. Overcoming these key challenges, the team will be able to conclude the project and have an innovative system to support security operations for a big variety of industries with the ability to process multiple standard CCTV cameras (without requiring the installation of new cameras) and have real-time reports and alerts.

1.2.3. Value Proposition

For the value proposition, we filled the NTT Data *Value Proposition Canvas* where we start by identifying the client's profile to select the *Customer Job* (this specific case relates to the store's security), identify the *Gains* that come with completing the tasks, and the *Pains* that come from failing to do so. Then we relate those *Gains* and *Pains* with the *Products & Services* the company offers and with that we can identify the *Pain Relievers* and the *Gain Creators*.

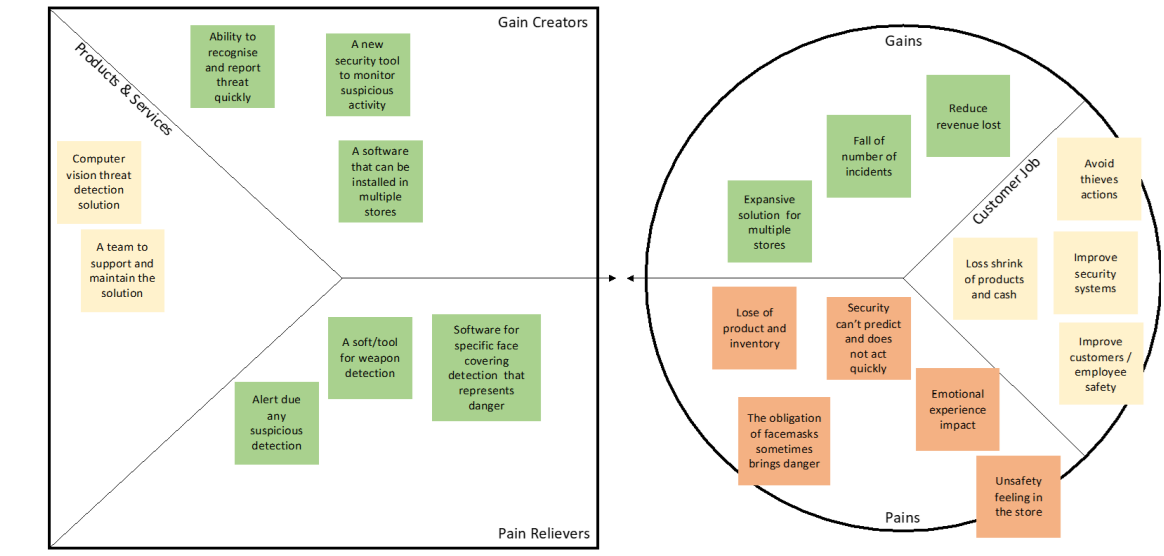

Figure 3 - NTT Data Value Proposition Canvas

1.2.4. Business Model

Same as in the Value Proposition case, the company already has a *Business Model Canvas* which is filled with all the information necessary to successfully develop the solution. The key information was already addressed above but with the canvas we can better illustrate it and present it to all stakeholders.

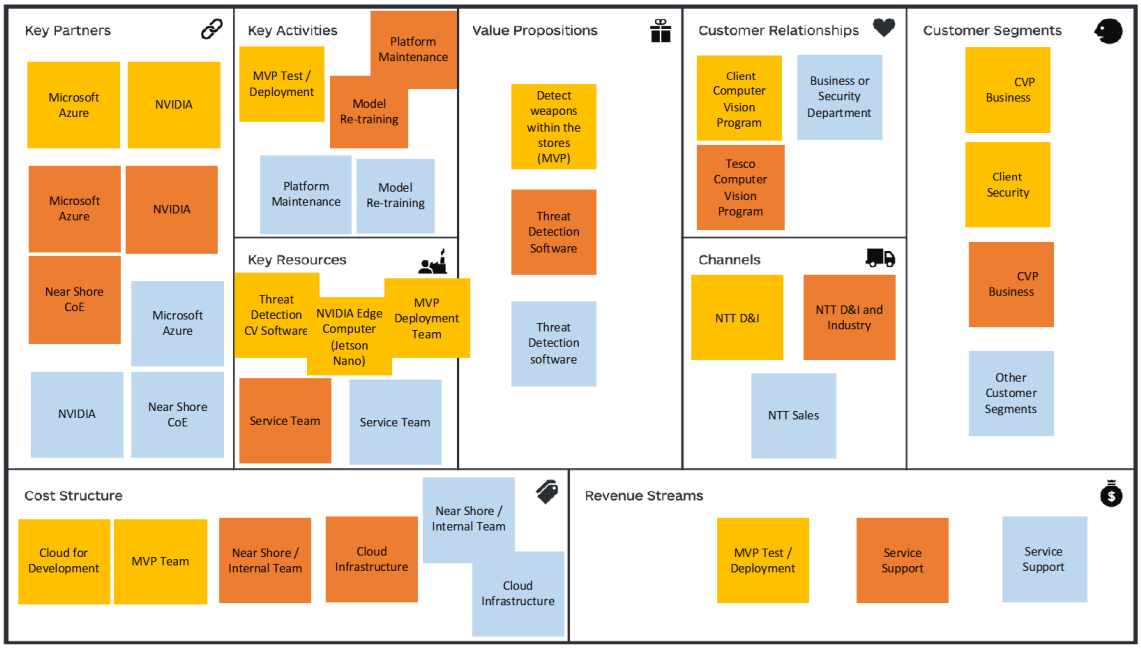

Figure 4 - Business Model Canvas

2. THEORETHICAL FRAMEWORK

2.1. DEEP LEARNING

"*Deep learning is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks*" [3]. Emerged after the 2000s computational growth, and with the winning of some complex machine learning competitions. After 20 years we still see deep learning mentioned everywhere.

Deep learning networks, such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), differ from traditional feed-forward multilayer networks. They have more neurons, enabling the representation of more sophisticated models. How layers t in deep learning networks are more specialized, such as using locally connected patches of neurons in CNNs and recurrent connections in RNNs. Deep learning networks require significant computing power to train, which has become more available in recent years. Additionally, deep learning networks can automatically extract features from data, eliminating the need for manual feature engineering. These factors have contributed to the success of deep learning networks in various applications [2].

Among the most known factors that contributed to this boost of deep learning use was the public availability of large, labeled datasets and the empowerment of parallel GPU computing, which enabled the transition from CPU-based to GPU-based training, accelerating the training process, which is a key feature used on the modeling phase of this project [4].

All this fueled a large variety of computer vision problems, such as object detection and face recognition which are the main ones related to this project.

2.2. DEEP LEARNING APPLICATIONS IN COMPUTER VISION

2.2.1. Computer Vision Introduction

Computer vision can be described as "*the automated extraction of information from digital image"s* [5]. Since images are merely sets of matrices without inherent meaning for computers, the objective is to train them to interpret the pixels in a manner that mimics human perception. The rise of deep learning had such an impact that in some tasks such as text recognition and face verification, it achieves performance levels even more accurately than humans.

There are a lot of tasks and ambitious problems being tackled, and this project covers some of them, specifically video analysis and object detection.

2.2.2. Neural Networks

NNs are inspired by how human brains work. In a neural network, multiple artificial neurons are organized into layers, with the output of one layer serving as the input to the next. By chaining

multiple layers together, a neural network can learn complex patterns in data and perform a wide range of tasks, such as image classification, speech recognition, and natural language processing.

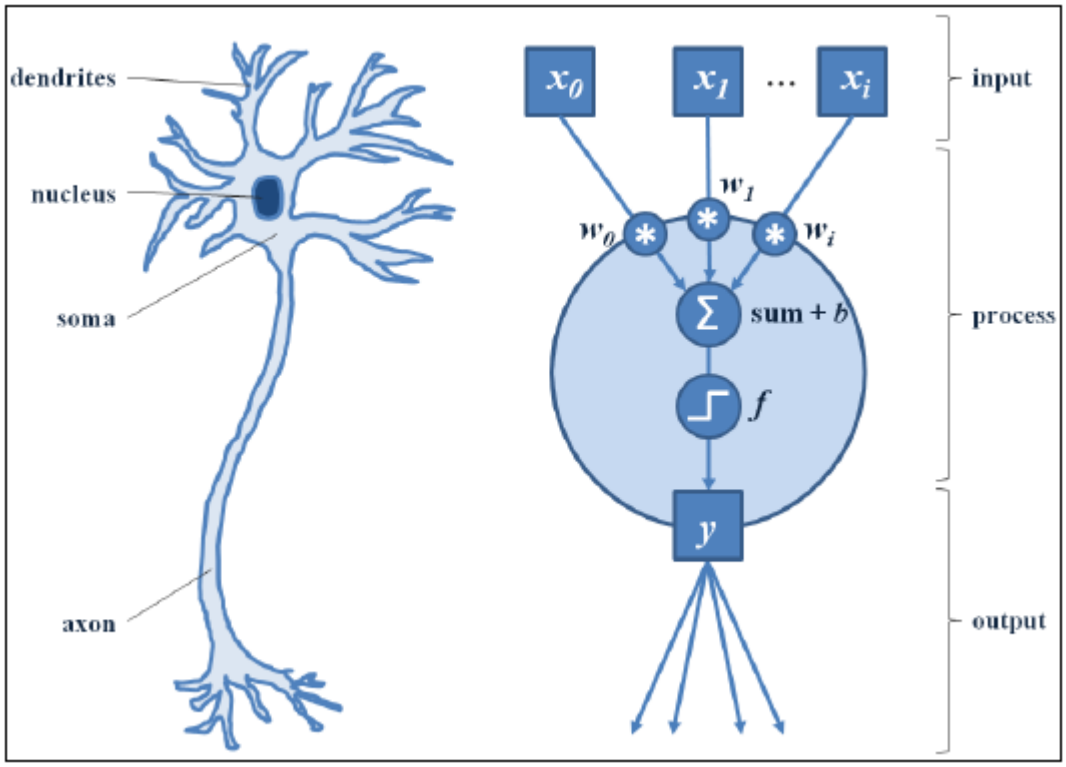

Figure 5 – Biological and artificial neurons [5]

The sum of the inputs is weighted, so each input is scaled depending on its corresponding weight. Together with the neuron's *bias*, they are the main parameters that we can adjust during the training phase to optimize the final results. Formalizing the process mathematically for one neuron that takes two inputs x_0 and x_1, weighted by a factor w_0 and w_1, respectively, (with an optional bias) we can express the input values as a horizontal vector, the weights as vertical vector and multiply them resulting in the following equation:

$$x \cdot w = \sum_i x_i w_i = x_1 w_1 + x_2 w_2$$

So, we can represent a simple artificial neuron in a network as it follows.

$$z = x \cdot w + b$$

Here, z is the output of the neuron, x is the input vector, w is the weight vector, and b is the bias. The weight vector determines the strength of the connections between the input and the output, and the bias determines the threshold of the activation function.

The equation represents the dot product between the input vector and the weight vector, with the addition of the bias term. This dot product captures the weighted sum of the inputs, and the bias term shifts the activation function to the left or right, which affects the range of outputs that can be produced by the neuron.

Before the neuron outputs, the signal has to pass a key component of the original perceptron, the **activation function**, which with linear inputs takes a binary form, returning 1 or a 0 (usually with t = 0).

$$y = f(z) = \begin{cases} 0 \text{ if } z < t \\ 1 \text{ if } z \geq t \end{cases}$$

In non-linearity cases (more complex behaviors) and continuous differentiability the most common functions are:

- The **sigmoid** function, $\sigma(z) = \dfrac{1}{1+e^{-z}}$ (with e the exponential function)
- The **hyperbolic tangent**, $\tanh(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$
- The **Rectified Linear Unit** (**ReLU**), $\mathrm{ReLU}(z) = \max(0, z) = \begin{cases} 0 \text{ if } z < 0 \\ z \text{ if } z \geq 0 \end{cases}$

To visualize the difference between each function, plots in the following figure:

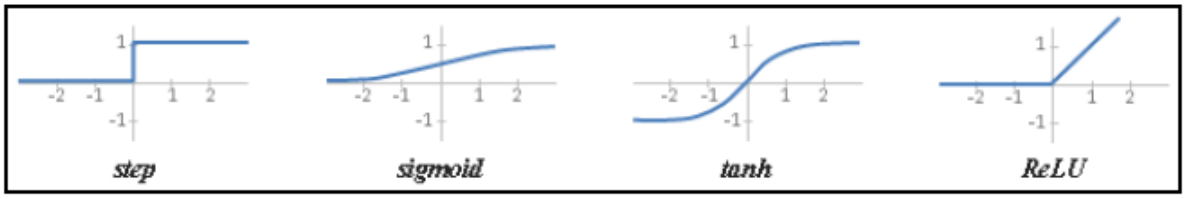

Figure 6 – Activation functions plots [5]

With this logic in mind, we modeled a simple artificial neuron, that can receive a signal, process it and out a value that can be forwarded to other neurons, building a network [5].

2.2.3. Convolutional Neural Networks

Knowing what Neural Networks are, we go a step deeper into the concept and went on the understanding what convolutional neural networks (CNNs) are and how these modern methods are trained to further improve their robustness. CNNs were introduced to solve some of the shortcomings of the original neural networks. The two main drawbacks of basic networks when dealing with images are **the explosive number of parameters** and **the lack of spatial reasoning**.

Images are very complex structures with a large number of values (*H* x *W* x *D* values with *H* indicating the image's height, *W* width, and *D* the depth). Even small single-channel images can have input vectors of size (e.g.) *28* x *28* x *1 = 784* values each and these numbers simply explode when considering larger RGB images or deeper networks.

Also, because their neurons receive all the values from the previous layer without any distinction (they are fully connected), these neural networks do not have the concepts of distance or spatiality. Spatial relations in the data are lost. Multidimensional data (e.g. images) could also be anything from column vectors to dense layers because their operations do not take into account the data dimensionality nor the positions of input values, more precisely this means that the idea of proximity between pixels is lost to fully connected (FC) layers, as all pixel values are combined by the layers without considering their original positions.

As it does not change the behavior of dense layers, to simplify, it *flattens* multidimensional inputs before passing them to these layers (reshapes them into column vectors). Neural layers would be smarter if they could consider the **spatial information**, meaning that some input values belong to the same pixel (channel values) or the same image region (neighbor pixels).

CNNs offers simple solutions to these shortcomings while working in the same way as the networks previously presented. First, CNNs can handle multidimensional data. For images, it takes as input the same three-dimensional data (*H* x *W* x *D*) and has its neurons arranged in a similar volume, and this leads to the second improvement of CNNs that unlike fully connected networks each neuron only has access to some elements in the neighboring region of the previous layer. This region is called the receptive field of the neurons:

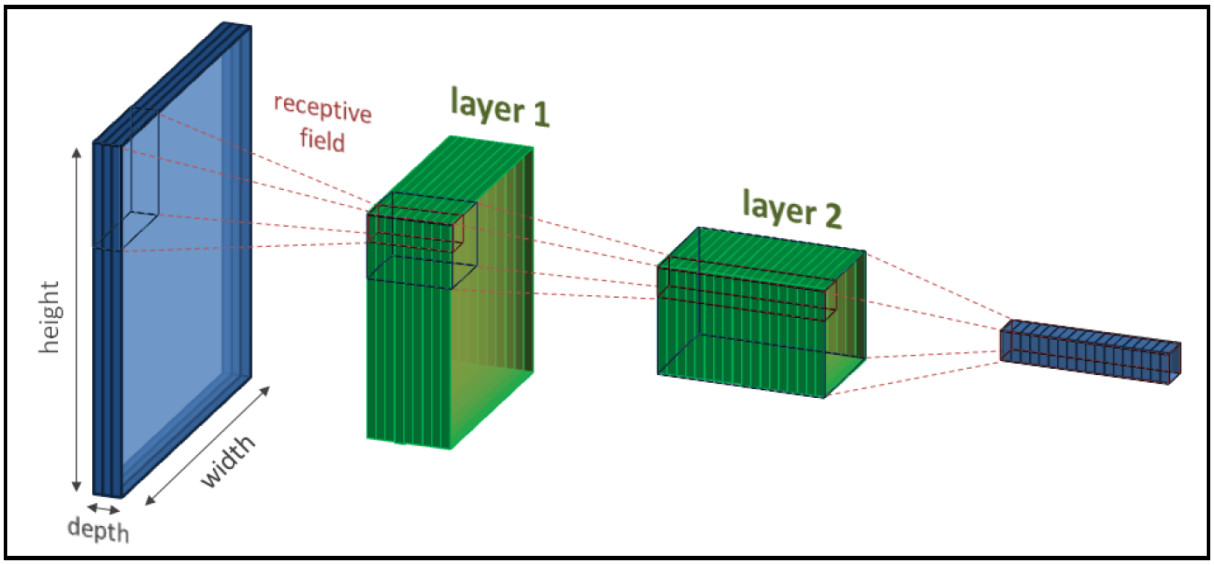

Figure 7 – CNN representation [7]

By linking neurons only to their neighboring ones in the previous layer, CNNs not only drastically reduce the number of parameters to train, but also preserve the localization of image features [7].

Now that we know the basic concept and logic of CNNs we can understand why they lay the robust training and optimization foundations of the models used for computer vision projects and used for the development of our solution.

2.2.4. TensorFlow

TensorFlow is an open-source library developed by Google primarily to simplify the deployment of machine learning solutions on various platforms.

TensorFlow's architecture is composed of a C++ layer, a python low-level API that wraps C++ sources, so when calling a python method in TensorFlow, it usually invokes C++ code behind the scenes but since python is considered to be easier to use this wrapper allows users to work more quickly. At the top layer is the high-level API made from two components, Keras and the Estimator API.

The Estimator API contains pre-made components that allow you to build machine learning models more easily, very similar to building blocks or templates. Keras is a user-friendly, modular, and extensible wrapper for TensorFlow that at first was designed as an interface to enable fast experimentation with neural networks.

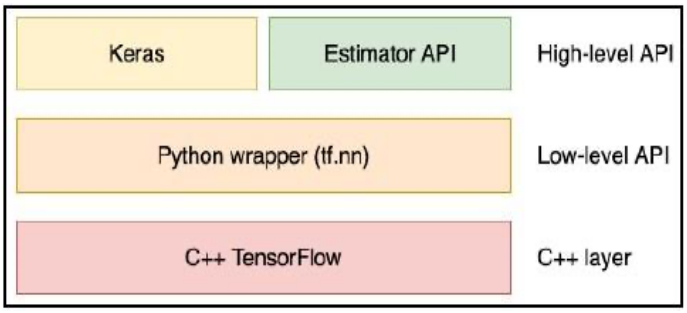

Figure 8 – Diagram of TensorFlow architecture [6]

In this project, we used TensorFlow 2 which was released with some new features and concepts. One of those core concepts is the tensors, which can be described as N-dimensional arrays that could take the form of a scalar, a vector, a 3D matrix, or an N-dimensional matrix. This component is used to store mathematical values which can be fixed values created using *tf.constant* or changing values created using *tf.variable*.

TensorFlow uses tensors as inputs and as outputs, and a component that transforms one into the other is called an operation, therefore a computer vision model is composed of multiple operations. These operations are represented using what is called a directed acyclic graph (DAC), also referred to as a TensorFlow graph.

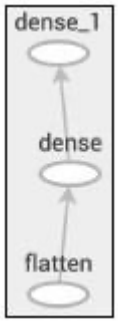

Figure 9 – A simplified graph corresponding to a model [6]

Each node is composed of smaller operations such as matrix multiplications and additions and while very simple this graph represents the different layers of a simple model in the form of multiple operations. By relying on graphs, TensorFlow can run part of the operations on the CPU and another part on the GPU, run different parts of the graph on different machines in case of a distributed model, and optimize the graph to avoid unnecessary operations, lending to better computational performance.

Another useful concept is backpropagating errors using gradient tape. In short, TensorFlow computes the results of an operation instead of storing the operation, with no information on the operation and its inputs it would be impossible to automatically differentiate the loss operation. That is where the gradient tape makes a difference, by running our loss computation in the context of *tf.gradienttape,* TensorFlow will automatically record all operations and allow us to replay them backward afterward.

```
def train_step():
with tf.GradientTape() as tape:
loss = tf.math.abs(A * X - B)
dX = tape.gradient(loss, X)
print('X = {:.2f}, dX = {:2f}'.format(X.numpy(), dX))
X.assign(X - dX)
for i in range(7):
train_step()
```

The code above defines one training step. Every time *train_step* is called, the loss is computed in the context of the gradient tape, then the context is used to compute the gradient. The *X* variable is then updated, and we can see it converging toward the value that solves the equation:

```
X = 20.00, dX = 3.000000
X = 17.00, dX = 3.000000
X = 14.00, dX = 3.000000
X = 11.00, dX = 3.000000
```

So, for innovative models or when experimenting, the gradient tape is a powerful tool that allows automatic differentiation without much effort [6].

2.3. NVIDIA DEEPSTREAM

2.3.1. What is NVIDIA Deepstream?

"*NVIDIA's Deepstream SDK is a complete streaming analytics toolkit based on GStreamer for AI-based multi-sensor processing, video, audio, and image understanding. It's ideal for vision AI developers, software partners, startups, and OEMs building IVA apps and services*" [8]. With this toolkit, developers can construct stream processing pipelines that include neural networks and other advanced processing tasks, such as tracking, video encoding/decoding, and video rendering. These pipelines enable real-time analysis on video, image, and sensor data, facilitating timely insights and analytics.

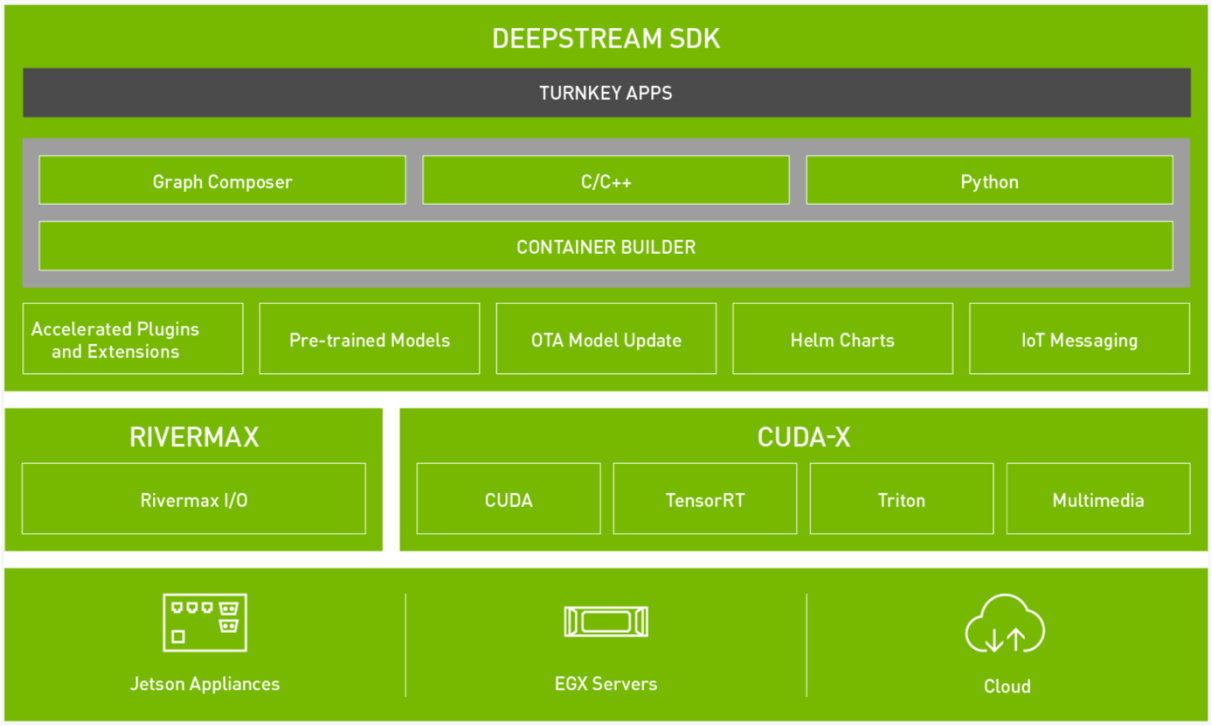

Figure 10 – NVIDIA Metropolis [8]

Deepstream takes the streaming data as input – from a USB/CSI camera, video from a file, or streams over RTSP (which is the available stream format for us) and uses "*AI and computer vision to generate insights from pixels for a better understanding of the environment*" [9]. This toolkit has the potential to serve as a fundamental building block for various video analytic applications, such as safety monitoring, retail self-checkout, and our specific focus, object detection.

DeepStream supports application development in C/C++ and Python, offering Python bindings for extra ease of use. The toolkit includes pre-built reference applications in C/C++ and Python to simplify the development process. The core SDK includes hardware accelerator plugins that leverage accelerators such as VIC, GPU, DLA, NVDEC, and NVENC, enabling compute-intensive tasks to be offloaded to dedicated accelerators for optimal performance in video analytics applications.

A notable feature of DeepStream is its secure bi-directional communication between edge and cloud environments. The toolkit includes built-in security protocols such as SASL/Plain authentication using username/password and 2-way TLS authentication to ensure data security.

DeepStream uses various NVIDIA libraries from the CUDA-X stack, including CUDA, TensorRT, NVIDIA Triton Inference server, and multimedia libraries. TensorRT accelerates AI inference on NVIDIA GPUs, and DeepStream abstracts these libraries into plugins, simplifying the development of video analytic pipelines without the need to learn each library.

DeepStream is optimized for NVIDIA GPUs and can integrate on embedded edge devices running the Jetson platform, as well as data center GPUs like T4. DeepStream applications can be containerized using NVIDIA container Runtime and are available on NGC, the NVIDIA GPU cloud registry [9].

2.3.2. Deepstream Graph Architecture

"*Deepstream is an optimized graph architecture built using the open-source GStreamer framework. The graph below shows a typical video analytic application starting from input video to outputting insights. All the individual blocks are various plugins that are used. At the bottom are the different hardware engines that are utilized throughout the application. Optimum memory management with zero-memory copy between plugins and the use of various accelerators ensure the highest performance*" [9].

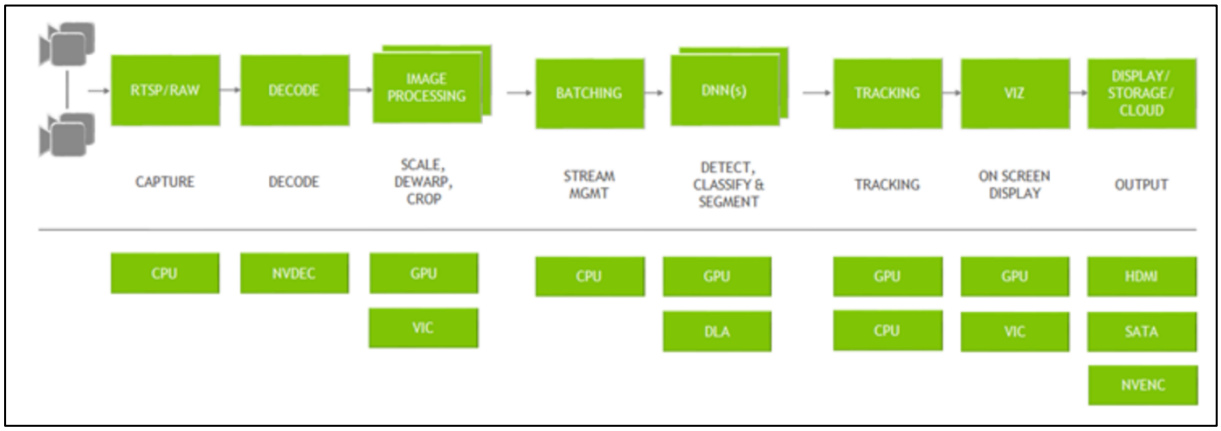

Figure 11 – Full Deepstream Architecture [9]

DeepStream provides a comprehensive suite of GStreamer plugins that serve as foundational building blocks for constructing efficient video analytics pipelines. These plugins are specifically designed to harness hardware acceleration capabilities and optimize performance in various video processing tasks:

- The streaming data can be captured from diverse sources such as RTSP, local file systems, or cameras and processed using the CPU.
- Optionally, image pre-processing can be applied after decoding, including image de-warping for correcting distortions from FishEye lenses or color space conversion.

- Frames are then batched for optimal inference performance using the gst-nvstreammux plugin.
- Inference can be performed using TensorRT, NVIDIA's inference accelerator runtime, or native frameworks such as TensorFlow or PyTorch with the Triton inference server. GPU or DLA (Deep Learning Accelerator) can be utilized for inference on Jetson AGX Xavier and Xavier NX.
- Object tracking can be performed after inference using built-in reference trackers in the SDK.
- Visualization artifacts such as bounding boxes, segmentation masks, and labels can be created.

DeepStream offers various options for outputting the results, such as rendering the output with bounding boxes on the screen, saving the output to a local disk, streaming out over RTSP, or sending metadata to the cloud. Built-in broker protocols such as Kafka, MQTT, AMQP, and Azure IoT are available, and custom broker adapters can be created to suit specific requirements [9].

2.3.3. Key Features

DeepStream provides comprehensive support for AI models, specifically for object detection and segmentation, encompassing cutting-edge models such as SSD, YOLO, FasterRCNN, and MaskRCNN. Moreover, DeepStream facilitates the integration of custom functions and libraries, catering to unique requirements.

The versatility of DeepStream spans from rapid prototyping to full-scale production-level solutions, affording the flexibility to choose the most suitable inference path. The platform seamlessly integrates with the NVIDIA Triton™ Inference Server, allowing for the deployment of models in native frameworks such as PyTorch and TensorFlow for inference. Additionally, DeepStream leverages the NVIDIA TensorRT™ to enable high-throughput inference with support for multi-GPU, multi-stream, and batching, leading to optimal performance.

In addition to supporting native inference, DeepStream's capabilities extend to communication with independent/remote instances of the Triton Inference Server via gRPC, enabling the implementation of distributed inference solutions. This empowers efficient and scalable deployment of AI models in distributed environments, ensuring robustness and scalability of the inference process.

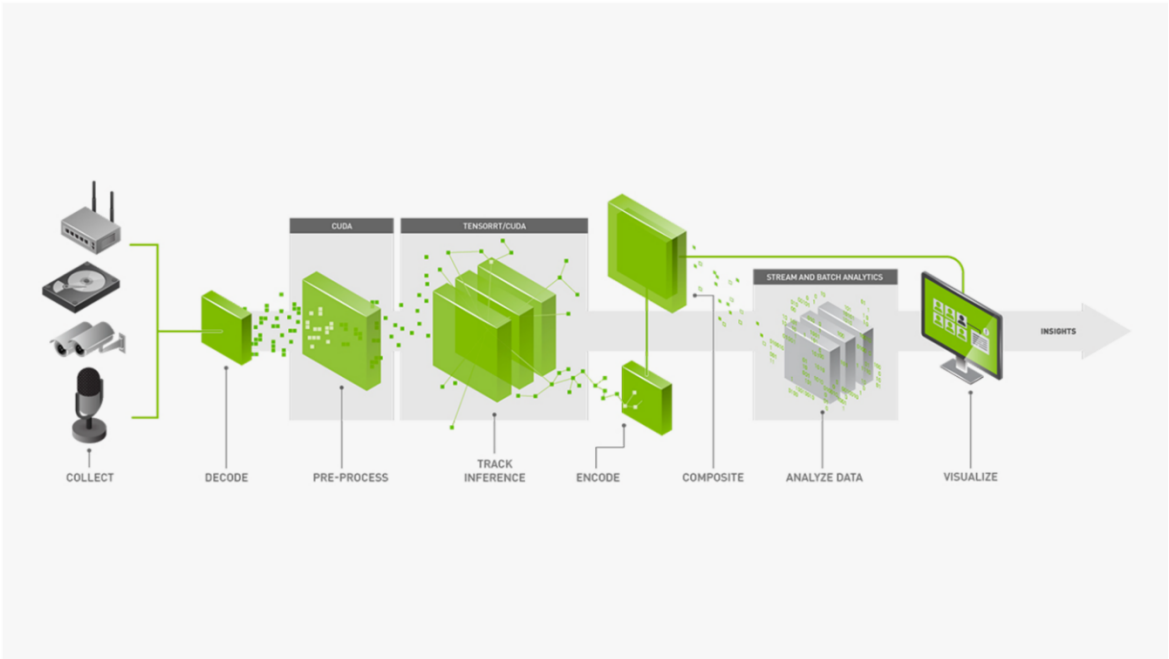

Figure 12 – Inference workflow [8]

Furthermore, DeepStream facilitates seamless integration with the Transfer Learning Toolkit (TAO Toolkit) within the NVIDIA Metropolis ecosystem, enabling accelerated development and enhanced real-time performance of vision AI systems. The TAO Toolkit enables developers to adapt and optimize production-quality vision AI models, including SSD, MaskRCNN, YOLOv4, RetinaNet, and other state-of-the-art models, while DeepStream offers turnkey integration of these models for deployment.This integrated approach empowers the use of pre-trained models and transfer learning techniques, resulting in a streamlined workflow for building end-to-end vision AI applications. By leveraging the TAO Toolkit in conjunction with DeepStream, developers can achieve faster development cycles and superior performance in real-time scenarios, making it a valuable tool for academic research and development in the field of vision-based AI [9].

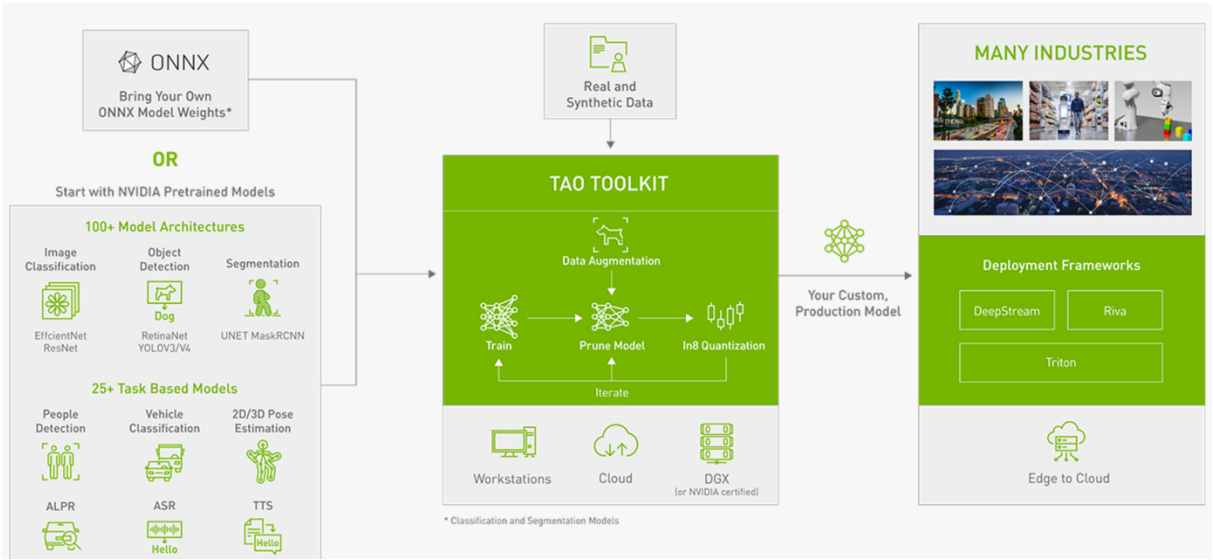

Figure 13 – TAO Toolkit integration [11]

3. SOFTWARES AND TOOLS

3.1. COMPUTER VISION ANNOTATION TOOL

3.1.1. Introduction

CVAT is a free and open-source interactive video and image annotation tool for computer vision. It was designed to provide users with a set of convenient instruments for annotating digital images and videos since data scientists need lots of annotated data to train (in our case) the CNNs at the core of AI workflows and obtain that data with the necessary quality is a huge and time-consuming challenge.

CVAT supports supervised machine learning tasks pertaining to object detection, image classification, image segmentation, and 3D data annotation. It allows users to annotate images with multiple tools (boxes, polygons, cuboids, circles, skeletons, etc).

3.1.2. CVAT Setup basics

In our particular case, we used machines with Windows 10 and for that, we needed to install a few other tools for further use:

- Install WSL2 (Windows Subsystem for Linux);
- Download and install Docker Desktop;
- Download and install git for Windows;
- Download and install Google Chrome, it is the only browser that supports CVAT;

Open the Git Bash application and clone the CVAT source code from the GitHub repository with the following command.

```
git clone https://github.com/opencv/cvat
cd cvat
```

Then we must run docker containers. It will take some time to download the latest CVAT release and other required images like Postgres, Redis, etc. from DockerHub and create containers.

```
docker-compose up -d
```

After this, we create a superuser. A superuser can use an admin panel to assign correct groups to other users.

```
# enter docker image first
Docker exec -it cvat_server /bin/bash
# then run
```

```
Python3 ~/manage.py createsuperuser
```

Now we must choose a username and password for the admin account and we're good to go.

Finally, we open the Google Chrome browser and go to *localhost:8080,* type the previously created superuser login/password account and we can create our first task.

3.2. AZURE VIRTUAL MACHINE

Azure virtual machines are dynamic and scalable computing resources that provision on demand in the Microsoft Azure cloud environment. These virtual machines are utilized on host applications when greater control over the computing environment is desired, as compared to other available computing resources. By leveraging virtual machines, organizations can benefit from virtualization without the need for investment in physical hardware. However, it is paramount to note that virtual machines require management tasks, such as configuration, patch management, and software installation, like traditional servers.

One of the notable advantages of utilizing Azure virtual machines is the ease and expediency of their setup, making them well-suited for deploying development and test environments. Moreover, organizations often employ Azure virtual machines for hosting their applications in Microsoft Azure due to the pay-as-you-go pricing model, which enables them to only pay for virtual machines when they are actively used. Furthermore, virtual machines in Azure can be utilized to extend on-premises data centers to the cloud through site-to-site VPN connectivity, facilitating communication between on-premises environments and virtual machines attached to a virtual network in Azure.

When designing an application infrastructure that incorporates Azure virtual machines, several crucial factors must be considered. These include establishing appropriate naming conventions for virtual machines and selecting optimal deployment locations based on proximity to end users. Additionally, determining the appropriated virtual machine sizing based on performance and resource requirements and understanding and managing CPU and virtual machine quotas imposed by Microsoft Azure are crucial factors to consider. Furthermore, careful consideration of the operating system and configuration requirements for virtual machines is essential to ensure optimal performance and functionality of the deployed applications.

3.3. NVIDIA JETSON NANO

"*NVIDIA® Jetson Nano™ Developer Kit is a small, powerful computer that lets you run multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing. All in an easy-to-use platform that runs in as little as 5 watts*" [10]. To use the system, it is necessary to insert a microSD card containing the system image, boot the developer kit, and commence using the NVIDIA JetPack SDK, which is uniformly employed across the entire NVIDIA Jetson™ product family. JetPack is compatible with NVIDIA's state-of-the-art AI platform for training and deploying AI software, simplifying the development process for researchers and practitioners. This

powerful tool facilitates exploration of the comprehensive functionalities of the DeepStream SDK for video and image understanding based on AI, as well as enables access and effective utilization of the Nvidia Transfer Learning Toolkit.

3.4. NVIDIA TRANSFER LEARNING TOOLKIT

3.4.1. Overview

"*The NVIDIA Transfer Learning Toolkit (TLT) is used with NVIDIA pre-trained models to create custom Computer Vision (CV) and Conversational AI models with our data. Training AI models using TLT does not require expertise in AI or deep learning. A simplified Command Line Interface (CLI) abstracts away all the AI framework complexity and enables users to build production-quality AI models using a simple spec file and one of the NVIDIA pre-trained models. A basic understanding of deep learning and minimal to zero coding is required"* [11].

With TLT we can:

- Refine models for computer vision (CV) applications, including object detection, image classification, segmentation, character recognition (CR), and point estimation, by leveraging NVIDIA's pre-trained CV models.
- Adapt pre-trained models for conversational AI applications, such as automatic speech recognition (ASR) or natural language processing (NLP), using NVIDIA's pre-trained conversational AI models.
- Incorporate new classes into an existing pre-trained model to expand its capabilities.
- Re-train a model to adapt to different use cases or scenarios.
- Apply a model pruning capability on CV models to reduce the overall size and optimizing its efficiency and performance.

With the Transfer Learning Toolkit (TLT), we can create customized artificial intelligence (AI) models by modifying the training hyperparameters specified in each spec file. It can be easily accomplished by following the provided guide, which includes sample spec files and parameter definitions for all models supported by TLT, in either the Computer Vision or Conversational AI section based on the desired model. In addition to creating accurate AI models, TLT also offers the capability to optimize models for inference, ensuring the highest throughput for deployment scenarios..

TLT *"is a Python package hosted on the NVIDIA Python Package Index. It interacts with lower-level TLT dockers available from the NVIDIA GPU Accelerated Container Registry (NGC) and its containers come pre-installed with all dependencies required for training" [11].* The CLI executes from Jupyter notebooks that include Docker containers and offer a set of straightforward commands, including data augmentation, training, evaluation, inference, pruning, and export, as part of the TLT workflow. The outcome of this workflow is a trained model that can be deployed for inference on NVIDIA devices using DeepStream, TensorRT, Riva, and the TLT CV Inference Pipeline, providing a comprehensive solution for custom AI model development and deployment. The TLT application layer, built on top of

CUDA-X, includes NVIDIA Container Runtime for GPU acceleration, CUDA, cuDNN, and TensorRT for optimized and accelerated deep learning model deployment.

3.4.2. Pre-trained Models

There are two types of pre-trained models:

- Purpose-built pre-trained models, built for a specific task, offer high accuracy, and can be used directly for inference or as a starting point for transfer learning with TLT using our own dataset.
- General purpose vision models, where pre-trained weights serve as a valuable starting point to build more complex models, as they are trained on Open image datasets and offer improved performance compared to random weight initialization.

There are 100+ permutations of model architecture with the general-purpose vision models.

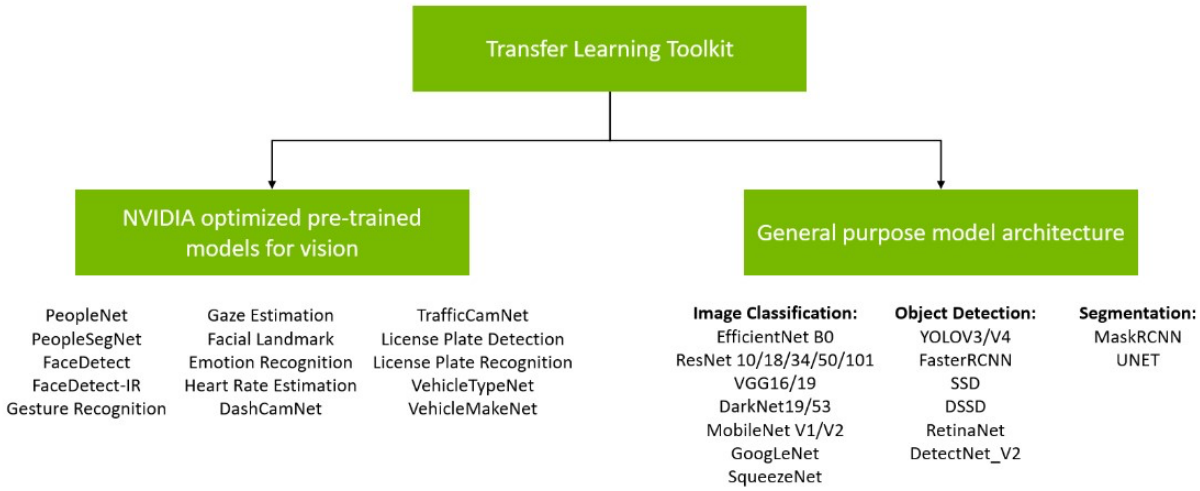

Figure 14 – TLT Pre-trained model's overview [11]

"*Purpose-built models are built for high accuracy and performance*" [11]. These pre-trained models design facilitate the deployment for enumerous applications, such as smart cities, retail, public safety, healthcare, and also they can be fine-tuned with the users' own data. All models are trained on proprietary images and demonstrate high accuracy on NVIDIA test data. Detailed information about each model is in individual model cards, including typical use cases and key performance indicators as summarized in the table below. For instance, PeopleNet is well-suited for people detection and counting in buildings, retail spaces, hospitals, and more. TrafficCamNet and DashCamNet are ideal for traffic applications, enabling vehicle detection and tracking on roads.

This table represents the inference performance measured using the trtexec tool in TensorRT samples.

			Nano		Tx2 NX		Xavier NX						AGX Xavier						T4		A100	
Model Arch	Inference resolution	Precision	GPU (FPS)	Batch Size	GPU (FPS	Batch Size	GPU (FPS)	Batch Size	DLA1 (FPS)	Batch Size	DLA2 (FPS)	Batch Size	GPU (FPS)	Batch Size	DLA1 (FPS)	Batch Size	DLA2 (FPS)	Batch Size	GPU (FPS)	Batch Size	GPU (FPS)	Batch Size
PeopleNet – ResNet18	960x544x3	INT8	14	1	36	2	218	8	72	4	72	4	395	16	97	8	97	8	1424	32	8462	256
PeopleNet- ResNet34	960x544x3	INT8	11	1	31	2	182	8	58	4	58	4	314	16	75	8	75	8	1043	16	6001	64
TrafficCamNet	960x544x3	INT8	19	1	51	2	264	8	105	4	105	4	478	16	140	8	140	8	1703	64	9520	256
DashCamNet	960x544x3	INT8	18	1	45	2	254	8	100	4	100	4	453	16	133	8	133	8	1666	64	9521	256
FaceDetectIR	384x240x3	INT8	101	4	252	8	1192	32	553	16	553	16	2010	64	754	16	754	16	9720	64	50541	256
VehilceMakeNet	224x224x3	INT8	173	8	448	16	1871	32	700	16	700	16	3855	64	945	16	945	16	15743	128	89141	1024
VehicleTypeNet	224x224x3	INT8	120	8	318	8	1333	32	678	16	678	16	3047	64	906	16	906	16	11918	128	52809	512
PeopleSegNet	960x576x3	INT8	0.6	1	2.5	1	8.5	1	5.7	1	5.7	1	12.2	1	8.2	1	8.2	1	35	4	202	16
FaceDetect	736x416x3	INT8	27	2	70	4																
LPD	640x480x3	INT8	66	4	178	8	770	16	194	8	194	8	1370	32	256	16	256	16	5921	128	21931	256
LPR	96x48x3	FP16	50	4	134	8	564	16	-	N/A	-	N/A	1045	32	-	N/A	-	N/A	3870	128	26600	256
Facial landmark	80x80x1	FP16	125	4	319	8	747	16	-	N/A	-	N/A	1451	32	-	N/A	-	N/A	4735	128	23117	256
GazeNet	224x224x1, 224x224x1, 224x224x1, 25x25x1	FP16	98	4	280	8	923	32	-	N/A	-	N/A	1627	64	-	N/A	-	N/A	5219	256	26534	1024
HeartRateNet	36x36x3	FP16	504	16	1300	32	2771	64	-	N/A	-	N/A	5405	64	-	N/A	-	N/A	20046	256	103K	1024
GestureNet	160x160x3	FP16	82	4	244	8	942	32	-	N/A	-	N/A	1646	64	-	N/A	-	N/A	5660	256	34086	1024
EmotionNet	1x136x1	FP16	108K	256	216K	512	957K	1024	-	N/A	-	N/A	1.72M	1024	-	N/A	-	N/A	5.3M	1024	9M	1024
PeopleSemSegNet	960x544x3	INT8	1.4	1	5.7	1	17	1	8.7	1	8.7	1	28	1	12	1	12	1	103	4	519	16
2D Body Pose Net	288x384x3	INT8	4.7	1	12.3	1	97	4	-	N/A	-	N/A	166	8	-	N/A	-	N/A	563	8	2686	8

Figure 15 – Performance of Pre-trained Models [11]

Users can train general-purpose image classification and object detection models using a variety of architectures, including ResNet, EfficientNet, VGG, MobileNet, GoogLeNet, SqueezeNet, DarkNet, YOLOV3/V4, FasterRCNN, SSD, RetinaNet, DSSD, and DetectNet_v2. [11].

3.5. PYTHON

"*Python is an interpreted, object-oriented, high-level programming language with dynamic semantics"* [12]. Its high-level built-in data structures, dynamic typing, and dynamic binding make it ideal for Rapid Application Development and as a scripting or glue language for connecting existing components. Its simple and readable syntax reduces the cost of program maintenance, and support for modules and packages promotes modularity and code reuse. The availability of Python interpreters and an extensive standard library for all major platforms without charge allows for easy distribution. Python's edit-test-debug cycle is fast due to the lack of a compilation step, and debugging is made easy with exceptions and stack trace. The source-level debugger, written in Python itself, showcases Python's introspective power. Additionally, adding print statements for debugging is effective due to Python's fast edit-test-debug cycle. [12].

3.6. MOBAXTERM

MobaXterm is a toolbox for remote computing where in a single application we have at our disposal loads of functions that are tailored for our remote jobs more simply and it provides all the important remote network tools (SSH, X11, RDP, VNC, FTP, MOSH, ...) and Unix commands (bash, ls, cat, sed, grep, awk, rsync, ...). "*There are many advantages of having an All-In-One network application for our remote tasks, e.g. when we use SSH to connect to a remote server, a graphical SFTP browser will automatically pop up to directly edit the remote files*" [13].

In our project, we use MobaXterm to launch a remote session on the VM with an SSH key. That way, using Unix commands we manage the data stored inside and apply the necessary files to train our model and export it easily.

3.7. DOCKER

"*Docker is an open platform for developing, shipping, and running applications. Docker enables us to separate applications from our infrastructure so we can deliver software quickly. With Docker, we can manage our infrastructure in the same ways we manage our applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, we reduce the delay between writing code and running it in production*" [14].

"*Docker provides the ability to package and run an application in a container*" [14]. Containers provide isolation, security, and portability, allowing for concurrent execution of multiple lightweight and self-contained application environments on a single host, facilitating easy sharing and consistent behavior across different environments.

Docker operates on a client-server architecture, where the Docker client communicates with the Docker daemon responsible for building, running, and distributing containers. The Docker client and daemon can be on the same system or connected remotely, using a REST API, UNIX sockets, or a network interface. Docker Compose is another Docker client that facilitates working with containerized applications composed of multiple containers.

"*The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services*" [14].

"*The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon*" [14].

"*Docker Desktop is an easy-to-install application for your Mac, Windows, or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see Docker Desktop"* [14].

"*A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your private registry. When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry*" [14].

"*When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects*" [14].

"*An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization*" [14]. Docker images can be created using a Dockerfile with simple syntax to define the steps, resulting in lightweight, small, and fast images that are composed of layers and can be built from scratch or obtained from a registry.

"*A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state"* [14]. Docker containers provide default isolation from other containers and the host machine, and their state changes are lost when the containers are removed, making them lightweight and flexible for application deployment.

4. MODELLING

4.1. STATUS OF THE PROJECT

4.1.1. Overview

By the time I got assigned to this project, there was already a system designed and tested. This system was based on the work that an Nvidia developer that goes by the name of *dusty-nv* in GitHub. On his GitHub, there is a repository with a shared work (https://github.com/dusty-nv/pytorch-ssd/tree/8ed842a408f8c4a8812f430cf8063e0b93a56803) that consists of the implementation of an SSD (Single Shot Detector) for object detection with PyTorch and using MobilleNet architecture.

In this git repository, we have all the necessary scripts (written in python) for the training, evaluation, and export of the trained model, we must prepare the input with our labeled data and run the scripts with our specifications.

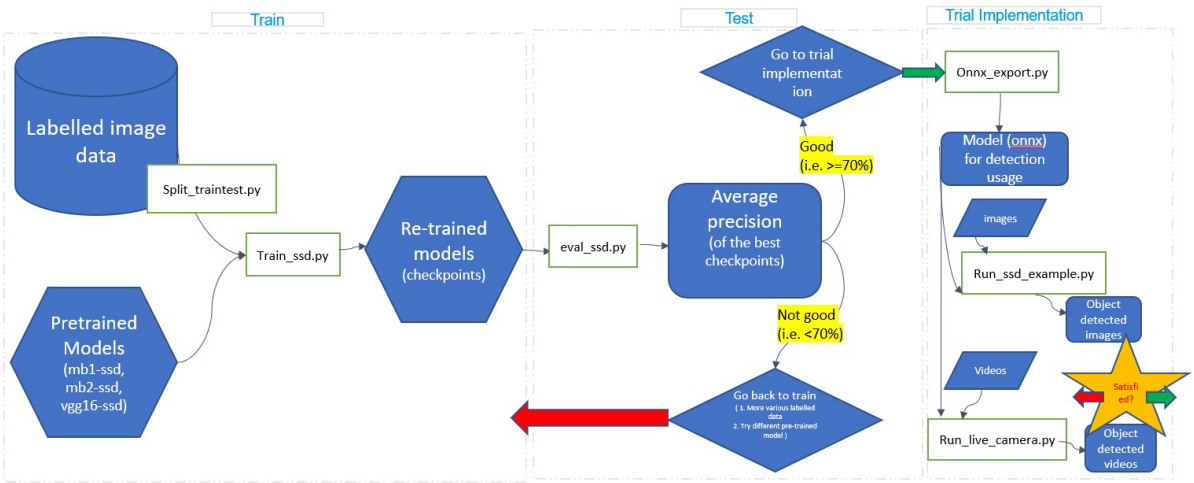

Figure 16 – Object Detection Workflow

To start the process, we have to get a usable input, meaning we have to label data with the classes we want to detect (e.g. knife, face mask…), which will be explained in more detail ahead.

Having the correct input, we call directly the *split_traintest.py* to separate our data into two datasets, one for training and the other for testing later. Then we run the *train_ssd.py* script, with the selected pre-trained model and subsequent specifications. When the training concludes we check the detection accuracy of our model for each class by calling the *eval_ssd.py*. With the output, we evaluate if the model is ready to be exported and tested or if we need to retrain it with more data labeled with a specific class or even with a different pre-trained model, the accuracy benchmark is 70% for each class detection.

Finally, if the accuracy is over our minimum benchmark, we call the *onnx_export.py*, which will give us a *.onnx* file containing the model ready to be tested with the *run_ssd_example.py*, where we confirm if our model detects anything in a set of random images. After we confirm detections, the model is stored in our VM and sent to the Jetson Nano, to run on the live CCTV feed connected to that machine.

4.1.2. Scripts

As mentioned above, in the training process with PyTorch we use 5 scripts:

- *split_traintest.py*
- *train_ssd.py*
- *eval_ssd.py*
- *onnx_export.py*
- *run_ssd_example.py*

The first is the *split_traintest.py*, which is a Python script that splits annotated images into training, validation, and test datasets. The script uses the *ArgumentParser* module to allow the user to input the path to the annotations directory and the images split text output directory. The *Path()* object is used to format the input paths according to the operating system being used. The *os* module's *listdir* function is used to get a list of all the files in the annotations directory. The names of the files are then extracted, split into training and test datasets using the '*train_test_split*' function from scikit-learn, and written to separate text files in the images split txt output directory.

```
### split the whole dataset to train and test: 8:2
from sklearn.model_selection import train_test_split
X_train, X_test = train_test_split(name_list, test_size=0.20, random_state=
1)

### write the names to sperate txt files: train_val and test
train_list = open(train_filename,'w')
for element in X_train:
     train_list.write(element)
     train_list.write('\n')
train_list.close()

test_list = open(test_filename,'w')
for element in X_test:
     test_list.write(element)
     test_list.write('\n')
test_list.close()
```

With the input data for the *train_ssd.py* script, the train can begin. This is a script for training and evaluating a single-shot multi-box detection model using the PyTorch deep learning framework. It supports several base architectures (VGG-16, MobileNet-V1, MobileNet-V2) and allows for training on different datasets (PASCAL VOC, OpenImages). The script covers the following steps:

1. Setting up logging and command line argument parsing.
2. Loading datasets and defining data loaders.
3. Defining the base architecture and customizing it with the desired parameters.
4. Setting up the loss function and optimizer.
5. Initializing the model and running training and evaluation.
6. Logging training statistics using TensorBoard.

This script uses several libraries such as *argparse*, *torch*, and *torchvision*. The *argparse* module is used to define command-line arguments for various hyperparameters and settings, and once these hyperparameters and settings are defined, the script will train the detector according to those specifications.

Parameters Groups	Parameters	Description
Parameters for network	--net	The network architecture, it can be mb1-ssd, mb1-lite-ssd, mb2-ssd-lite or vgg16-ssd
Parameters for datasets	--dataset-type	Specify dataset type. Currently supports voc and open_images
	--data	Dataset directory path
Parameters for tensorboad directory	--tensorboard-log-dir	tensorboard loss plotting data
Parameters for loading pretrained basenet or checkpoints	--pretrained-ssd	Pre-trained base model
Parameters for SGD	--learning-rate	initial learning rate
	--base-net-lr	initial learning rate for base net
Train Parameters	--model-dir	Directory for saving checkpoint models
	--batch-size	Batch size for training
	--workers	Number of workers used in dataloading
	--epochs	the number epochs
Parameters for cuda card	--cuda-card	Define the GPU card to be used during the training

Table 2 – The *train_ssd.py* arguments

The '*train()*' is the function that trains a deep learning model on an object detection task. The input includes a data loader, a network model, a loss criterion, an optimizer, and a device (GPU in this case is). During each iteration, the gradients of the model parameters are computed and updated using the optimizer. The running loss, regression loss, and classification loss are computed and logged every "*debug_steps*" iterations. The TensorBoard scalar values for the average loss, average regression loss, and average classification loss are also recorded. Additionally, the network graph is added to TensorBoard after the training is completed.

```
def train(loader, net, criterion, optimizer, device, debug_steps=100, epoch
=-1):
    net.train(True)
    running_loss = 0.0
    running_regression_loss = 0.0
    running_classification_loss = 0.0
    for i, data in enumerate(loader):
        images, boxes, labels = data
        images = images.to(device)
        boxes = boxes.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        confidence, locations = net(images)
        regression_loss, classification_loss = criterion(confidence, locati
ons, labels, boxes)  # TODO CHANGE BOXES
        loss = regression_loss + classification_loss
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        running_regression_loss += regression_loss.item()
        running_classification_loss += classification_loss.item()
        if i and i % debug_steps == 0:
            avg_loss = running_loss / debug_steps
            avg_reg_loss = running_regression_loss / debug_steps
            avg_clf_loss = running_classification_loss / debug_steps
            logging.info(
                f"Epoch: {epoch}, Step: {i}/{len(loader)}, " +
                f"Avg Loss: {avg_loss:.4f}, " +
                f"Avg Regression Loss {avg_reg_loss:.4f}, " +
                f"Avg Classification Loss: {avg_clf_loss:.4f}"
            )
```

```
            running_loss = 0.0
            running_regression_loss = 0.0
            running_classification_loss = 0.0
            tb.add_scalar('Train_Loss', avg_loss, epoch)
            tb.add_scalar('Train_Regression_Loss', avg_reg_loss, epoch)
            tb.add_scalar('Train_Classification_Loss', avg_clf_loss, epoch)
    tb.add_graph(net, images)
```

Then the '*test()'* function is used to evaluate the performance of the object detection model. It sets the network to evaluation mode by calling '*net.eval()*', and then calculates the loss on the validation dataset. The function loops over the validation data and calculates the loss, regression loss and classification loss. The loss is calculated by calling the '*criterion'* function and passing it the network's output, '*confidence'* and '*locations'*, along with the ground truth '*labels* 'and '*boxes*.' The loss is accumulated for each batch and divided by the number of batches to obtain the average loss for the entire validation dataset.

```
def test(loader, net, criterion, device):
    net.eval()
    running_loss = 0.0
    running_regression_loss = 0.0
    running_classification_loss = 0.0
    num = 0
    for _, data in enumerate(loader):
        images, boxes, labels = data
        images = images.to(device)
        boxes = boxes.to(device)
        labels = labels.to(device)
        num += 1

        with torch.no_grad():
            confidence, locations = net(images)
            regression_loss, classification_loss = criterion(confidence, lo
cations, labels, boxes)
            loss = regression_loss + classification_loss

        running_loss += loss.item()
        running_regression_loss += regression_loss.item()
        running_classification_loss += classification_loss.item()
    return running_loss / num, running_regression_loss / num, running_class
ification_loss / num
```

At the end of the function, it returns the average of the total loss, average regression loss, and average classification loss.

The next step is to evaluate the model's precision with the *eval_ssd.py* script.

It has two functions, the first one is the '*group_annotation_by_class()*', which is the function that groups the annotations in the dataset, loops over all of those annotations and extracts information such as the ground-truth bounding boxes, the class labels, and whether the annotation is considered difficult or not. The extracted information is then stored in three dictionaries: '*true_case_stat'*, '*all_gt_boxes'*, and '*all_difficult_cases'*. The '*true_case_stat'* dictionary stores the number of true (i.e.,

non-difficult) cases for each class. The '*all_gt_boxes'* dictionary stores the ground-truth bounding boxes for each image and class, and the '*all_difficult_cases'* dictionary stores the difficulty status for each annotation for each class and image. Finally, the function converts the lists of bounding boxes and difficulties into tensors and returns the three dictionaries.

Then it runs the second function on this script, the *compute_average_precision_per_class(),* which is the function that computes the average precision of object detection. The input arguments include:

- *'num_true_cases'*: a dictionary with class index as key and number of true positive cases for that class as value.
- *'gt_boxes'*: a dictionary with class index as key and for each class, a dictionary of image IDs and their corresponding ground truth bounding boxes.
- *'difficult_cases'*: a dictionary with class index as key and for each class, a dictionary of image IDs and the corresponding ground truth "difficulty" labels.
- *'prediction_file'*: a file containing the model predictions, with each line having the format '*image_id*\t*score*\t*box_coordinates'*.
- *'iou_threshold'*: the Intersection over Union (IoU) threshold used to determine a true positive match between a prediction and ground truth.
- *'use_2007_metric'*: a flag indicating whether to use the 11-point interpolation method (VOC 2007) or not.

The function first reads the prediction file and stores the image IDs, boxes, and scores in separate lists, sorted by score in descending order. Then, for each prediction, it finds the corresponding ground truth box (if there is one) and computes its IoU with the prediction. If the IoU is above the threshold, the prediction is considered a true positive, unless the ground truth is marked as "difficult". The true and false positive counts are accumulated as the loop progresses and are used to compute precision and recall. Finally, the average precision is computed using either the VOC 2007 method (if '*use_2007_metric'* is set) or the standard method.

```
def compute_average_precision_per_class(num_true_cases,gt_boxes,
                                        difficult_cases, prediction_file,
                                        iou_threshold, use_2007_metric):
    with open(prediction_file) as f:
        image_ids = []
        boxes = []
        scores = []
        for line in f:
            t = line.rstrip().split("\t")
            image_ids.append(t[0])
            scores.append(float(t[1]))
            box = torch.tensor([float(v) for v in t[2:]]).unsqueeze(0)
            box -= 1.0
            boxes.append(box)
        scores = np.array(scores)
        sorted_indexes = np.argsort(-scores)
        boxes = [boxes[i] for i in sorted_indexes]
        image_ids = [image_ids[i] for i in sorted_indexes]
        true_positive = np.zeros(len(image_ids))
```

```
        false_positive = np.zeros(len(image_ids))
        matched = set()
        for i, image_id in enumerate(image_ids):
            box = boxes[i]
            if image_id not in gt_boxes:
                false_positive[i] = 1
                continue

            gt_box = gt_boxes[image_id]
            ious = box_utils.iou_of(box, gt_box)
            max_iou = torch.max(ious).item()
            max_arg = torch.argmax(ious).item()
            if max_iou > iou_threshold:
                if difficult_cases[image_id][max_arg] == 0:
                    if (image_id, max_arg) not in matched:
                        true_positive[i] = 1
                        matched.add((image_id, max_arg))
                    else:
                        false_positive[i] = 1
            else:
                false_positive[i] = 1

    true_positive = true_positive.cumsum()
    false_positive = false_positive.cumsum()
    precision = true_positive / (true_positive + false_positive)
    recall = true_positive / num_true_cases
    if use_2007_metric:
        return measurements.compute_voc2007_average_precision(precision, re
call)
        # 11 point interpolation for average precision
    else:
        return measurements.compute_average_precision(precision, recall)
```

The *argsparse* input hyperparameters and settings for this script.

Parameters	Description
--net	The network architecture, it should be of mb1-ssd, mb1-ssd-lite, mb2-ssd-lite or vgg16-ssd
--dataset	The root directory of the VOC dataset or Open Images dataset
--trained_model	The trained model file path
--label_file	The label file path
--eval_dir	The directory to store evaluation results
--cuda-card	Define the GPU card to be used during the training

Table 3 – *eval_ssd.py* arguments

If the precision across all classes is over 70%, the model is ready to be exported, tested, and sent to the team that will test the inference in the store's CCTV live feed with the Jetson Nano.

To export the file there is the *onnx_export.py* script. This script converts the trained PyTorch object detection model to ONNX format. It uses the '*torch.onnx*' module for the conversion and '*argparse*' for parsing the command-line arguments of network architecture, input and output paths, labels file, input dimensions, and batch size. The script sets the device to use CUDA if available and specified by the user, otherwise it uses the CPU.

The script supports multiple network architectures, including VGG16, MobileNet v1, MobileNet v1 Lite, SqueezeNet Lite, and MobileNet v2 Lite.

It formats the input model paths and automatically selects the checkpoint with the lowest loss if an input path is not specified, determines the number of classes by counting the lines in the labels file, constructs the network architecture based on the specified architecture argument, then loads the PyTorch model checkpoint, converts it to ONNX format, and saves it to the specified output path.

Parameters	Description
--net	The network architecture, it can be mb1-ssd (aka ssd-mobilenet), mb1-lite-ssd, mb2-ssd-lite or vgg16-ssd.
--input	Path to input PyTorch model (.pth checkpoint)
--output	Desired path of converted ONNX model (default: <NET>.onnx)
--labels	Name of the class labels file
--width	Input width of the model to be exported (in pixels)
--height	Input height of the model to be exported (in pixels)
--batch-size	Batch size of the model to be exported (default=1)
--model-dir	Directory to look for the input PyTorch model in, and export the converted ONNX model to
--cuda-card	Define the GPU card to be used during the training

Table 4 – *onnx_export.py* arguments

The last script of this model train workflow is the *run_ssd_example.py,* which is the inference test of the object detection model. The script loads the input images and the label file and creates a list of image paths. After that, it chooses the network architecture based on the provided "*--net*" argument and creates the corresponding SSD network, then it loads the trained model weights and runs object detection on each image in the '*images_paths'* list. The output of the detection is the images with the bounding boxes around the detected objects, which are saved in the specified '*output_images_dir*'.

Parameters	Description
--net	The network architecture, it should be of mb1-ssd, mb1-ssd-lite, mb2-ssd-lite or vgg16-ssd
--trained_model	The trained model file path
--label_file	The label file path
--images_dir	Images for detection
--output_images_dir	Output of the detected images
--cuda-card	Define the GPU card to be used during the training

Table 5 - *run_ssd_example.py* arguments

4.1.3. Model train step guide

The whole model train workflow with PyTorch can be resumed to a few steps. Before running the scripts and assuming that there already is a VOC PASCAL annotations dataset, we start by accessing the GPU machine (Azure virtual machine with 4 GPUs) with MobaXterm.

```
ssh everisai@**.***.***.**
password: *********
```

Next we activate the '*opencv*' environment with a *conda* command. It allows to use a specific version of the library in isolation from the rest of our system. After executing this command, we should be able to run the scripts that use OpenCV.

```
conda activate opencv
```

Then we run a command to start a PyTorch Docker container using the Nvidia GPU acceleration and map the host directory, where we have stored the annotations dataset and the scripts, to the container directory.

```
sudo nvidia-docker run --ipc=host --gpus all -it --rm -v /home/everisai/pyt
hon/:/python nvcr.io/nvidia/pytorch:21.05-py3
```

The '*--ipc=host*' flag is used to share the host inter-process communication namespace with the container, allowing it to access the host's GPU devices, the '*--gpus all*' flag specifies that all available GPUs should be used, the '*-it*' starts the container in interactive mode and the '*—rm*' automatically removes the container when it exits. The '*nvcr.io/nvidia/pytorch:21.05-py3*' is the image name for the PyTorch environment with the CUDA version.

The next step is to install the libraries that we need and are missing from the container.

```
pip install torch==1.7.0+cu110 torchvision==0.8.1+cu110 torchaudio===0.7.0
-f https://download.pytorch.org/whl/torch_stable.html
```

The command will install (outdated versions used by the time of this project) PyTorch version 1.9.0, torchvision 0.8.1 and torchaudio 0.7.0. The '*+cu110*' indicates that these packages are built with CUDA 11.0 support, which requires an Nvidia GPU. The '*-f*' option allows you to specify the official PyTorch repository for stable releases URL from which to download the package.

Then we set the directory to the scripts folder and start to run them.

```
cd /python/training/detection/model_v1.0
```

Inside the container we start by running the *train_ssd.py* script. The *split_traintest.py* is ran in our local machines and then the separated data is imported into the container.

```
python3 train_ssd.py --net=vgg16-ssd --dataset-type=voc --pretrained-ssd=mo
dels/vgg16-ssd-mp-0_7726.pth --data=data/ --model-dir=models/vgg16/ --batch
-size=30 --workers 1 --learning-rate=0.001 --base-net-lr=0.001 --epochs 100
```

The command runs the training script for the object detection model using the architecture with the VGG16 backbone. The model is set to use a batch size of 30, a learning rate of 0.001, and is trained for 100 epochs.

I specified the use of a single worker for training with the '*--workers 1*'. This argument is used to specify the number of worker threads to use for data loading, which can help speed up the training process by loading data in parallel with the training process.

The next step is the evaluation of the object detection model with the *eval_ssd.py* script. The evaluation script will use the specified SSD architecture (vgg16-ssd), the trained model and the dataset to evaluate the model's precision across all classes.

```
python3 eval_ssd.py --net vgg16-ssd --dataset data/ --trained_model models/vgg16/vgg16-ssd-Epoch-93-Loss-2.5332502018321645.pth --label_file models/ labels.txt
```

If the model has an acceptable precision, it is exported with the *onnx_export.py* script.

```
python3 onnx_export.py --model-dir=models/vgg16
```

Then we run the trial implementation with the *run_ssd_example.py* script.

```
python3 run_ssd_example_WY.py vgg16-ssd models/vgg16/vgg16-ssd-Epoch-93-Loss-2.5332502018321645.pth models/vgg16/labels.txt data/JPEGImages
```

If there is confirmation of correct detections in the test images, the model moves to the inference process.

4.1.4. Limitations

This solution works and has positive results, but it also has limitations that can become liabilities for this project. The main problem here was that the time it takes an immense amount of time to fully run the *train_ssd.py* script. Each train attempt with a dataset of approximately 30 thousand annotations took between 15 and 16 hours.

```
2021-09-30 14:43:06 - Using CUDA...
2021-09-30 14:43:06 - Namespace(balance_data=Fals
2021-09-30 14:43:06 - Prepare training datasets.


2021-10-01 05:22:08 - Epoch: 99, Validation Loss: 
2021-10-01 05:22:08 - Saved model models/threat_de
2021-10-01 05:22:08 - Task done, exiting program.
```

Figure 17 – *train_ssd.py* running log

That time was not ideal since each hour of VM usage has its associated cost, and those costs were over the budget, which raised a financial liability. With this code, we were only able to train our model with one CUDA card at a time, but since our VM machine has four available GPU cards there is a possibility of using multiple cards to run the same script.

My challenge here is to change the workflow and enable the multi-GPU model training. That will be accomplished with the NVIDIA Transfer Learning Toolkit since it is compatible with our hardware, has a lot of architectural frameworks flexibility, and has a multi-GPU selection feature which in theory will accelerate the whole process.

4.2. TRANSFER LEARNING TOOLKIT DEMO

The first step to implement the NVIDIA Transfer Learning Toolkit is to install and configure an NGC Catalog account (NVIDIA GPU Cloud). This catalog is a curated set of GPU-optimized software for AI, HPC and Visualization.

NGC offers a diverse collection of containers, including deep learning frameworks, that bundle software applications, libraries, dependencies, and run-time compilers in a self-contained environment for seamless deployment across different computing environments, facilitating software portability and enabling easy scaling of applications across the cloud, data center, and edge environments with a single command [16].

Then I had to create an NGC account in *https://catalog.ngc.nvidia.com/* and get an API $KEY that it can be generated in the account options.

Having the previous step concluded, I am able to fetch and start any NGC container. In the TLT collection, there's a detailed overview that goes through of what it is, how it works, where to get started. After studying the TLT available options, I settled for the **Transfer Learning Toolkits For Streaming Video Analytics** container.

The next step was to fetch the container from NGC onto the GPU instance, and that is done by running a provided docker command on the VM terminal.

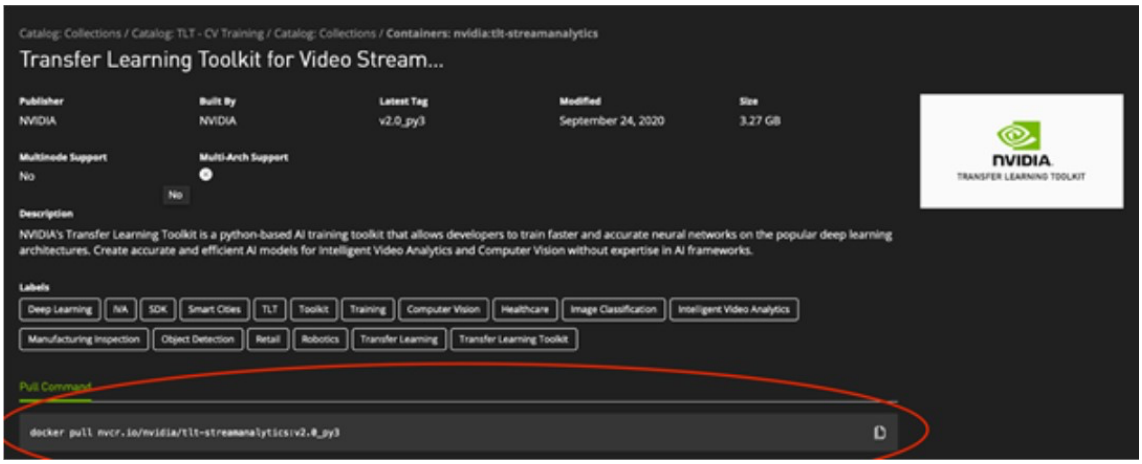

Figure 18 – TLT page on the NGC catalog for video streaming [16]

The docker command for pulling the container.

```
$ docker pull nvcr.io/nvidia/tlt-streamanalytics:v2.0_py3
```

Concluding the container download, we need to run the command to start the same container in an interactive terminal session. Http requests are forwarded from the port 80:8888 for the Jupyter notebook, so a local directory is mounted on the GPU instance with access to the dataset for the transfer learning with the following command:

```
$ docker run --gpus all -it -v "/path/to/dir/on/host":"/path/to/dir/in/dock
er" \

-p 80:8888 nvcr.io/nvidia/tlt-streamanalytics:v2.0_py3 /bin/bash
```

The command starts by running the Docker container using the *docker run* command, followed by some flags and options:

- The *--gpus all* flag is used to enable GPU support for the container.
- The *-it* flag is used to run the container in interactive mode with a TTY.
- The *-v* flag is used to mount a host directory to a directory inside the container, in this case, the *"/path/to/dir/on/host"* on the host is being mounted to the *"/path/to/dir/in/docker"* directory inside the container.
- The *-p* flag is used to publish a container's port to the host.
- The *"nvcr.io/nvidia/tlt-streamanalytics:v2.0_py3"* is the image name from the NGC registry.
- The "*/bin/bash*" is the command that is run when the container starts.

This command will start a new container from the specified image with the specified options. The *-v* flag allows to access files from the host machine in the container and the *-p* flag allows to access the web-based interface of the TLT inside the container by going to localhost:80 in the host machine.

The data type used with the TLT features has to be in KITTI format and since the data that I had was in Pascal VOC, I downloaded a suggestion of data samples from Kaggle [17] and ran a face mask detection demo [18] provided by NIVIDIA.

The downloaded datasets must be in a specific data tree for the example to work:

Figure 19 – Data tree for NVIDIA demo [18]

This dataset is also in XML format and in the git repository there is a script to convert this data into KITTI format (data2kitti.py). The script is specifically conceived for this and other provided datasets, thus not working on my data. So, before preparing the dataset I have to run a command to clone the NVIDIA face mask detection repository into my docker image.

```
git clone https://github.com/NVIDIA-AI-IOT/face-mask-detection.git
```

After running the conversion script and the dataset in KITTI format, we need to create the TensorFlow records (tfrecords) which is a binary file format used to store data for TensorFLow. The format allows for storing large amounts of data in a single file, as well as for efficient reading and decoding of the data. Each tfrecord file contains a sequence of binary records, where each record is a binary string that represents a serialized example protobuf. The example protobuf is a data structure that contains the actual data, as well as metadata about the data such as its shape and data type. TFRecords can be used to store a variety of data types, including images, text, and numerical data [19].

To create the tfrecords I just need to access the specific file for a given architecture, which will be detectnet_v2.

```
tlt-dataset-convert -d $SPECS_DIR/detectnet_v2_tfrecords_kitti_trainval.txt
-o $DATA_DOWNLOAD_DIR/tfrecords/kitti_trainval/kitti_trainval
```

In this command calls the tlt-dataset-convert feature. It has two flags has input, the *-d* flag which is specifying the path to the dataset and the *-o* flag, which is specifying the path to where the converted dataset will be saved.

Having the tfrecords, it's necessary to download the pre-trained model from the NGC model registry. As mentioned, the selected architecture framework is DetectNet_v2 since the input is expected to be 0-1 normalized with input channels in RGB order, therefore for optimum results I just must select a model with the *_detectnet_v2 tag in the name.

Available I have the following supported networks:

- Resnet10/ Resnet18/ Resnet34/resnet50
- Vgg16/vgg19
- Googlenet
- Mobilenet_v1/mobilenet_v2
- Squeezenet
- Darknet19/darknet53

Since Resnet18 is the best network classification, I can test it in this demo and evaluate the results myself, since the face mask is one of the necessary detections of our solution.

Before running the download command I need to create a destination folder with the *mkdir -p* bash command.

```
mkdir -p pretrained_resnet18/
```

And then download the selected model from NGC.

```
ngc registry model download-version nvidia/tlt_pretrained_detectnet_v2:resn
et18 \
    --dest $USER_EXPERIMENT_DIR/pretrained_resnet18
```

This command is composed by the model identifier and the *–dest* flag to specify the path to where the pre-trained model will be saved.

Having the model, I just need to customize the training specifications in the *detectnet_v2_train_resnet18_kitti.txt* file. The specifications file is already set to work with the test data and train a model for 'mask' and 'no-mask' detection, but although there was no need for changes to properly use it, I studied it and listed for future reference some potential parameter changes.

The data sources section, where the data and tfrecords directory paths have to be specified.

```
data_sources {
    tfrecords_path: "/home/data/tfrecords/kitti_trainval/*"
    image_directory_path: "/home/data/train"
  }
```

The target classes, where the change is necessary to detect the classes that we want.

```
target_class_mapping {
    key: "balaclava"
    value: "balaclava"
  }
  target_class_mapping {
    key: "knife"
    value: "knife"
  }
```

The preprocessing section where we can customize the output image width and height.

```
preprocessing {
    output_image_width: 960
    output_image_height: 544
    min_bbox_width: 1.0
    min_bbox_height: 1.0
    output_image_channel: 3
  }
```

The classes in the postprocessing configuration section.

```
postprocessing_config {
  target_class_config {
    key: "balaclava "
    value {
      clustering_config {
        coverage_threshold: 0.00499999988824
        dbscan_eps: 0.20000000298
        dbscan_min_samples: 0.0500000007451
        minimum_bounding_box_height: 20
      }
    }
  }
  target_class_config {
    key: "knife"
    value {
      clustering_config {
        coverage_threshold: 0.00499999988824
        dbscan_eps: 0.15000000596
        dbscan_min_samples: 0.0500000007451
        minimum_bounding_box_height: 20
      }
```

The model configurations section where there is the actual model file selection and the architecture specification.

```
model_config {
  pretrained_model_file:  "/home/detectnet_v2/pretrained_resnet18/tlt_pretra
ined_detectnet_v2_vresnet18/resnet18.hdf5"
  num_layers: 18
  use_batch_norm: true
  objective_set {
    bbox {
      scale: 35.0
      offset: 0.5
    }
    cov {
    }
  }
  training_precision {
    backend_floatx: FLOAT32
  }
  arch: "resnet"
}
```

The classes in the evaluation configurations and the validation epoch.

```
evaluation_config {
  validation_period_during_training: 10
  first_validation_epoch: 10
  minimum_detection_ground_truth_overlap {
    key: "balaclava "
    value: 0.5
  }
  minimum_detection_ground_truth_overlap {
    key: " knife "
    value: 0.5
  }
  evaluation_box_config {
    key: " balaclava "
    value {
      minimum_height: 20
      maximum_height: 9999
      minimum_width: 10
      maximum_width: 9999
    }
  }
  evaluation_box_config {
    key: " knife "
    value {
      minimum_height: 20
      maximum_height: 9999
      minimum_width: 10
      maximum_width: 9999
    }
  }
  average_precision_mode: INTEGRATE
}
```

The classes in the cost function configurations section.

```
cost_function_config {
  target_classes {
    name: " balaclava "
    class_weight: 1.0
    coverage_foreground_weight: 0.0500000007451
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
      weight_target: 10.0
    }
  }
  target_classes {
    name: " knife "
    class_weight: 8.0
    coverage_foreground_weight: 0.0500000007451
    objectives {
      name: "cov"
      initial_weight: 1.0
      weight_target: 1.0
    }
    objectives {
      name: "bbox"
      initial_weight: 10.0
      weight_target: 1.0
  }
```

The training configurations, where we can adjust some important parameters, such has the batch size per gpu, number of epochs and the learning rate.

```
training_config {
  batch_size_per_gpu: 24
  num_epochs: 120
  learning_rate {
    soft_start_annealing_schedule {
      min_learning_rate: 5e-06
      max_learning_rate: 5e-04
      soft_start: 0.10000000149
      annealing: 0.699999988079
    }
  }
```

And the last necessary change I identified were the classes in the bounding box configurations.

```
bbox_rasterizer_config {
  target_class_config {
    key: " balaclava "
    value {
      cov_center_x: 0.5
      cov_center_y: 0.5
      cov_radius_x: 0.40000000596
      cov_radius_y: 0.40000000596
      bbox_min_radius: 1.0
    }
  }
  target_class_config {
    key: " knife "
    value {
      cov_center_x: 0.5
      cov_center_y: 0.5
      cov_radius_x: 1.0
      cov_radius_y: 1.0
      bbox_min_radius: 1.0
    }
  }
```

Having these changes listed and understood, everything was ready to run the *tlt-train* feature.

```
tlt-train detectnet_v2 -e $SPECS_DIR/detectnet_v2_train_resnet18_kitti.txt \
                       -r model_unpruned \
                       -k $KEY \
                       -n resnet18_detector \
                       --gpus $NUM_GPUS
```

After the train is completed, I can evaluate the model's accuracy on each class.

```
tlt-evaluate detectnet_v2 -e $SPECS_DIR/detectnet_v2_train_resnet18_kitti.t
xt\
                          -m model_unpruned/weights/resnet18_detector.tlt \
                          -k $KEY
```

The *tlt-evaluate detectnet_v2* command is used to evaluate a trained model using the detectnet_v2 architecture. The *-e* flag is specifying the path to the evaluation configuration file, is the same used for the training process, the *-m* flag is specifying the path to the trained model, and the *-k* flag is the access key which has to be the same used for the train process.

As of this step, I have a trained model and its detection accuracy. It already can be exported for testing purposes, but before that, there is the optional step of prune the model. Model pruning is the process of removing unnecessary parameters from a trained model in order to reduce its size and computational requirements. Pruning a model with TLT is done using the prune command, but first I must create a new directory to store the pruned model:

```
mkdir -p model_pruned
```

And now the TLT model prune command:

```
tlt-prune -m $USER_EXPERIMENT_DIR/model_unpruned/weights/resnet18_detector.
tlt \
           -o $USER_EXPERIMENT_DIR/model_pruned/resnet18_nopool_bn_detectne
t_v2_pruned.tlt \
           -eq union \
           -pth 0.8 \
           -k $KEY
```

Here, the *-eq* flag is specifying the pruning method: in this case, it is 'union', a method that removes the weights with the lowest absolute. The *-pth* flag is specifying the pruning threshold: in this case the suggestion is to use 0.8, which means that only the weights that are below 0.8 will be pruned but later I can adjust for accuracy and model size trade off. A higher *pth* gives higher inference speed but worse accuracy.

Now, the pruned model must be retrained.

```
tlt-train detectnet_v2 -e $SPECS_DIR/detectnet_v2_retrain_resnet18_kitti.tx
t \
                       -r $USER_EXPERIMENT_DIR/model_retrain \
                       -k $KEY \
                       -n resnet18_detector_pruned \
                       --gpus $NUM_GPUS
```

This command is like the previous one, but to run the retrain there's the need to specify a different specifications file, which is very similar to the train file. The *--gpus* and *-k* input must be the same used in the train process.

When the retrain process is finished, I can re-evaluate the model.

```
tlt-evaluate detectnet_v2 e $SPECS_DIR/detectnet_v2_retrain_resnet18_kitti.
txt \
                           -m $USER_EXPERIMENT_DIR/model_retrain/weights/re
snet18_detector_pruned.tlt \
                           -k $KEY
```

After concluding these steps, I have a *.tlt* file, and to test if it properly detects the intended classes I can run the TLT inference feature in a data sample with at least 8 images and see if there is any detection.

```
tlt-infer detectnet_v2 -e $SPECS_DIR/detectnet_v2_inference_kitti_tlt.txt \
                       -o $USER_EXPERIMENT_DIR/tlt_infer_testing \
                       -i          $DATA_DOWNLOAD_DIR/test_images          \
                       -k $KEY
```

This command will perform inference on a trained model using the *.tlt* file on the input images located in the directory specified by the *-i* flag and save the inference output to the directory specified by the *-o* flag. Same as before, the specifications file is already set to work with this data and these classes, but looking at it I could identify that besides the class names I also will have to change the inference dimensions to match what the model was trained for.

```
# Inference dimensions.
  image_width: 960
  image_height: 544
```

Looking into the tlt_infer_testing files, I confirmed that the model was detecting both classes properly.

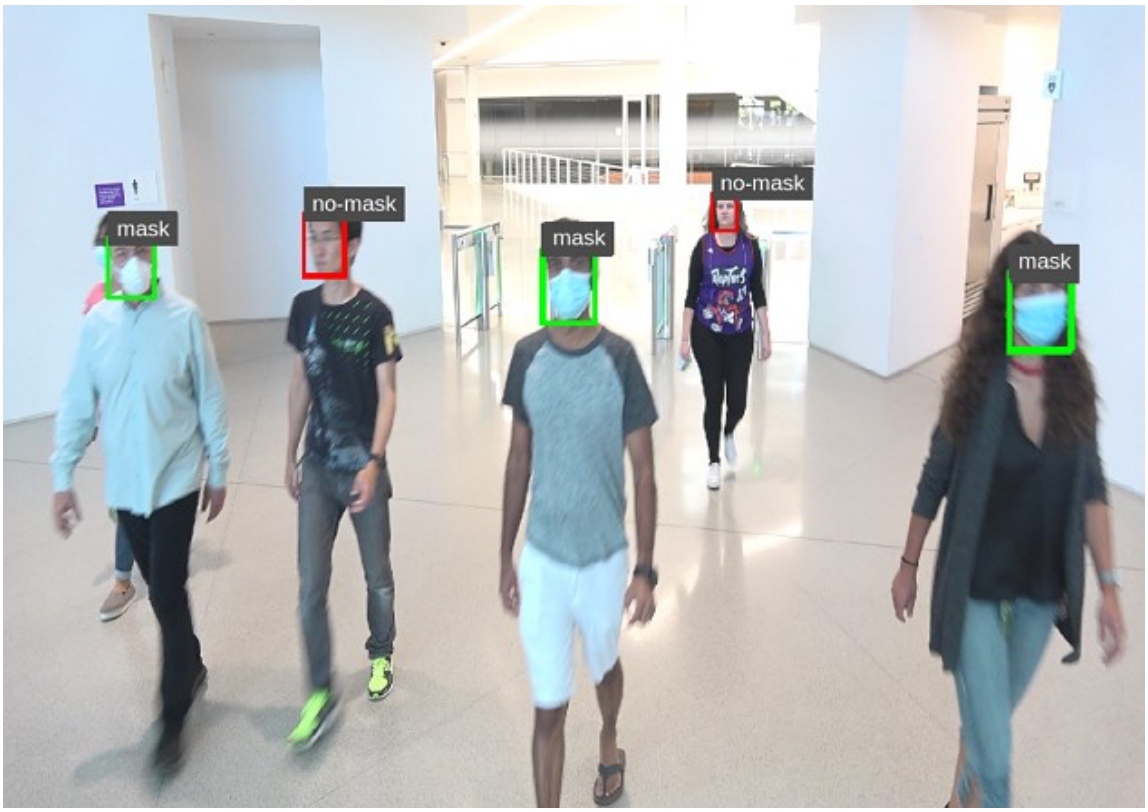

Figure 20 – TLT inference with mask/no mask demo model [18]

Finally the next and last step is to export the file to .etlt format. In summary, the .tlt format is used to store the model's architecture, weights, and other metadata and it can be used for both training and inference while the .etlt format is an optimized format for deployment and inference and it can be used only for inference.

```
tlt-export detectnet_v2 \
-m experiment_dir_retrain/weights/resnet18_detector_pruned.tlt \
-o experiment_dir_final_pruned/resnet18_detector_pruned.etlt \
-k $KEY
```

Now the model is ready for deployment, and I can move on to the next phase and apply these TLT features with our data and detect a different set of classes.

4.3. DATA PREPARATION

4.3.1. KITTI Conversion

This is the final step to fully integrate our data with the TLT features. The TLT train process only works if the input data is in KITTI format, which is not possible to export directly from CVAT, so I need to export the data in VOC PASCAL which gives me the annotations files in .xml, and then convert to KITTI with a script.

To fulfill this task, I cloned a github repository (https://github.com/krustnic/xml2kitti) with that same purpose and created some extra features to customize the code according to our necessities.

Another TLT train specification is that all images in the dataset have to be the same size, so if in the configuration file I set the width to 960 and the height to 544, all images must have that exact same dimensions.

```
path = r'…/JPEGImages'
c=0
for file in os.listdir(path):
    f_img = f+"/"+file
    img = Image.open(f_img)
    img = img.resize((height,width))
    img.save(f_img)
    c+=1
print(f'{c} resized images')
```

This code uses the Python imaging library (PIL) to resize the set of images located in a directory. The first line assigns the path of the directory containing the images to the variable *'path'* then it initializes a variable *'c'* to zero, which will be used to keep track of the number of resized images.

In the *for* loop, the '*os.listdir(path)*' function is used to list all the files in the directory specified by the '*path'* variable. The loop iterates through each file in the directory. For each iteration, the code opens the file using the PIL's '*Image.open()*' function and assigns it to the variable *'img'*. Then it uses the '*.resize()*' method on the *'img'* variable to resize the image to the given *'height'* and *'width'* size. Then it saves the resized image to the same file using the '*.save()*' method.

Finally, the '*c*' variable is incremented by 1 for each iteration of the loop, to keep track of the number of resized images. At the end of the loop, the code prints the total number of resized images.

The *xml2kitti.py* successfully converted most of the files. The output was sent into the same folder of the input and there were some conversion exceptions, raising the issue of having more images than KITTI annotations, which must be the same number. To solve this problem first I added a file moving

function to move the output to another directory ('.../Annotations_kitti') and a file remover function to remove the images without a corresponding KITTI annotation.

```
dest_dir = '.../Annotations_kitti'
for file in glob.glob('.../Annotations/*.txt'):
    shutil.move(file, dest_dir)
```

In this code snippet I use the python *'glob'* and *'shutil'* libraries to move a set of files from one directory to another. The first line assigns the path of the destination directory to the variable '*dest_dir'.* The *for* loop uses the *'glob'* library function *'glob.glob()*' to search for all the files in the directory '.../Annotations/' that have the '.txt' extension and then it iterates through each file found. For each file, the code uses the *'shutil'* library function '*shutil.move()*' to move the file from its current location to the destination directory specified by the *'dest_dir'* variable.

```
kitti_list=[]
c=0
for file2 in os.listdir(".../Annotations_kitti"):
    name2 = file2.rsplit('.',1)[0]
    kitti_list.append(name2)

for file in os.listdir(".../JPEGImages"):
    name1 = file.rsplit('.',1)[0]
    filename = ".../JPEGImages/"+file
    if name1 not in kitti_list:
        os.remove(filename)
        c+=1
        print(f'Removed-------{file}')
print(f'Removed {c} files')
```

Here I use python's *os* library to remove a set of files from a directory based on a comparison with the files in another directory. The variable *'c'* will be used to keep track of the number of removed files, and the *'kitti_list'* will have the names of all the KITTI files, which are added in the first loop using the '*os.listdir()'* function to list all the files in *".../Annotations_kitti"* and iterates through each file in the directory. For each file, I apply the '*.rsplit()*' function to remove the file extension and append the name to the variable *'kitti_list'*. The second *for* loop uses the '*os.listdir()*' function to list all the files in the image directory ("*.../JPEGImages*") and iterates through each file in the directory. For each file, I again apply the '*.rsplit()*' function to remove the file extension and assign the name of the file to the 'name1' variable to then compare it to the names in the '*kitti_list',* if is not found I apply the '*os.remove()*' to delete the image file and increment the '*c*' variable by 1.

By the end, to apply these changes I just need to run the following in the prompt.

```
python3 xml2kitti.py .../Annotations
```

4.3.2. Annotations and Folder Structure

The first step of this segment is the labeling process in CVAT. We start by running the docker CVAT image and access through the http://localhost:8080/ on google chrome. Next I have to create a new task, name it, add the labels that will be the classes in the exported annotations, upload the images to be labeled and manually select in the image the objects associated to the labels.

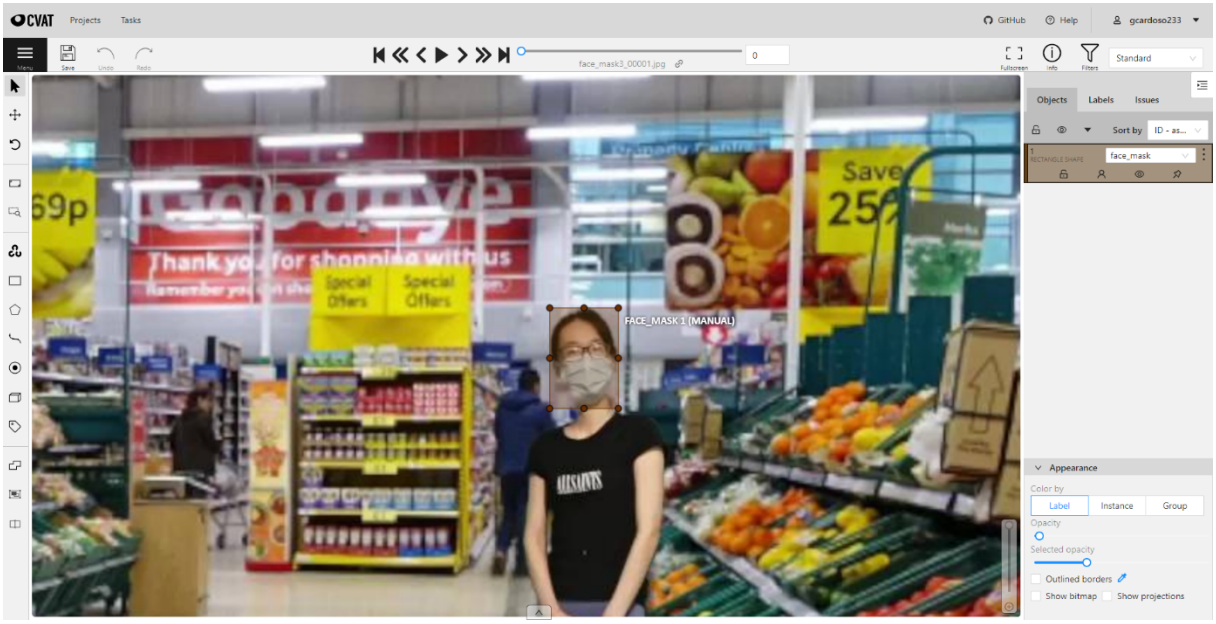

Figure 21 – CVAT Annotation example

After finishing the label process in all images there is an export option to dump all the annotations in PASCAL VOC format into a folder. That folder will be the one used to convert the files to KITTI and import the data for the model training, but there are some necessary changes to have the folder in the right structure.

The dumped folder is composed by the following:

- Labels '.txt' file
- 'Annotations' folder
- 'ImageSets' folder

For the desired structure I must do these changes:

- Insert the images folder with the name 'JPEGImages'.
- Rename the labels file to 'labels'.
- In the labels file, delete everything except the classes names and I also need to make sure that there are not any empty lines.
- Delete the 'ImageSets' folder.
- Create an empty folder named 'Annotations_kitti'.

By the end of these changes the desired folder structure should look like this.

```
|---Annotations
|       image_name.xml
|       ..........
|       ..........
|
|---Annotations_kitti
|       (empty)
|
|---JPEGImages
|       image_name.jpeg
|       ..........
|       ..........
|
|---labels.txt
```

4.4. TRAINING

With the annotations in KITTI format I finally can use the TLT features. The first data batch is composed of close to 30000 annotations of two classes, balaclava, and knife. The main goal here is to apply the train feature and guarantee a lower training time with the multiple gpus option and still maintain an acceptable precision (>70%).

Before the whole process, download the tlt_specs folder from the NVIDIA demo to my local machine and adjust the configuration files as listed above.

The docker image directory name will be "train_tlt", so the data sources change must be according to that, set the target classes names to “balaclava” and “knife”, select the output image width and height to 1280 and 720 respectively (the resolution of the input images) and to reduce the train time I can set number of training epochs to 100.

Having done these changes on all files I can start the process steps according to the NVIDIA demo.

First, pull the docker TLT container.

```
docker pull nvcr.io/nvidia/tlt-streamanalytics:v2.0_py3
```

Run the docker image with the name “train_tlt”.

```
docker run --gpus all -it -v "/path/to/dir/on/host/":"train_tlt" \
-p 8888:8888 nvcr.io/nvidia/tlt-streamanalytics:v2.0_py3 /bin/bash
```

The path on host we should apply is the one where the TLT specifications files and the annotations are. For this first run I am going to select the same architecture framework used in the demo, the Detectnet_v2 and the pretrained model will be the Resnet18. Later, when the process is fully working, I can test another frameworks and models and tune it to optimize the detection.

With the docker image created and the necessary data there, I must convert the tf records from the KITTI annotations.

```
tlt-dataset-convert -d tlt_specs/detectnet_v2_tfrecords_kitti_trainval.txt -
o /train_tlt/tfrecords/kitti_trainval/
```

After confirming that the records are in the output directory, I need to create a target destination folder and download the pretrained model.

```
mkdir pretrained_resnet18

ngc registry model download-version nvidia/tlt_pretrained_detectnet_v2:resn
et18 \
--dest pretrained_resnet18
```

Next, open the *detectnet_v2_train_resnet18_kitti.txt* to confirm that everything is correctly set to run the TLT train feature. After guarantee that the specifications are correct, I can train the model with my data. First, I am going to use 2 gpus and compare the total training time with the performance I got using just 1. The previous train time was around 15h-16h.

```
tlt-train detectnet_v2 -e tlt_specs/detectnet_v2_train_resnet18_kitti.txt \
-r model_unpruned \
-k $KEY \
-n resnet18_detector \
--gpus 2
```

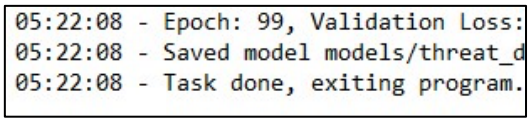

Figure 22 – TLT running log

As observed in the figure above, the time it took to complete the model training was 5 hours and 22 minutes, which reduces by 10 hours the model train time and fulfills the main goal of this implementation. Now I will directly prune and retrain the model and after that evaluate and try to test the inference on some test images.

I will create a different directory for the pruned model and then run the *tlt-prune* feature.

```
mkdir model_pruned

tlt-prune -m model_unpruned/weights/resnet18_detector.tlt \
          -o model_pruned/resnet18_nopool_bn_detectnet_v2_pruned.tlt \
          -eq union \
          -pth 0.01 \
          -k $KEY
```

After the model prune the model must be retrained.

```
tlt-train detectnet_v2 -e tlt_specs/detectnet_v2_retrain_resnet18_kitti.txt
\
-r model_retrain \
-k $KEY \
-n resnet18_detector_pruned \
--gpus 2
```

Again, the time of retrain is around the 5 hours mark. Now I run the evaluation and check the model's detection quality.

```
tlt-evaluate detectnet_v2 -e tlt_specs/detectnet_v2_retrain_resnet18_kitti.
txt \
-m model_retrain/weights/resnet18_detector_pruned.tlt \
-k $KEY
```

```
Average Precision Per-class:
Balaclava: 0.9033411948976582
knife: 0.661061663112135

Average Precision Across All Classes: 0.7822014290048966
```

Figure 23 – Trained model 2 class precision

The results are very promising, we can observe that the balaclava classes detection precision is very good and well above the 70%, which is not the case for the knife detection which is quite understandable since a knife is much more difficult to detect.

Although the knife precision is not ideal, is very close to the minimum necessary and the precision across all classes meets the requirements, so I can proceed to the inference test phase and depending on the results I decide if this model will be exported and tested *in loco* or if it needs to be improved immediately.

With this thought, I will run the inference command to generate detections on some random images that I have put together and imported to the docker image.

```
tlt-infer detectnet_v2 -e train_tlt/detectnet_v2_inference_kitti_tlt.txt \
                       -o tlt_infer_testing \
                       -i test_images \
                       -k $KEY
```

Looking to the output, I confirm that the results are satisfactory and in fact detection is running accordingly the expectations.

Figure 24 – Tlt inference test

Has observed the values associated to the detections are according to the *tlt-evaluation* results. Since these results demonstrate that the model can in fact detect I will export and share with the local team to test it in the client's premises. To do so, I apply the export command.

```
mkdir -p model_final_pruned

tlt-export detectnet_v2 \
-m model_retrain/weights/resnet18_detector_pruned.tlt \
-o model_final_pruned/resnet18_detector_pruned.etlt \
-k $KEY
```

The *.etlt* file is the final product of the whole process and is what I will share with the inference team.

5. RESULTS DISCUSSION

By the end of my internship, the business objectives were not yet fulfilled. My implementation was available in Q4 and there was the deployment and testing in 3 different stores to be done and approved before the next step.

The Data Mining goals of the model also were not yet fulfilled due to the knife's precision of 66%, below the established 70% minimum. This will be solved in the next steps and future work that will come with this implementation.

The *in-loco* test of the first balaclava/ knife detection model had very positive feedback. This was the process to use on the solution due to the speed detection *.etlt* provided and the significant training time reduction. The detection quality was not the desired, but the TLT implementation brought much more flexibility to changes, and parameters tuning and with the train time reduction, the resources necessary for new model tests and developments also decreased.

Before this implementation, the training process was rigid due to the code used. The architecture framework was much harder to change due to the code being set to work exclusively with SSD. The TLT process not only works with ssd architecture but is also compatible with other architectures which consequently guarantees access to numerous more pre-trained models.

These are the new training flow steps:

1. Get the annotations with CVAT
2. Convert dataset to KITTI format
3. Download the pretrained model
4. Train model with tlt-train
5. Prune trained model with tlt-prune
6. Retrain pruned model
7. Evaluate the re-trained model on validation data with tlt-evaluate
8. If the accuracy is not satisfactory get back to step 1 with more data or prune and retrain again, else move on to the next step
9. Export trained model with tlt-export

Overall, the process got faster, simpler and with a few minor tweaks it will outperform the previous implementation.

For future work, besides monitoring the model's inference performance with the CCTV's live feed, there's also the quality issue since one of the classes had values bellow the objective, namely of the knife. There's a need to obtain more knife images and balance the dataset in favor of this class. The main cause of this performance disparity can be traced to the object itself. Knives are smaller and reflect light, which affects the detection as opposed to the balaclava class, a bigger and more obvious object that does not reflect light as much, thus impacting the detection performance. Also, to reach better overall results, we can adjust the configuration parameters, such as the learning rate, the training batch, or the pruning threshold.

Another task ahead is the inclusion of more classes, not only more threatening objects but also other face covers, such as the face mask or a burka, since both can be classified as non-threatening in a real-life context but the similarity with other face covers can lead our model to a detection mistake. This is a task for a more advanced phase but necessary for the successfull implementation of this solution in a store.

6. CONCLUSIONS

This internship at Everis was an amazing opportunity to have my first experience in the technological sector, and to have the chance to work on my first Computer Vision and AI project in a professional context.

My integration into this project was a challenge that came from the company, since they knew that the process they had was not ideal, and although it worked, it lacked efficiency. The time it took to train detection models was too high, consequently spending excessive money due to the Azure VM usage. Also, it was inflexible on what architecture framework and pre-trained models could be used.

To integrate this project, I had to be comfortable with the tools and the whole process so the first couple of months the tasks were mainly theoretical and research. After understanding the whole process, I focused a lot on the image labeling and training process (I labeled over 80000 images for training and testing purposes) to fulfill new data and the trained model's demand. That high demand for models to test at the client's stores, was what evidenced the resource and time ineffectiveness and what triggered the improvement necessity (multiple GPU training) and my role in the project.

I had the responsibility of doing the research and improving the solution. After some trials and several time-consuming tests, I found the NVIDIA Transfer Learning Toolkit (now called "TAO Toolkit"), which provides a powerful and efficient framework for implementing computer vision models. The use of pre-trained models and the CUDA-enabled GPU architecture allows for a quick and easy model fine-tuning for specific use cases with minimal data and computational resources. The ability to train and deploy models in different environments (including on-premises and in the cloud) makes TLT a versatile and valuable tool for computer vision implementation. This report demonstrated the potential of TLT through the implementation of object detection models for a retail store use case, where it could accurately detect and classify threatening objects using CCTV footage in real time. The results show that TLT can accelerate the development and deployment of computer vision models, making it an ideal solution for organizations looking to implement computer vision technology. Overall, this thesis highlights the potential of TLT as a powerful tool for computer vision implementation and its ability to improve security, operational efficiency, and the customer experience in a retail setting.

During this internship, there were many lessons learned. New software tools and new fields made this an amazing learning experience where i acquired and developed multiple skills.

Working in a multinational tech company, and particularly in an innovation department highlighted that curiosity, motivation, and know-how help us thrive and get innovative breakthroughs. There is the need for great people leading and a resourceful structure behind us, but also there is a need to work outside our comfort zone and test our limits. There was lots of new information I had to deal with, and every day was a challenge, but that made me understand that this is the right path, and if I keep the same urge and motivation to learn, I can be happy and enjoy working in such projects.

Artificial intelligence is here to stay, and my learning curve on the subject increased exponentially on subjects such as Pytorch or Tensorflow. Working and getting comfortable with lots of new tools like git and bash commands, Jira tickets, docker containers and images, cloud virtual machine, and a computer vision annotation tool was another highlight of this experience.

Looking back to those six months it became evident that every day was not only technically challenging but also a test for my soft skills, which I must point out that improved a lot by developing my teamwork, organizational, problem solving and communication skills. In conclusion, by the end of the internship I felt a personal improvement and growth and ready to take on my next challenge.

7. REFERENCES

[1] NTT Data, 2021. *URL:* https://pt.nttdata.com/about-us

[2] Josh Patterson & Adam Gibson, 2017. "Deep Learning A Practitioner's Approach", Chapter 3 *"Fundamentals of Deep Networks"*

[3] Ian Goodfellow, Yoshua Bengio, & Aaron Courville, 2015. "*Deep Learning*"

[4] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis & Eftychios Protopapadakis, 2018. *"Deep Learning for Computer Vision: A Brief Review"*

[5] Benjamin Planche & Eliot Andres, 2019. "Hands-On Computer Vision with TensorFlow 2", Chapter 1, "*Computer Vision and Neural Networks*"

[6] Benjamin Planche & Eliot Andres, 2019. "Hands-On Computer Vision with TensorFlow 2", Chapter 2, "*Tensorflow Basics and Training a Model*"

[7] Benjamin Planche & Eliot Andres, 2019. "Hands-On Computer Vision with TensorFlow 2", Chapter 3, "*Modern Neural Networks*"

[8] NVIDIA Corporation, 2022. *URL:* https://developer.nvidia.com/deepstream-sdk

[9] NVIDIA Corporation, 2022. *URL:* https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_Overview.html#nvidia-deepstream-overview

[10] NVIDIA Corporation, 2022. *URL:* https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[11] NVIDIA Corporation, 2022. *URL:* https://docs.nvidia.com/tao/archive/tlt-30/text/overview.html

[12] Python Software Foundation 2001-2002 *URL:* https://www.python.org/doc/essays/blurb/

[13] Mobatek 2008-2022. *URL:* https://mobaxterm.mobatek.net/

[14] Docker Inc. 2013-2022. *URL:* https://docs.docker.com/get-started/overview/

[15] scikit-learn developers (BSD License) 2007 – 2022. *URL:* https://scikit-learn.org/stable/modules/sgd.html

[16] NVIDIA Corporation, 2022.
URL: https://docs.nvidia.com/ngc/ngc-catalog-user-guide/index.html#what-is-ngc-catalog

[17] Evan Danilovich (2020 March). Medical Masks Dataset. Version 1. Retrieved May 14, 2020. *URL:* https://www.kaggle.com/datasets/ivandanilovich/medical-masks-dataset-images-tfrecords

[18] NVIDIA Corporation, 2022. *URL:* https://github.com/NVIDIA-AI-IOT/face-mask-detection

[19] Apache 2.0, 2022. *URL: https://www.tensorflow.org/tutorials/load_data/tfrecord*

[20] Shiming Ge, Jia Li, Qiting Ye, Zhao Luo, 2017. "*Detecting Masked Faces in the Wild With LLE-CNNs*", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

[21] Vidit Jain and Erik Learned-Miller, 2010. "*FDDB: A Benchmark for Face Detection in Unconstrained Settings*". Technical Report UM-CS-2010-009, Dept. of Computer Science, University of Massachusetts, Amherst.

[22] Yang, Shuo and Luo, Ping and Loy, Chen Change and Tang, Xiaoou, 2016. "*WIDER FACE: A Face Detection Benchmark*", IEEE Conference on Computer Vision and Pattern Recognition (CVPR).