LIU-ITN-TEK-A-019/049--SE

A Client-Server Solution for Detecting Guns in School Environment using Deep Learning Techniques

Johan Olsson

2019-09-27

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LIU-ITN-TEK-A-019/049--SE

A Client-Server Solution for Detecting Guns in School Environment using Deep Learning Techniques

Examensarbete utfört i Medieteknik
vid Tekniska högskolan vid
Linköpings universitet

Johan Olsson

Handledare Daniel Nyström
Examinator Sasan Gooran

Norrköping 2019-09-27

Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Medieteknik
202019 | LIU-IDA/LITH-EX-A--2019/001--SE

A Client-Server Solution for Detecting Guns in School Environment using Deep Learning Techniques

En klient-server-lösning för att detektera vapen i skolmiljöer med hjälp av maskininlärning

Johan Olsson

Supervisor : Daniel Nyström
Examiner : Sasan Gooran

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

With the progress of deep learning methods the last couple of years, object detection related tasks are improving rapidly. Using object detection for detecting guns in schools remove the need for human supervision and hopefully reduces police response time. This paper investigates how a gun detection system can be built by reading frames locally and using a server for detection. The detector is based on a pre-trained SSD model and through transfer learning is taught to recognize guns. The detector obtained an Average Precision of 51.1% and the server response time for a frame of size 1920 x 1080 was 480 ms, but could be scaled down to 240 x 135 to reach 210 ms, without affecting the accuracy. A non-gun class was implemented to reduce the number of false positives and on a set of 300 images containing 165 guns, the number of false positives dropped from 21 to 11.

Acknowledgments

The author would like to thank supervisor Daniel Nyström along with examiner Sasan Gooran. Also, the author would like to thank Ann-Sofie Rase, Andreas Wrangsjö, Andy Truong, Jonas Råberg and all the other employees at Axis Communications for always being available to answer questions and give feedback.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

School shootings are a massive problem in the United States. 2018 was by far the worst year up to date, with 110 incidents involving a gun being brandished or fired in a school property. This was an increase of 86% from the previous high in 2006. By September 2019, there have been 48 incidents, killing and injuring a total of 59 persons, see figure 1.1 and figure 1.2. The graphs and statistics in this section are taken from the *K-12 School Shooting Database* which is conducted as part of the Advanced Thinking in Homeland Security program at the Naval Postgraduate School's Center for Homeland Defense and Security. [1].

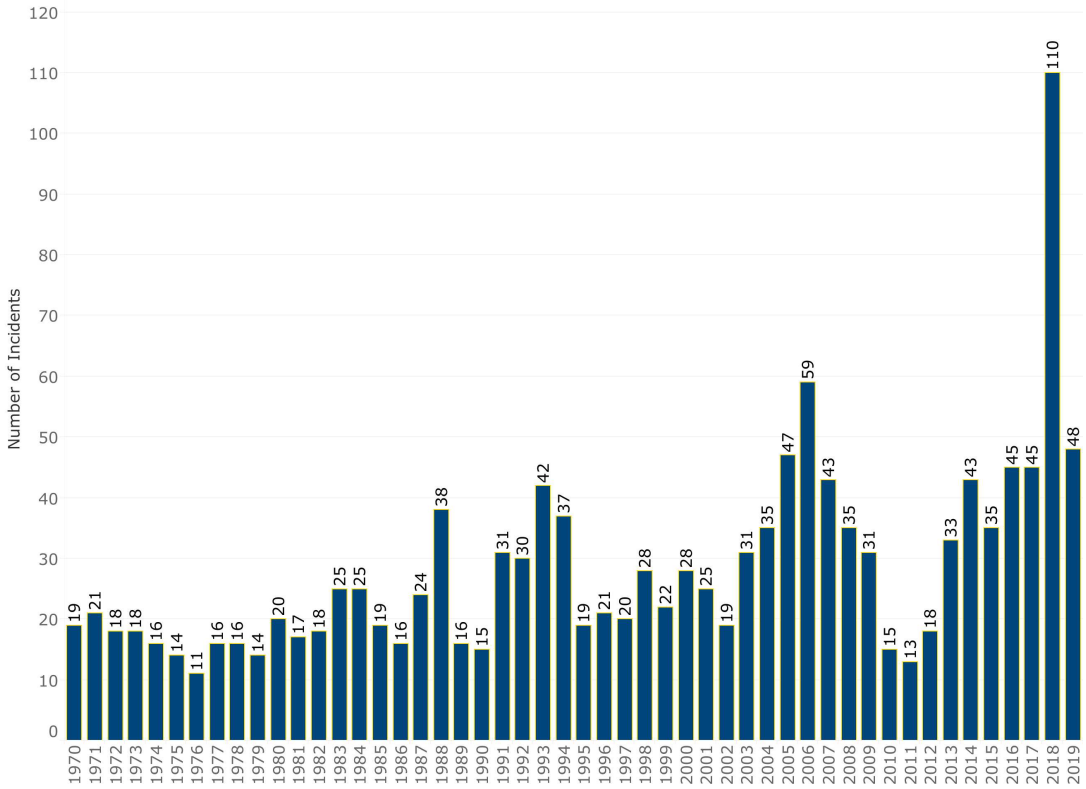


Figure 1.1: Incidents by year

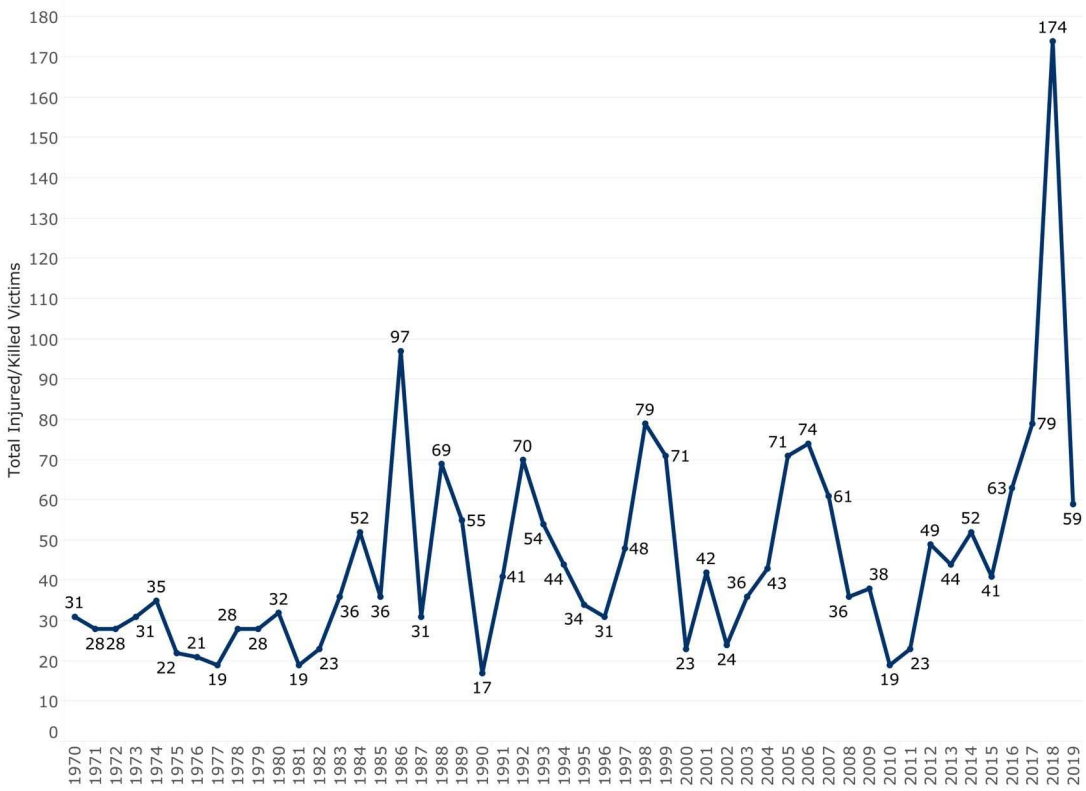


Figure 1.2: Incidents by injured and killed annually

Alongside the number of shootings increasing, deep learning techniques are rapidly improving [2] and object detection in video footage has come a long way the last couple of years. State of the art techniques even allow browsers to detect objects through the web cam. Most surveillance systems still require human supervision to determine whether someone is a threat or not. To automate the process, object detection methods can be used.

1.2 Aim

The purpose of this thesis project is to build an application which can detect guns in a school environment. This can hopefully lead to a reduced police response time, since the perpetrator can be detected the very second a gun is visible.

1.3 Research questions

1. How can earlier knowledge of school shootings be used to build a gun detection system?
2. How can a neural network be trained so that objects similar to guns are not recognized as guns?
3. How can data sent to the server be minimized to achieve greater speed without compromising too much of the accuracy of the detector?

1.4 Delimitations

The most common type of firearm to be used in school shootings are handguns. In a total 946 incidents from 1970 to September 2019, a handgun was used. The second most common is marked as unknown. This category includes the guns which could not be determined from the available information in the news articles. The third most common firearm are the rifles, which were used in 75 incidents, see 1.3 [1]. With this information in mind, the main focus of this application will be to detect handguns in a school environment.

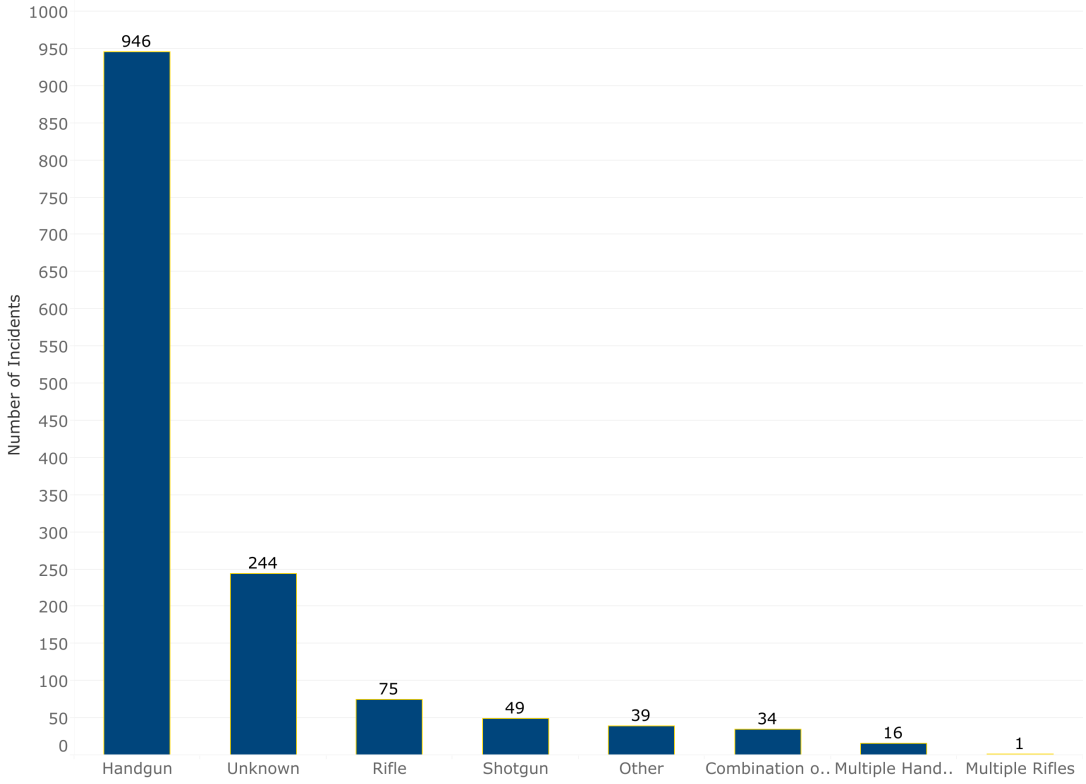


Figure 1.3: Incidents by firearm type

2 Related work

Object detection can be defined as the localization and classification of an object in an image [2]. This is a wide field with a large area of use. This chapter will present related work done in the field of object detection, both early work and the current state of art.

2.1 Early work

Viola-Jones Detector

In 2001 Paul Viola and Michael Jones released the Viola-Jones detector [3]. This was the first object detector [2], and was mainly used for facial detection. The idea behind the algorithm was to create simple rectangular features, where the sum of the pixels in the light area is subtracted from the sum of the pixels in the dark area. Figure 2.1 shows an example of two-rectangle, three-rectangle and four-rectangle features.

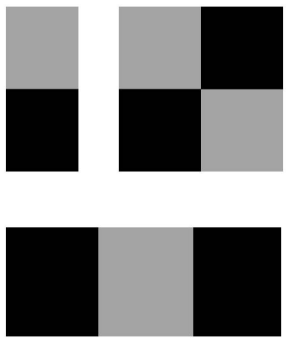


Figure 2.1: Two-rectangle, three-rectangle and four-rectangle features

To calculate the sum and subtraction of many pixels is quite computationally heavy which was solved by introducing an integral image. The integral image at position x, y was defined by taking the sum of the pixels above and to the left in the original image [3]. Using the integral image to calculate differences of large areas sped up the process considerably, and was the key to reaching real time speeds. Finally, a classifier was trained with a set of positive and negative samples, where the various result of the features used on an image were fed into the classifier through several passes to decide if a face was detected or not.

Histograms of Oriented Gradients (HOG)

In 2005, a more efficient detector was introduced by Navneet Dalal and Bill Triggs [4]. This was a person/non-person classification system. The basic idea was to look at how dark a pixel was and compare that to all the surrounding pixels to see in which direction the image got darker (see figure 2.2).

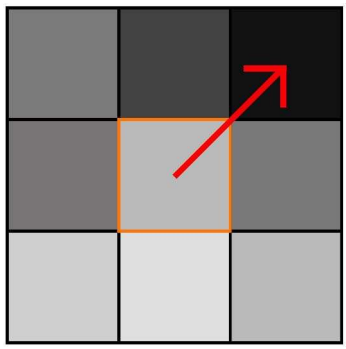


Figure 2.2: Deciding in which direction the image gets darker

This was repeated for every pixel in an image and by dividing the image into smaller parts, the most prevalent direction of all the pixels in this part could be decided. The result was a simple representation of the image and could be compared to a predefined template to decide whether it was a person or not. An example of a HOG image can be seen in figure 2.3.

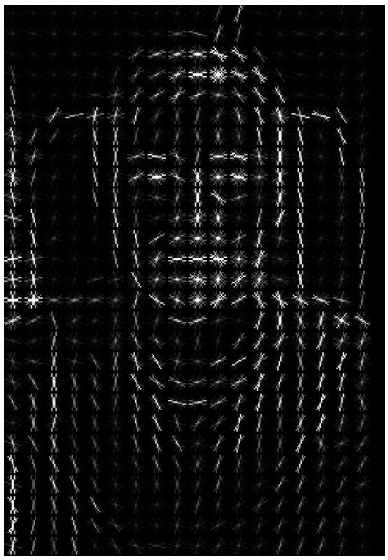


Figure 2.3: HOG image representing a person

2.2 Deep learning era

In 2012, Krizhevsky et al won the yearly image classification and object detection contest *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) [5] with their implementation of what sometimes is refered to as the *AlexNet*. The AlexNet was a *Deep Convolutional Neural*

Network [6] and outperformed all the other contestants. This was a great achievement for deep learning and algorithms have performed significantly better in the contest every year since then. A neural network is loosely based on the brain and its network of connected neurons [7]. The network consists of an input layer, a series of hidden layers and an output layer with neurons connected to each other. In the case of object detection, the input to the network could be an image or a small region from an image, and the output a classification result. The network contains a set of weights which are adjusted through the process of *training* to find the optimal weights for solving the classification problem. A simple example of the architecture of a three-layer neural network can be seen in figure 2.4.

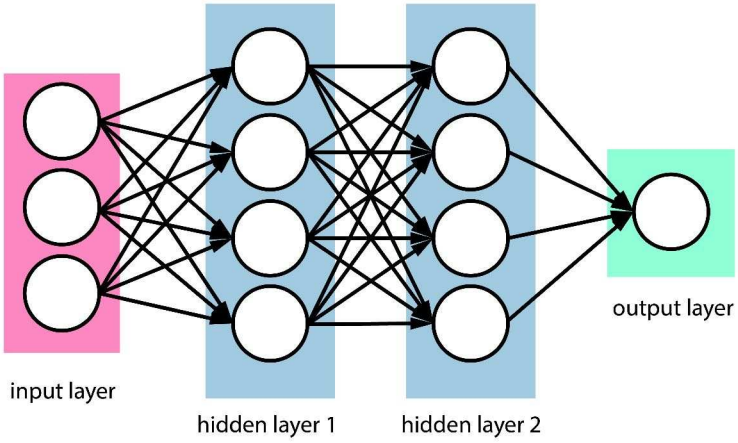


Figure 2.4: An example of a neural network

A *convolutional* neural network (CNN) takes advantage of that the input to the network is an image, and makes certain improvements from that information. Common hidden layers in a CNN are *pooling layers* (for downsampling), *fully-connected layers* but most importantly *convolutional layers* where a set of filters, or kernels, are located [8]. These filters are convolved over small areas of the input image, generating an activation map which describes the filters response at every position of the input. This way, the network creates filters which activate when a certain visual feature is detected, such as an edge. This takes place on the lower layers of the network, and as these layers build on top of each other, the last layers can recognize bigger patterns, and at last entire objects. The convolutional layers do the heavy feature extraction, but to interpret the features at a higher level, a classification is needed. The last layer (before the output layer) of the network is typically a fully-connected layer. This layer is connected to all activations in the precious layer and is used to make the actual classification.

Convolutional Neural Networks in Object Detection

In 2013, the *Overfeat Network* was released by Sermanet et al [9]. This network used CNNs and was the first object detector to use deep learning along with some sort of region proposal method [2]. It used a *sliding window approach* to classify several parts of the image and combine the result to a single prediction which was enclosed by a bounding box. The basic idea of the sliding window approach is to divide the image into smaller regions, classifying every single one and only keeping the ones that classified the wanted object. In 2014, the *Region-based Convolutional Neural Network* (R-CNN) was presented by Girshick et al [10] and it used a more efficient way of doing this. The idea was to propose a set of areas with the help

of *selective search*, before feeding data to the input layer of the network. The selective search approach creates randomly sized areas covering the image. The content in those areas are grouped together based on color, intensity, texture, etc to obtain regions where there might be objects [11]. This led to that the algorithm needed to detect a significantly less amount of areas, which sped up the process. Since the R-CNN, improvements have been released such as Fast R-CNN [12] and Faster R-CNN [13]. These approaches are called *two step detectors* since they are performing a *region proposal step* and an *object detection step* [2].

Single Step Detectors

Unlike the R-CNN, there are also detectors called *single step detectors*, which do not have a region proposal step but tries to do everything in a single step, gaining a large amount of speed. The idea is to directly classify the objects in the proposed areas and leave out low score predictions instead of just guessing if there is an object there or not. Successful single step detectors are the *Single Shot MultiBox Detector* (SSD) released in 2016 by Liu et al [14] and YOLO (You Only Look Once) [15] released the same year by Redmon et al. Since then a lot of new improvements to YOLO have been made such as the latest YOLOv3 [16].

2.3 Gun detection

Historically, there have been mostly work done on detecting concealed guns with X-Ray techniques. These system are often expensive to install and can not be run in open areas [17]. The detector also has to have a constant human supervision. The latest years progressions in the deep learning area has opened for testing new type of detectors. A couple of papers where deep learning has been used to detect guns have been studied. One of these is *A Handheld Gun Detection using Faster R-CNN Deep Learning* [18] by Verma et al. As the title states, the detector recognizes handguns and is based on a Faster R-CNN model. Training data was gathered from the website *Internet Movie Firearms Database* (IMFDB) [19] which stores images of guns, and guns being used in movies in video games. The concept of *transfer learning* was used. The advantage of this technique is that low level features that the network already has learned can be kept, while still retraining the last layers to make it useful in a different use case [20]. The second paper that was studied was *Automatic Handgun Detection Alarm in Videos Using Deep Learning* [17] by Olmos et al. This implementation also uses a Faster R-CNN and focuses on detecting pistols with the motivation that is it the most used handgun in crimes. The concept of transfer learning is used here as well, where a model pre-trained on the ImageNet [21] dataset is used, and extended with 3000 images of guns. The detector was also improved by creating a new class with objects belonging to the background, such as cell phones and pencils. The detector was tested on low quality videos and successfully triggered an alarm in 27 out of 30 situations involving guns.

3 Method

This chapter will present how the implementation of the application was done.

3.1 Structure

The project was implemented in the scripting language *Python* [22]. Python has a number of libraries that are helpful with handling video frames, array operations and most importantly machine learning related tasks. The implementation is structured so that video streams are read locally. This can be from, for example, a web cam, an IP Camera via the Real Time Streaming Protocol (RTSP) [23] or even a video file. The frames are sent to the server to be analyzed by the object detection algorithm. The machine learning framework *Tensorflow* [24], or more specifically the *Tensorflow Object Detection API* [25], was used for the object detection part. The response from the server consists of bounding boxes with labels, scores and category indices. To better understand the upcoming sections, the overall structure is presented in figure 3.1 and will be discussed in detail throughout section 3.2, 3.3 and 3.4.

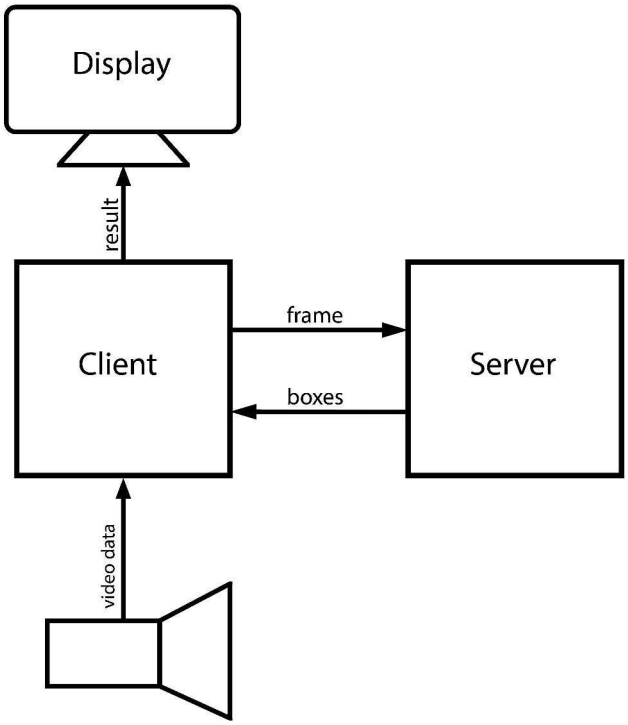


Figure 3.1: Structure of the implementation

With the information from the server, the boxes can be drawn locally on a frame and be displayed. An example of a result drawn on an image can bee seen in figure 3.2. This image is taken from the official GitHub page for the Tensorflow Object Detection API and is the example image of how the output from the API looks like. Here, the labels tell what the algorithm has detected the object as, and the scores represent how certain it is.

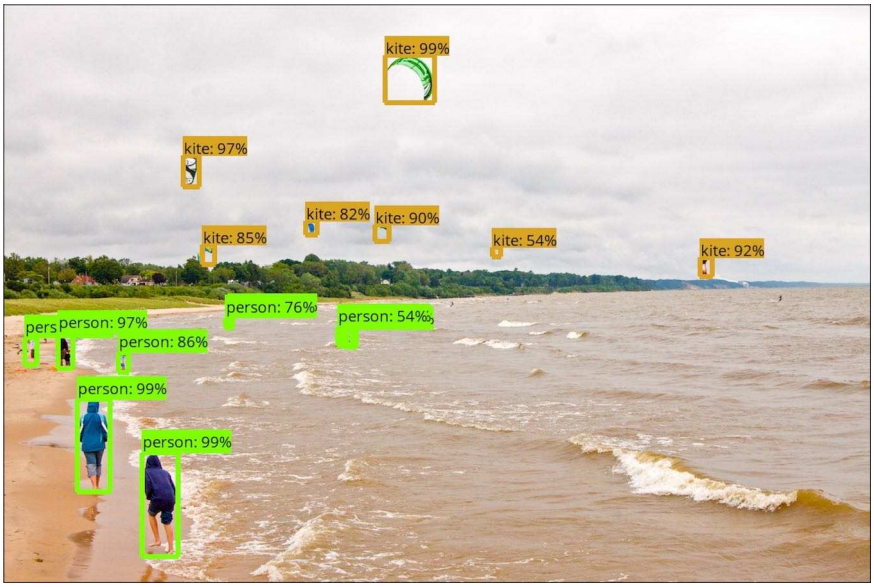


Figure 3.2: Bounding boxes drawn on an image with scores and labels

Earlier approaches

The structure of the application grew from trying different approaches. The first approach was to run everything locally. This involved both reading frames and performing the object detection. The project was implemented on a MacBook Pro 2017 and Tensorflow is not compatible with running on the GPU on MacOS. Therefore, the detection was running on the CPU which is slower. Even if it would be possible to run on the MacBook's GPU, more GPU power would be convenient. This lead to setting up an Ubuntu system on an *Amazon EC2* instance with a more powerful GPU. Amazon EC2 is web service which provide computation power in the cloud [26]. The idea was to use this as the only working station. Both reading, detecting, and displaying would be performed on the instance. This approach was used a while by logging in to it through *SSH* (Secure Shell). The instance is a headless one, meaning there is no monitor connected to it. This meant that a way of displaying the result of the detection was also needed, which had to be done on the local computer, the MacBook. To achieve this, *X11 Forwarding* was used. X11 Forwarding enables a server to run graphical applications on a remote computer [27]. The result of this approach was always a slow one, even when there was no detection involved. To compare, the first approached was tested again, and even if the GPU was more powerful on the instance, there was almost no noticeable difference in speed. Reading and displaying without detection was tested locally as well and even then, the result was slow. Finally, it turned out that the retina screen of the MacBook was the problem. The retina screen has double the amount of pixels than a standard screen [28]. This meant that there were two times as many pixels to be drawn on the screen. To solve this, the application *EasyRes* [29] was used. This application lets the MacBook run on a lower resolution which solved the problem. The X11 Forwarding to the MacBook was still slow though, and before the reason why was determined, the new approach was already decided. Instead, the GPU

related operations were to be performed on the instance, while reading and displaying was to be done locally, as the figure in the beginning of this section suggested. This was a structure which meant that any computer could get the detection results relatively fast, even if, in reality, their GPU or CPU was not powerful enough.

3.2 Client side

The client side is where the frames are read and displayed. This section will present the different aspects of reading frames, but also sending and receiving the data.

Video input

Video streams are read locally frame by frame with *OpenCV*. OpenCV, or Open Computer Vision Library, is an open source computer vision framework which is commonly used when developing applications based on video or image processing [30]. The source of input is chosen and, if desired, the capture of a frame is set to a lower resolution. This is because we do not want to send too large images over network later on.

From the video source, one frame at a time is extracted. From the start, the idea was to send each frame to the server and wait for the response before the frame is displayed back at the client side. Even if the response would be fast, for example 100 milliseconds, the stream would still be perceived as choppy, since the actual frame rate of the stream would be 10 frames per second. This affects the stream in a negative way, and therefore an object detection solution which does not affect the original stream was preferable.

Handling frames and requests

To make the detection part stand-alone, different threads were used. Threads help with performing different tasks in parallel, meaning the frame display does not have to wait for the response from the server. So, the reading and displaying of the frames are performed in the main thread, and sending data and receiving the response is performed in a second thread. The different threads are dependant on each other though. The read frames need to be sent to the server, and the response is needed to draw the final result on a frame. This means that a so called *thread safe* container is needed. Thread safe means that the container can be used between threads in a reliable way, which is an absolute necessity. To achieve this, two queues were used. One queue of frames, which a frame is added to every time a new one is read from the video source. The second queue is holding all the successful responses from the server. In this particular solution *LIFO*, *Last in first out*, queues [31] were used because of several reasons. They are thread safe, and objects are put in order, which is desired in this scenario. This means that the newest frame will always be sent to the server when the last request is done, and the newest response will always be used when drawing the frames. If the queue would be used the other way around, that is *Last in last out*, the object detection would fall behind fast. The LIFO queue makes sure that the latest info is always exchanged between the two threads. The process is shown in figure 3.3.

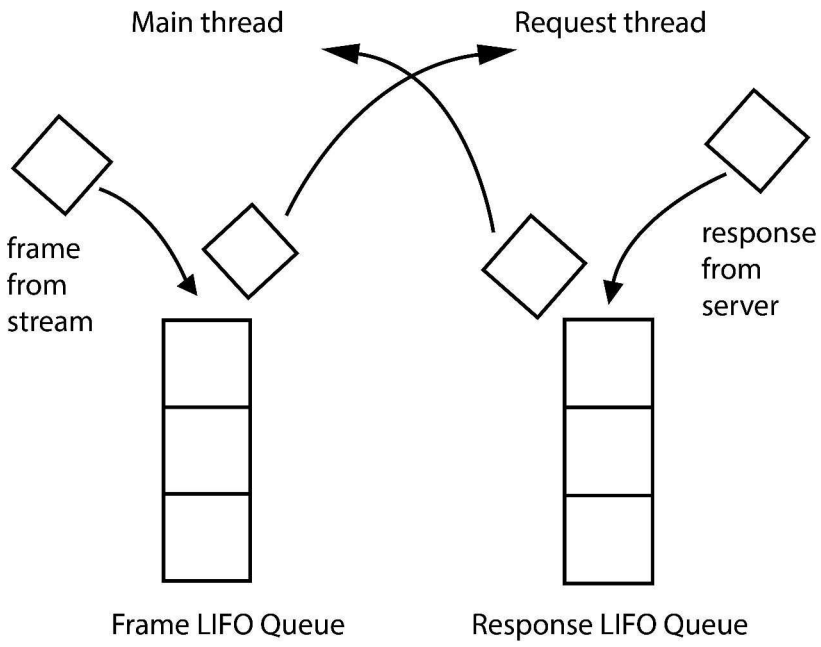


Figure 3.3: LIFO queues between threads

Encoding image and server request

After a frame is pulled from the frame queue it has to be encoded to something that can be sent to the server, but before the image is encoded, it is converted to grayscale, meaning an image with size W x H instead of W x H x 3 is obtained. This is to minimize the amount of data sent, so that the response from the server is as fast as possible. In this case, the *Base64* method was chosen for encoding. Base64 is an encoding scheme commonly used in a number of areas including transferring of image data [32]. Finally, when the image is encoded, a POST request to the server is made and the encoded string is sent along with the original shape of the image. If the client receives a successful response, the response is stored in the response queue.

Visualizing boxes and displaying result

After putting frames in the frame queue, the latest response from the response queue is pulled. This response is created from a couple of frames before the current one since the request takes longer time than pulling a frame from the video source, but this is also what lets the video stream run smoothly, independently of the detection. From the response the client receives bounding boxes, classes, scores and category indices. Tensorflow's Object Detection API is used to draw the boxes on a given frame and the detection threshold is set to 50%, meaning detections with scores under 50% are filtered out. The resulting image can be scaled up if desired, and is then displayed.

3.3 Server side

This section will present the implementation on the server side.

Amazon EC2 instance

For this project an Ubuntu system was set up on an Amazon EC2 instance. As mentioned earlier, Amazon EC2 is a web service which provides computation power in the cloud. The exact type of instance used was a *g3s.xlarge*, which has 4 virtual CPUs and an NVIDIA Tesla M60 GPU. The GPU is optimized for graphics-accelerated applications and has 2048 cores [33], which makes it an optimal GPU for deep learning tasks.

Server

On the Ubuntu system, a server was hosted so that the client can make requests to the server side. Before the first first request, at the start up of the server, the detection model is initialized. To handle the requests, *Flask* was used. Flask is a web framework for python based on the libraries Werkzeug and Jinja2 [34].

Decoding string

To obtain an actual image which the detection can be performed on, the string has to be decoded. The same base64 technique that was presented in section 3.2 was used to decode, with the difference that the shape sent from the client is used. The shape of the frame is sent from the client since the server has to know how to reshape the string it obtained. The encoded string is only one row of characters, and the image is two dimensional. This means that the string has to be reshaped after the decoding. Figure 3.4 shows a simplified example of the process.

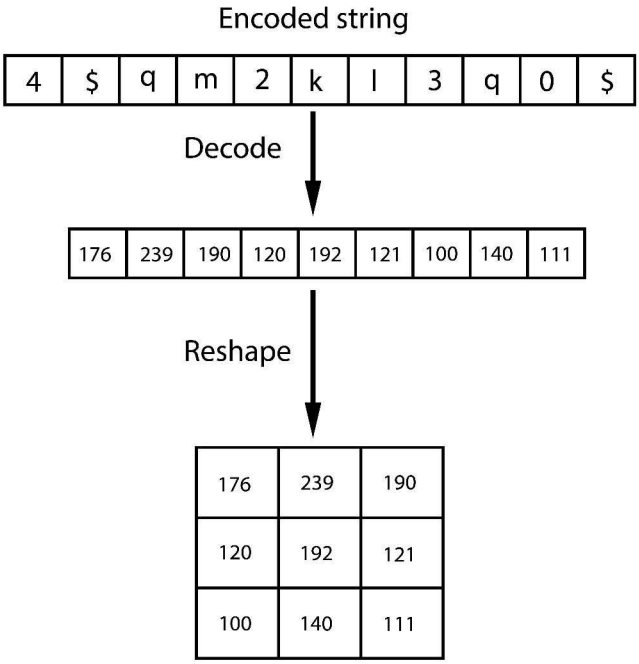


Figure 3.4: Decoding process

Inference

For the *inference* part, *Tensorflow* was used. Tensorflow is an open source machine learning library developed by *Google* and is thoroughly used by a number of famous companies such

as *airbnb*, *Intel*, *Twitter*, *Coca-Cola* and Google itself [24]. On top of Tensorflow, the Tensorflow Object Detection API was built. This is a framework created to simplify the process of detecting objects, constructing models, and training models [25]. The reason for choosing Tensorflow is mainly the large community that surrounds it. It is the most popular framework and its repository on Github has the largest number of contributers and watchers compared to the most popular frameworks for machine learning [35]. When the string is decoded and reshaped, the inference can be performed, that is, the actual detection. At this stage, a *frozen inference graph* is ready to be used to detect the guns. A frozen inference graph is a serialized file which contains information from the training of a model and is created to be read efficiently when performing inference. One problem occurred in the detection process. The object detection API expects color images, or more specifically, images of size W x H x 3. As mentioned before, the frames sent to the server were converted to grayscale to minimize the information sent. This means that they only had a two dimensional size when the inference were to be made. This was solved by checking if the third place in the array was not equal to 3 channels. If it was not, two new channels were created and the content of the first was copied to these two, see figure 3.5. How the final inference graph was created is presented in section 3.5.

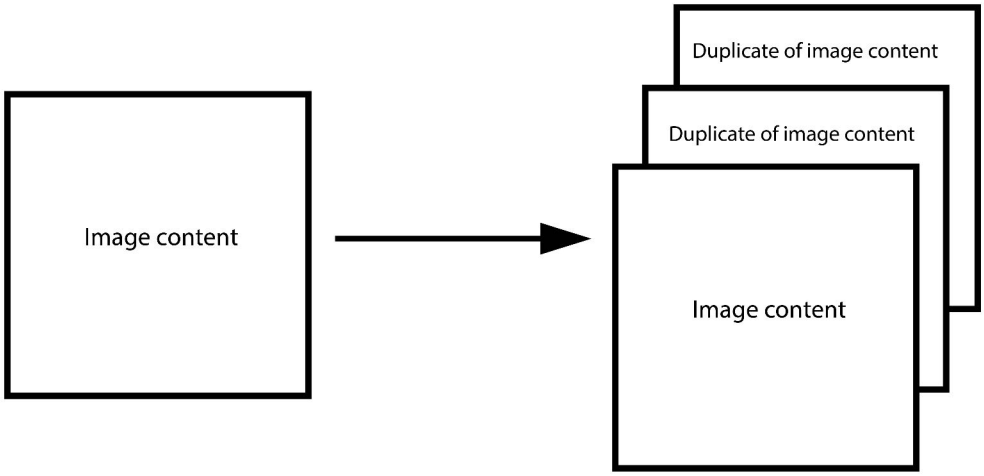


Figure 3.5: Extending a two dimensional image array

3.4 Summary of the architecture

This section will summarize the architecture and the client-server relationship. The architecture is presented in two figures. Figure 3.6 shows the client side and figure 3.7 shows the server side. The figures includes each python script involved in the process.

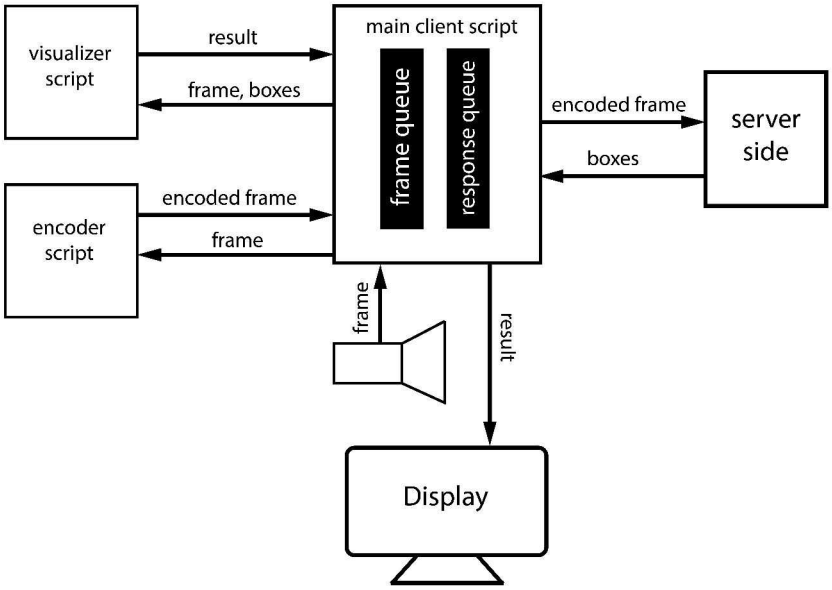


Figure 3.6: Architecture of the client side

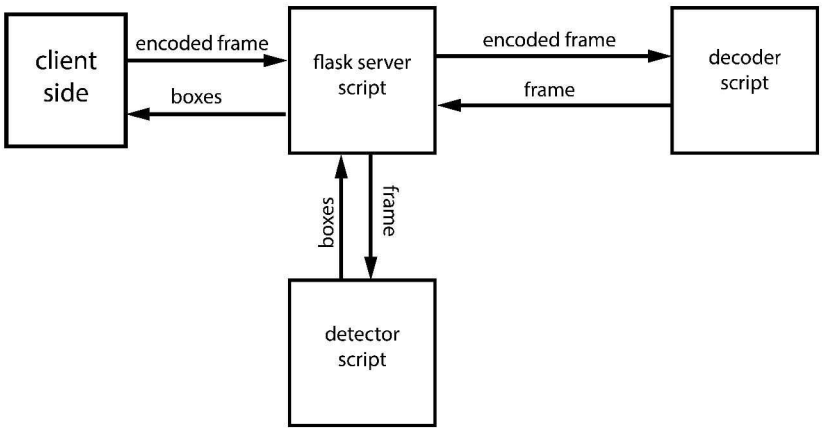


Figure 3.7: Architecture of the server side

All frames that are read from the camera are put into the frame queue. From the frame queue, frames are grabbed and encoded. The encoded frames are sent to the flask server where they first are decoded. The decoded frame, is sent to the detector. In the detector

script, the actual gun detection is performed, and the output consists of bounding boxes, classes, scores and category indices. These are sent as the server response back to the client. On the client side, the response is put into the response queue. From this queue, the response is grabbed and used to visualize boxes on the latest frame in the frame queue. The result is finally displayed on the screen.

3.5 Building the detector

The model that was built for detecting guns was based on a pre-trained model using transfer learning. A pre-trained model is a network trained on a large data set, ready to be used for detection. The idea is to choose a model which solves a similar problem. As an example, a model that can detect cats and dogs can be repurposed to instead recognize persons. This section will present the different steps in the process of creating the gun detector model.

Collecting training data

As was done in [18], training data was gathered from IMFDB. In chapter 1, the most used weapon in school shootings, handguns, was presented. Thus, images from the categories Pistol and Revolver, which can be considered handguns, were used. The images on IMFDB are either close-ups of a gun, or images when they are in use in a movie scene or in a game, see figure 3.8.

(a) A close-up of a Vektor Z88

(b) The same gun used in the movie Death Race 3: Inferno

Figure 3.8: Examples of images from IMFDB

Selecting what type of images to train the network on is important. The training data should be based on in which scenarios the detection algorithm will be used. To gather and analyze data from surveillance footage of school shootings is hard because there are not many videos released to the public. Some of the few examples found can be seen in figure 3.9.

(a) Footage from the Cleveland shooting (b) Footage from the Columbine shooting

Figure 3.9: Footage from two different school shootings

In addition to these images, similar cases were researched such as robberies and other scenarios where guns were used in public places. Figure 3.10 is from a robbery attempt in Mexico [36].

Figure 3.10: Surveillance footage of a robbery attempt in Mexico

From these images, there were several observations made:

- Perpetrators are often seen from a distance
- Surveillance camera videos have varying, not seldom low, quality
- When a gun is seen on a surveillance camera, it is often handheld

With this information in mind, the training images were gathered from IMFDB and chosen so that the guns were handheld and often not too clear or close to the camera as for example in figure 3.8(a). To gather a large amount of images, the *Google Chrome* [37] plugin *Fatkun Batch* [38] was used. This plugin lets a user sort images in one or all open tabs by width, height and keywords. The plugin can also remove duplicates. When the sorting is done, all chosen images can be downloaded at once. The GUI of the plugin can be seen in figure 3.11.

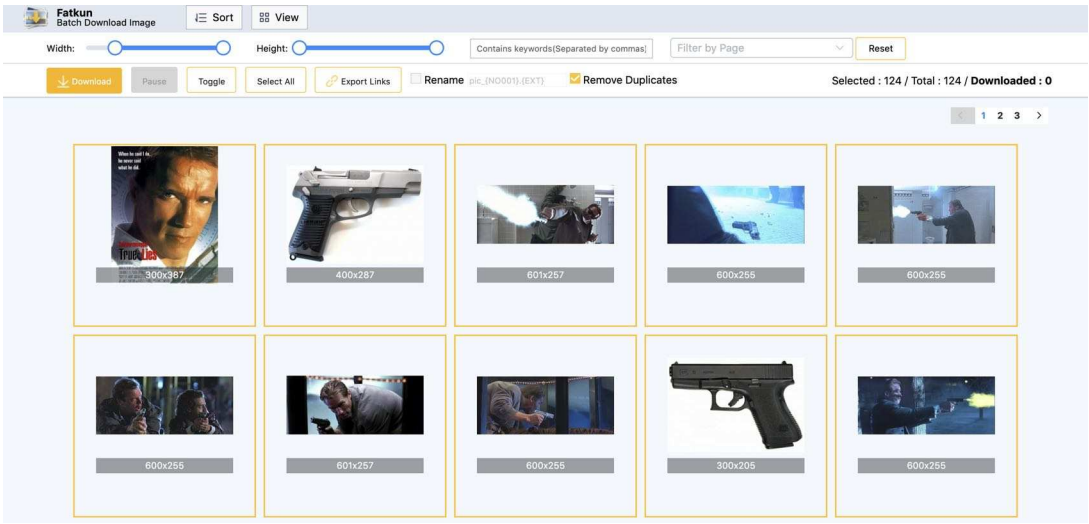


Figure 3.11: Fatkun Batch

The images where guns were handheld and in action had varying sizes but often with a width around 600 pixels and a height around 300 pixels. With the plugin, 1000 images were gathered from the website and the guns in the images were ensured to have a variation when it came to the angle of the gun, the angle of the camera and the distance to the camera. The amount of images were chosen so that it was reasonable to collect and annotate all images within the time frame of this project. Images containing guns that were hard to see even with human observation was not selected for the training data.

Preparing training data

After all the images were gathered, a script was implemented to convert all the images to grayscale. As mentioned in section 3.2, the images sent to the server were converted to grayscale so that less information had to be sent, but as presented in section 3.3 this caused a problem which was solved by duplicating the only channel of the grayscale image to obtain a three dimensional one. To match the format of the frames which were detected, the same technique was used when converting the data set. A selection of images from the set is shown in figure 3.12.

Figure 3.12: Training data

After the images had been processed, a sheet with the right solutions had to be created. This sheet was used in the training process to let the network know what it was supposed to learn. To achieve this, the application *LabelImg* [39] was used. This program lets a user draw bounding boxes around objects and set a label name to those objects. The bounding box is drawn as tight around the object as possible. An example of how the process works is seen in 3.13. Some images of guns had differences such as silencers and long clips. For this implementation, these were considered variations within the gun class and were included in the bounding box.

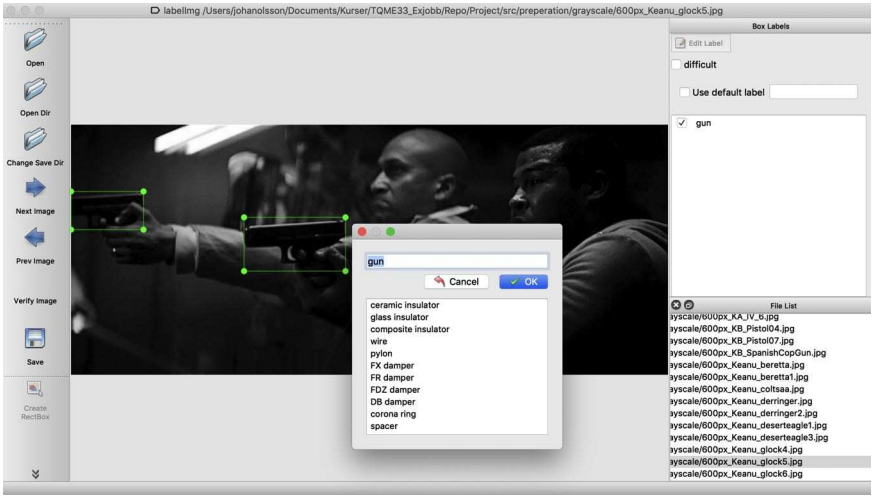


Figure 3.13: Drawing bounding boxes and choosing labels in LabelImg

The application generates an xml-file with objects. Each file contains the name of image file and the width and height of it. It also contains one object per gun, consisting of the class name and four values describing the position of the bounding box in the image: *xmin*, *xmax*, *ymin* and *ymax*. These values make up the points acting as corners of the bounding box, see figure 3.14.

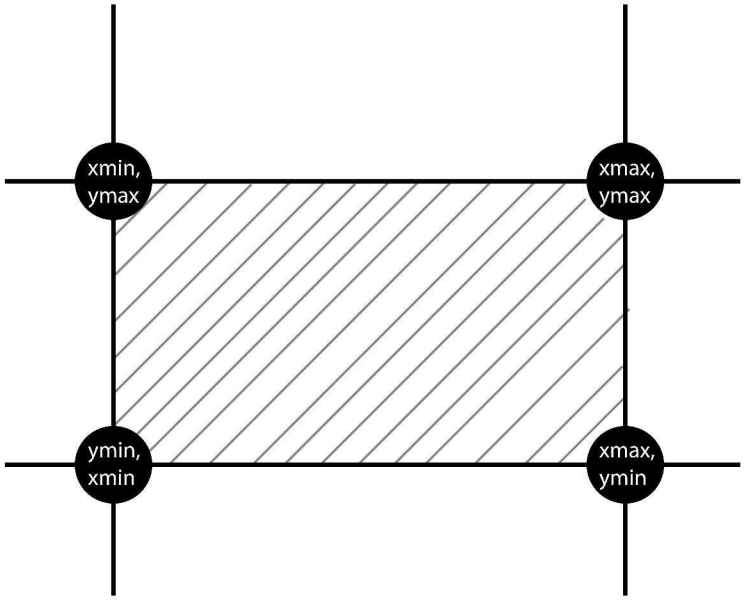


Figure 3.14: Bounding box corner points

The xml files were converted to one csv file holding all annotations for all images in the set. This gives a table of more than 1000 rows, since some of the images contain more than one gun. Figure 3.15 shows a small part of the table.

filename	width	height	class	xmin	ymin	xmax	ymax
600px_ATeamS&W15.jpg	600	255	gun	313	160	400	230
600px_BB2_PPK.jpg	600	329	gun	347	99	433	200
600px_Chuck_4172000.jpg	600	338	gun	388	64	586	189
600px_Blacklist_S01E07_003.jpg	600	337	gun	62	15	112	96
600px_BlindspotS2E04_07.jpg	600	341	gun	177	98	244	133
600px_BlindS2E01_06.jpg	600	341	gun	167	74	204	100
600px_Carlos_pistol_4.jpg	600	248	gun	164	109	266	163
600px_ChuckS4_170.jpg	600	338	gun	265	170	306	214
600px_BlindS2E01_12.jpg	600	341	gun	233	203	322	293
600px_Carlos_P210_1.jpg	600	255	gun	436	56	465	80
600px_B99_0119_G17_01.jpg	600	338	gun	267	115	276	133
600px_B99_0119_G17_01.jpg	600	338	gun	344	122	379	149

Figure 3.15: Train labels

Finally the csv file and the training images were converted to a *TFRecord* file. TFRecord is a format created by Tensorflow. The record's purpose is to be efficiently read linearly [40] and is used during the training process.

Choosing a pre-trained model

There are many pre-trained models trained on various data sets. The models are based on everything from SSDs to R-CNNs. To choose the right model for a certain assignment is always a trade-off between speed and accuracy. This project was developed on a high-end GPU, but there is still some speed difference between models. Since the detection was to be performed on a stream of frames, and the response from the server already took time, it was important to consider the latency of the model. SSDs are generally fast, and therefore, three different SSDs were tested on the server solution. This was done to see how they actually performed on the implementation and to decide which one of the models that would fit this project. The models were downloaded from TensorFlow's official GitHub page where both their speed and *COCO mAP* are displayed, see table 3.1. The COCO mAP is the *mean average precision* for all classes in the *COCO dataset* and is a metric that describes the accuracy of a model on the dataset where a greater value means a more accurate model. COCO mAP values of models on the page vary from 16 to 43. COCO stands for *Common Objects in Context* and is a large-scale dataset commonly used in object detection to train or evalute a model. The dataset contains everything from images of cats and dogs to images of persons and associated annotations [41].

Model name	Speed (ms)	COCO mAP
ssd_mobilenet_v2_coco	31	22
ssd_resnet_50_fpn_coco	76	35
ssd_mobilenet_v1_fpn_coco	56	32

Table 3.1: Speed and COCO mAP for three different models

The *ssd_mobilenet_v2_coco* model is the fastest of the three chosen models. It is built with a MobileNet V2 architecture and is mainly used in mobile devices. It is created to be as light as possible, hence its speed [42]. Its accuracy is considerably lower than the other two models, though. The *ssd_resnet_50_fpn_coco* is an implementation of the *RetinaNet*. RetinaNet was introduced in 2018 and is an SSD with a new *loss function* which increased accuracy [43]. *ssd_mobilenet_v1_fpn_coco* is a lighter version of the RetinaNet built with a MobileNet architecture but still using the loss function of the RetinaNet. The three models were tested by using a web camera from the client side and sending frames of sizes 640 x 480 pixels to the server. The response times are presented in table 3.2 and are average values from 10 responses from the server in a row. The used bandwidth was 500/500 mbps.

Model name	Response time (ms)
ssd_mobilenet_v2_coco	280
ssd_resnet_50_fpn_coco	343
ssd_mobilenet_v1_fpn_coco	318

Table 3.2: Response times for three different models

The ssd_resnet_50_fpn_coco model caused the longest response time and was therefore not considered to be an appropriate model for this setup. The ssd_mobilenet_v2_coco model and the ssd_mobilenet_v1_fpn_coco both performed well on the response test. Considering table 3.1, the ssd_resnet_50_fpn_coco model is more accurate than the ssd_mobilenet_v2_coco, but this was hard to tell by just using the detector on frames from the web camera. Because of this, both models were taken in to the training step to compare how they performed.

Training

When the appropriate models had been chosen, they were used to train the gun class. To use the concept of transfer learning, the top layers of the network is retrained. This means that all COCO classes is forgotten and instead, the network learns to recognize the gun class. The ssd_mobilenet_v1_fpn_coco model was trained with a batch size of 64 images, each image being resized to 640 x 640. The ssd_mobilenet_v2_coco was trained with a batch size of 48 images where each image was resized to 300 x 300. The batch size specifies how many images is put in to the network each training step. The reason for having less and smaller images in the latter mentioned training was because of GPU related memory issues with that specific model. While training, the process could be followed in *Tensorboard*. Tensorboard is a set of visualization tools by Tensorflow which is created to simplify debugging and optimizing models [44]. The total losses from the models were observed during training. The loss is a value describing by how far off the models prediction is from the ground truth and is used to know how much the weights of the network should be adjusted. Figure 3.16 shows a simple example where the red line is a models prediction and the arrows represent the loss.

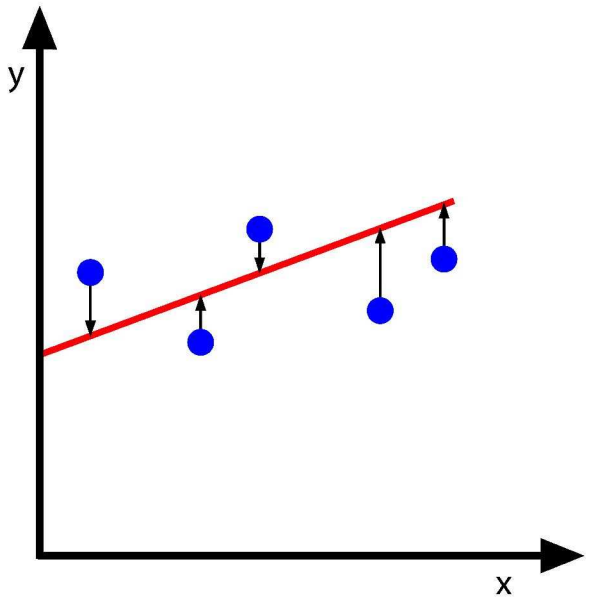


Figure 3.16: Simple example of loss

The total loss is a combination of the *localization loss* and the *classification loss*. The localization loss describes by how far off the predicted bounding boxes are from the actual bounding boxes drawn during the construction of the training set. The classification loss is where classes have been classified as something it is not. In this case, if a gun has not been classified or if something else has been classified as a gun. There are different functions for calculating loss, but the aim is to minimize the given loss function during training. In 2018, Lin et al proposed *Focal loss*, which for the first time let single step detectors achieve the same accuracy as two step detectors, while still maintaining speed [43]. The authors recognized the problem of class imbalance during training as the main obstacle for reaching similar accuracy results as two-step detectors like the R-CNN. Class imbalance can occur if a data set has a lot of negative areas (background) and few positive areas. This makes the training inefficient, but the proposed focal loss could dynamically handle the class imbalance, which led to the network achieving state-of-the-art accuracy results.

The ssd_mobilenet_v2_coco model was trained for 17 hours and 30 minutes, going through 32000 steps, achieving a loss of about 1.3. The second model, the ssd_mobilenet_v1_fpn_coco, was trained for 13 hours and 4400 steps, achieving a loss of about 0.46. The training of the latter mentioned was stopped earlier since the loss was smaller. Figure 3.17 shows the training loss over the number of steps.

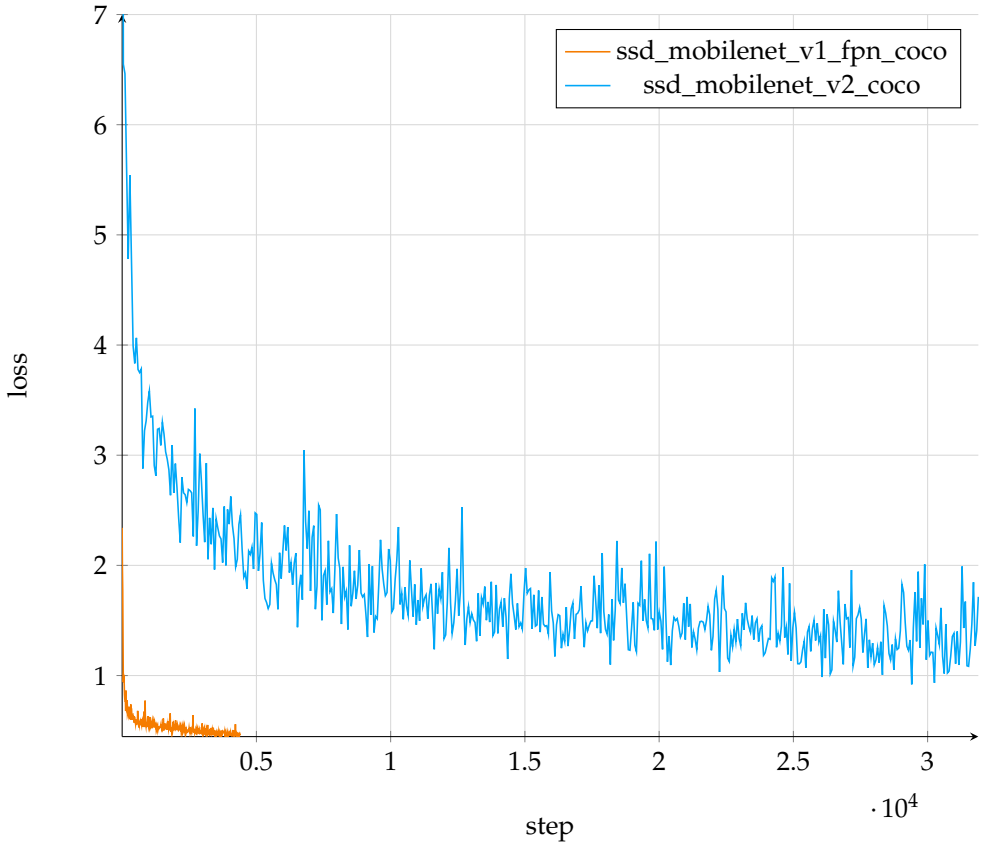


Figure 3.17: Total training loss for both models

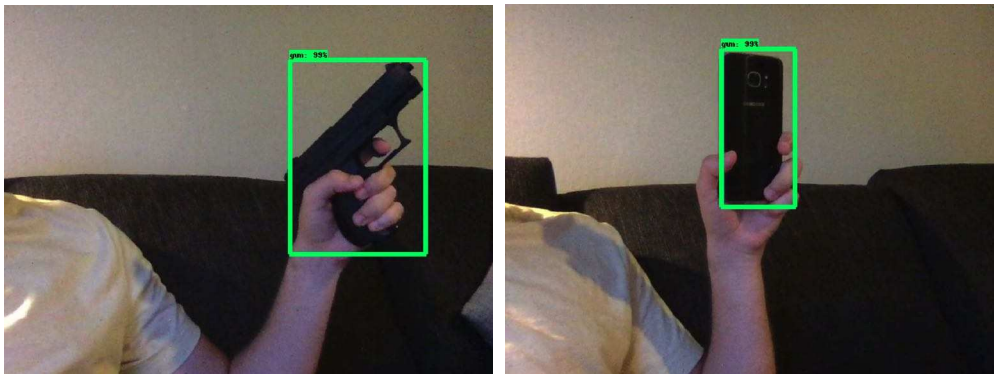
As can be observed from the graph, the ssd_mobilenet_v1_fpn_coco model clearly has a smaller loss than the ssd_mobilenet_v2_coco model from start to end. With that said, the models have two different loss functions so comparing only the losses is not trivial. By once again comparing the COCO mAP of the models and studying the impact of the focal loss the ssd_mobilenet_v1_fpn_coco was chosen as the model to be used in this implementation. The training was stopped after 4400 steps, achieving a loss of 0.46. What it actually meant for the performance was tested later in the evaluation to see if had to be retrained or not. The data from the training was exported as a frozen inference graph which was used on the server side for inference.

Adding a non-gun class

At this stage, the detector was able to recognize guns [1] in a video frame (see figure 3.18(a)), but because of the network only being trained on one class, the classification problem it faced was always to decide if there was a gun or not in an image. This led to the detector recognizing

[1]Throughout this paper, guns can also mean replica guns, toy guns or soft air guns. The difference is assumed to be clear from context.

other objects as guns since it did not have any knowledge of other objects to compare with. An example of this were cell phones, see figure 3.18(b).



(a) The detector recognizing a gun (b) The detector recognizing a cell phone as a gun

Figure 3.18: The detector recognizing two different objects as guns

To improve the detector, and give it examples of non-guns, another class was added. To construct this class, 500 images of various objects were gathered from arbitrary websites. These objects were often handheld objects or objects that had similar visual characteristics to a gun such as flashlights, screwdrivers and cell phones. As with the gun training set, the images were converted to grayscale, annotated and finally converted to a TFRecord. A part of the non-gun training set is shown in figure 3.19.

Figure 3.19: Training data for the non-gun class

The ssd_mobilenet_v1_fpn_coco model was retrained with the TFRecord generated from the gun class and the TFRecord generated from the non-gun class. The model was trained for 18000 steps achieving a loss of 0.34. The training took a total of 55 hours and figure 3.20 shows the total loss over the number of steps.

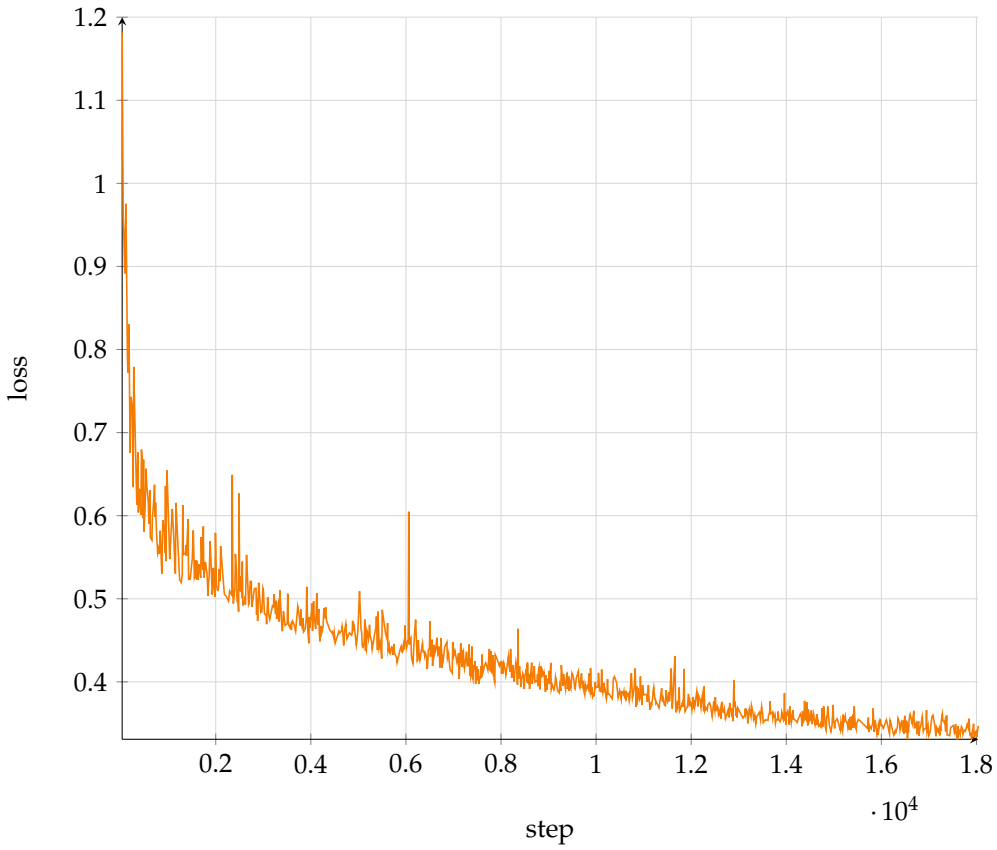


Figure 3.20: Total training loss

After the training, the detector was tested again with the new class. Often, the detector could recognize non-guns as non-guns, and guns as guns, see figure 3.21.

(a) The detector recognizing a gun

(b) The detector recognizing a cell phone as a non-gun

Figure 3.21: The detector recognizing both classes

There were some situations where an object was detected as both a gun and a non-gun because of the classes being similar, but because of the fact that the non-gun class should only be a complement to the gun class, the non-gun detections were hidden in the visualization process. The main purpose for implementing the non-gun class was to make the detector

recognize cell phones and other objects as non-guns *instead* of guns. So, in the cases where a phone only was detected as a non-gun, the class filled its purpose, and after hiding the non-gun detections, the final result of such a case meant that the detector did not recognize the cell phone as anything interesting at all. Cases where a gun was detected as a non-gun was also approved as long as it was recognized as a gun as well. The cases where the non-gun class could impair the detector would be if a gun is detected as a non-gun and only that.

3.6 Evaluating the detector

For the evaluation, a test set of 150 images containing guns were gathered from IFMDB. The images were gathered and processed the same way as the training data were. To evaluate the detector, the metric *mean average precision* (mAP) was used. The mAP metric is defined according to equation 3.1, where the mAP is an avarage value of the *average precision* (AP) for N classes [45].

$$mAP = \frac{1}{N} \sum_{i=1}^{N} AP_i \tag{3.1}$$

The non-gun class was only created to strengthen the precision in which the detector could recognize the gun class, meaning the gun class is the only class, which leads to the mAP being equal to the AP in this case. Tensorflow's Object Detection API was used to automatically calculate the metrics but the following sections present the process behind it and what the various steps mean.

Precision and recall

To calculate the AP, the *precision* and *recall* is calculated. The precision measures how accurate the model is and the recall measures how well the model finds all positives. The *true positives* are the detections where a gun is detected as a gun. The *false positives* are the detections where something other than a gun has been detected as a gun. Finally, the *false negatives* are the guns that the detector failed to recognize. The precision and recall is defined as equation 3.2 and 3.3.

$$precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{3.2}$$

$$recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{3.3}$$

Intersection over union

To find the true positives, false positives and false negatives and also take into account how well the detector locates the objects, the *Intersection over Union* (IoU) is calculated. The IoU is given by the ratio of the area of intersection and the area of union between the predicted bounding box and the ground truth bounding box. It is calculated as suggested in equation 3.4 and the areas are illustrated in figure 3.22, where the intersection is the light blue area and the union covers both the dark and the light blue area.

$$IoU = \frac{area\ of\ intersection}{area\ of\ union} \tag{3.4}$$

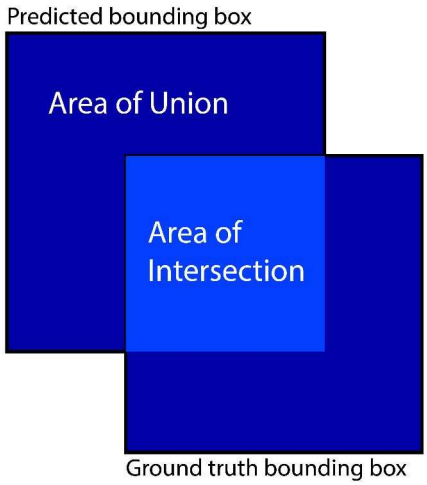


Figure 3.22: Area of union and area of intersection

With the calculated value of IoU, the true positives, false positives and false negatives can be determined by using different thresholds as follows:

- **True positive:**
  A detection is considered true positive if IoU is greater than the threshold value
- **False positive:**
  A detection is considered false positive if IoU is less than the threshold value or if a duplicate bounding box is obtained
- **False negative:**
  A detection is considered false negative if IoU is greater than the threshold value but the classification is wrong

With the number of true positives, false positives and false negatives determined, the precision and recall is calculated for each image in the testing set. The result is plotted in a precision-recall curve where the y-axis represents the precision and the x-axis represents the recall. Figure 3.23 shows a simplified example of what a precision-recall curve could look like.

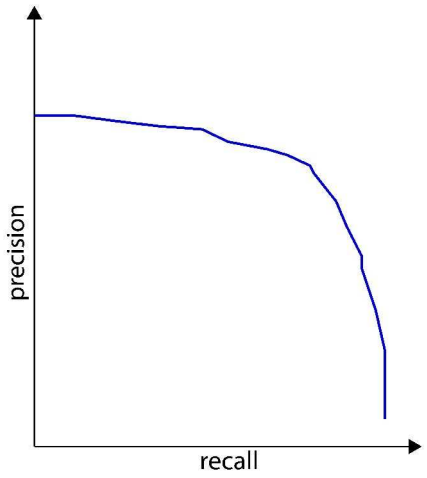


Figure 3.23: Simplified example of a precision-recall curve

The AP is defined as the area under the precision-recall curve and is given by equation 3.5, where *p(r)* is the precision as a function of the recall.

$$AP = \int_0^1 p(r)dr \tag{3.5}$$

Before actually calculating the area, the curve is smoothened to make the calculated AP value less sensitive to variations. This is done by taking each recall value and setting it to be equal to the next maximum value to the right of it. From this, a new interpolated precision-recall curve is obtained. The area under the curve is calculated, resulting in the AP. Different competition uses different metrics and different definitions of the metrics. In this implementation, COCO's detection evaluation is used where 12 different metrics are given under 4 different categories [46]. COCO is using an interpolated AP as described above and in their definition of AP, an average of multiple IoU:s are used. The twelve metrics are listed below:

- **Average precision (AP)**
  - *AP*: Average precision with an average of several IoU:s with different thresholds from 0.50 to 0.95 with a step size of 0.05
  - *AP IoU=0.50*: Average precision with a IoU threshold of 0.50
  - *AP IoU=0.75*: Average precision with a IoU threshold of 0.75
- **AP across scales**
  - *AP Small*: Average precision for small objects with an area of < 32 x 32
  - *AP Medium*: Average precision for medium objects with an area larger than 32 x 32 but smaller than 96 x 96
  - *AP Large*: Average precision for large objects with an area larger than 96 x 96
- **Average recall (AR)**
  - *AR Max=1*: Average recall given 1 detection per image
  - *AR Max=10*: Average recall given 10 detections per image
  - *AR Max=100*: Average recall given 100 detections per image
- **AR across scales**
  - *AR Small*: Average recall for small objects with an area of < 32 x 32
  - *AR Medium*: Average recall for medium objects with an area larger than 32 x 32 but smaller than 96 x 96
  - *AR Large*: Average recall for large objects with an area larger than 96 x 96

3.7 Evaluating the non-gun class

Since the metrics evaluate the detectors precision for certain classes the non-gun class itself can not be evaluated the same way. This is because it was constructed to increase the accuracy of the gun class, and it is not obvious what to include and not to include in such a test. Instead, the detectors number of true positives, false positives and false negatives were counted. In this case, the accuracy of the localization was not considered and instead, these rules were applied:

- *True positive*: If the detector recognized a gun correctly (even if more than one box covered the gun)
- *False positive*: If the detector recognized something else as a gun

- *False negative*: If the detector failed to recognize a gun

The same 150 test images from IFMDB were used during this test, with the addition of 150 images not containing guns, gathered from arbitrary websites and recorded from a security camera. An example of the test images can be seen in figure 3.24.

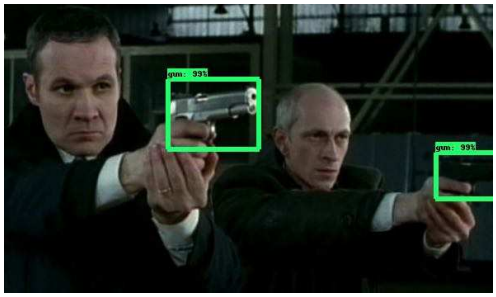
Figure 3.24: Some example of images in the test set

The test was performed with two different stages of the detector: Before and after the non-gun class was added. The result is presented in the next chapter.

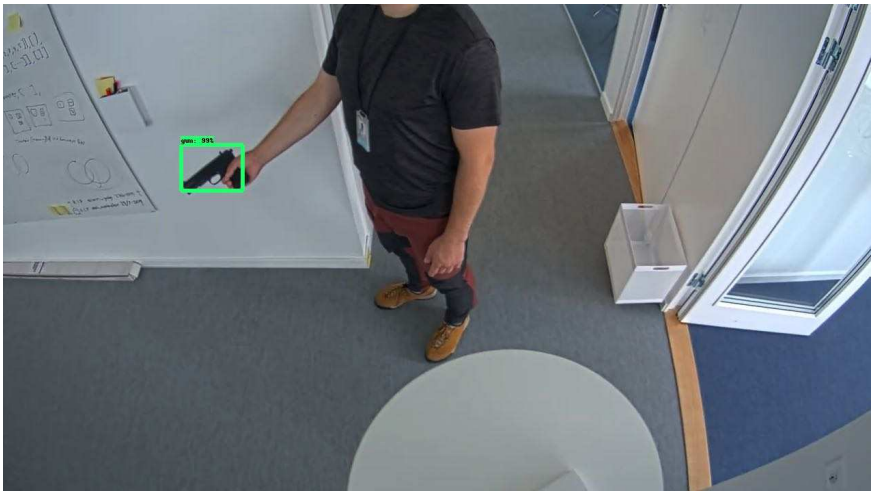
4 Results

In this chapter, the result is presented. The result is divided into testing the response time of the server solution, and the accuracy of the gun detection algorithm. The visual result of the implementation is shown in figure 4.1, where the gun detector has been applied to a few different video frames and images.

(a) Detector recognizing a gun through a web camera



(b) Detector recognizing guns in an image from IFMDB



(c) Detector recognizing a gun through an AXIS security camera

Figure 4.1: The detector recognizing guns in video frames and images

4.1 Client-server solution

The time it takes for the client to receive a result back from the server is highly dependent on how large the sent data is. In this implementation, frames are converted to grayscale to decrease the size. Besides that, frames are scaled down to decrease the size even more, but

there is a draw back to this. If the size of the detected frame is smaller, there will be fewer pixels in the image that represent the gun, and it can therefore affect the accuracy of the detector on larger distances. To test this, distances of 0.5 m, 1 m, 2 m, 3 m, 5 m, 8 m and 10 m were marked up in a corridor and for each scale, a gun was held at the checkpoints with an AXIS P1367 Network Camera [47] pointing towards it. The setup can be seen in figure 4.2.

(a) Corridor marked up with distance checkpoints

(b) AXIS Camera pointing towards the corridor

Figure 4.2: Setup for testing the ability to detect a gun at various distances

The following table present the detectors ability to detect guns at the various checkpoints and at different scales. The input frames are of size 1920 x 1080 and is captured with a bandwidth of 500/500. The response time from the server, without performing any detection, was 5 ms. If the detector recognized a gun at least one frame at a certain distance, the ability to detect was considered true.

Scale	Response time (ms)	Longest distance (m)
1 (1920 x 1080)	480	3
1/2 (960 x 540)	340	3
1/4 (480 x 270)	270	3
1/8 (240 x 135)	210	3
1/16 (120 x 68)	180	2
1/32 (60 x 34)	110	1

Table 4.1: Longest distance for detecting gun at various scales

4.2 Gun detector

The section presents the result of the final detector with various metrics, and also a comparison before and after the non-gun class was added.

Final result

Tables 4.2, 4.3, 4.4 and 4.5 present the twelve metrics commonly used when evaluating models on COCO as presented in section 3.6.

Average precision (AP)	Result
AP	51.1 %
AP IoU = 0.5	85 %
AP IoU = 0.75	56.3 %

Table 4.2: Average precision for different IoU thresholds

AP across scales	Result
AP Small	30 %
AP Medium	53 %
AP Large	56.4 %

Table 4.3: Average precision for different object sizes

Average recall (AR)	Result
AR Max=1	52.4 %
AR Max=10	62 %
AR Max=100	65.2 %

Table 4.4: Average recall for different max detections

AR across scales	Result
AR Small	43.1 %
AR Medium	64.8 %
AR Large	76 %

Table 4.5: Average recall for different object sizes

With and without non-gun class

Table 4.6 shows the number of true positives, false positives and false negatives for the gun detector at two different stages. The detector without the non-gun class was trained for 4400 steps with 1000 images of guns, achieving a total loss of 0.46. The detector with both the gun class and the non-gun class was trained for 18000 steps with 1000 images of guns and 500 images of other objects, achieving a total loss of 0.34. The total number of guns in the test set of 300 images were 165.

	True positives	False positives	False negatives
Without non-gun class	140	21	25
With non-gun class	140	11	25

Table 4.6: Number of true positives, false positives and false negatives for the detector with and without the non-gun class

The precision and recall is calculated as equation 3.2 and 3.3 suggests and the result is presented in table 4.7.

	Precision	**Recall**
Without non-gun class	86.9 %	84.8 %
With non-gun class	92.7 %	84.8 %

Table 4.7: Precision and recall for the detector with and without the non-gun class

5 Discussion

This chapter contains discussions about the results and the method as well as ethical and societal aspects related to the project.

5.1 Results

Client-server solution

The response time from the server included several steps like sending data, decoding data, performing inference and sending results back to the client. Sending image data to the server was indeed a time consuming task, and sending frames of 1920 x 1080 took 480 ms which equals about 2 frames per second. The one obvious thing that could be changed to achieve greater speeds was to send less data. The results show that the frames could be scaled down as much as 1/8 before affecting the longest distance of the detector and then instead achieving a response time of 210 ms (circa 5 frames per second). On longer distances there is of course always a risk of loosing too much image information and thus affecting the detector. If the response time is too high though, it can affect the detector's ability to detect guns flashing by fast, for example if someone would run with a gun. Since the latest frame available in the frame queue is grabbed right after a response is finished, frames will be missed and if someone is running by fast and only one or two frames are analyzed then one false negative response from the detector can lead to the gun passing by unnoticed.

When testing different distances, several distance checkpoint from 0.5 meters to 10 meters were marked up in belief that at least the highest resolution could lead to a detection at 10 meters, but the detector had trouble detecting guns even at 5 meters. This is of course problematic from a security camera point of view, where guns often are seen from a distance, especially if the cameras are used for monitoring large areas. Possibly, the training data could have been chosen differently to use more images where guns were further away, but such an image set would be hard to find. A second problem would also be the response time. When detecting guns on large distances, the frame could not be scaled down as much, and the response time would then be too slow.

Gun detector

The gun detector was evaluated using the twelve metrics commonly used when evaluating on the COCO data set and the detector achieved an AP of 51.1 %. With an IoU threshold of 0.50, the result was significantly better, 85 %. This means there were quite a lot of localization error. The AP across scales show that the detector did not recognize smaller guns or guns further away in an image as good as larger ones. AP Small gave a value of 30 % while the AP Large was 56.4 %. This was expected and matches the result of the client-server solution where smaller frames made the detector more inaccurate. The average recall for different max detection values did not give any unexpected results. For a larger value of max detections,

the result was better but the difference was not unusual. The AR across scales showed that the detector once again had troubles detecting small objects and the difference between AR Small (43.1 %) and AR Large (76 %) was big. Overall, the metrics gave a satisfactory result, but there will be a trade off between recognizing guns on large distances and the response time from the server. One important thing to keep in mind as well is that the constructed test data were also images from IFMDB, just as the training data were. Of course, none of the training images reoccurred in the test data but the images where gathered from different movies with similar light settings, similar quality and often similar distance to the camera. This could have led to that the result looks better than it would on data from a real life scenario where the distance to the object, the angle of the camera and quality of the image could be a lot different. Once again, it comes down to the gathering of the data. A data set which would match real life scenarios was considered very hard to find.

Non-gun class

The reason for implementing a non-gun class was to remove some of the false detections on cell phones and other handheld objects that the detector could recognize as guns. A wide range of objects was trained and put in to a joint class. The objects were often cell phones, remote controls, screw drivers, etc. As table 4.6 shows, the non-gun class did exactly what it was supposed to. The number of true positives and false negatives were unchanged while the number of false positives decreased, resulting in a precision of 92.7 % instead of 86.9 % on the test set. Worth mentioning is that the detector was trained for less steps without the non-gun class, meaning that some of the results could be thanks to a longer training, but with the non-gun class, there were also more images to train on, so it is not obvious what would be a perfectly fair comparison when only measuring the impact of the non-gun class.

5.2 Method

The concept of reading frames locally, and sending them to a server for detection had both advantages and disadvantages. The main advantage is that a fast detection can be done on the server side, independently of the GPU power on the client side. This opens up for anyone being able to run this system as long as the bandwidth is fast enough. The main disadvantage was that the process was never as fast as if it would be if it ran on the same high-end GPU locally without any network involved. The main problem is that such a GPU is costly and for a customer, the cheaper solution would probably be the server-client solution.

Client side

On the client side, OpenCV was used for reading frames and displaying frames. OpenCV was simple to use and could read from both video files, web camera and RTSPs. Handling frames and responses through LIFO queues between threads worked well and made the stream independent of the requests, making the displayed video more pleasant to look at and easier to follow. The data was minimized by converting it to grayscale, only sending two channel frames to the server. There were never any investigation on how grayscale images affected the detection, but no visual differences could be seen with color vs grayscale. Probably, it could affect objects which have distinguishing colors but since the training data was constructed to match the frames sent for detection, that problem was eliminated. To encode and decode frames, the Base64 scheme was used. During the implementation, two reflections of the choice of this scheme have been made. Firstly, the data encoded by this scheme is approximately 33% larger than the original data [48], and since it is the encoded that is being sent to the server, this affects the response time. Another aspect of the encoding process is safety. At this stage, there is nothing done to protect the sent data. To solve this, one could example

encrypt data on the client side and decrypt it on the server side. Because of the limited time frame of this project, these two reflections are considered to be future work.

Server side

The server was hosted on an Amazon EC2 instance with a Tesla M60 GPU. The setup was easy to use while in place but the instance was pretty costly in the long run. A solution would be to host your own GPU server to lower the cost for the customer. The use of Tensorflow worked well and the fact that the community around it is wide, it was often easy to find solutions and documentation to help with the implementation. The API helped with everything from inference to exporting graphs and visualizing boxes back at the client side. The one disadvantages was that the boxes was not customizable and a custom design was hard to implement.

Building the detector

The first step in building the detector was to gather training data. Data matching such a specific situation was hard to find. Some observations were made such as that the cameras often captures at lower quality and a perpetrator is often seen from a distance. The gathered data was from IMFDB and were images from movies. The light and quality of these images were of course not matching the real life scenario perfectly but it was the best alternative. The tools used to gather and annotate data worked well but sometimes it was not obvious to know what to include in the training data. Guns which were hard to see with human observation was not chosen for the training data, but could maybe have helped the detector. The chosen pre-trained model did well. No problems with accuracy occurred and the speed was not too slow. The next step was the training process. Here, only the training loss was used too see how the training progressed. This could have been done a lot better by involving evaluation during the training. The evaluation could have shown how the mAP improved, and through that, it would have been more obvious to know when to stop the training. The comparison between the two models made in section 3.5 may not have been fair. The models obviously used two different loss function and had slightly different architecture, which may have made the comparison inaccurate. The COCO mAP did show that the ssd_mobilenet_v1_fpn_coco was more accurate, so the model chosen was probably still the best choice for this implementation. The last step in building the gun detector was to add the non-gun class. This class did what it was suppose to but to create such a wide class, containing several different types of objects is not optimal. For this project, it was a question of what could be done within the time frame, and therefore a non-gun class was created. If there was more time, this class would have been divided into several classes, such as a cell phone class, a screw driver class, a remote control class and so on. This is further discussed in section 6.2.

5.3 The work in a wider context

There is always a trade-off between the security and the privacy of people [49]. More and more people value their integrity and since the introduction of GDPR [50], there is quite a large difference between how personal data can be handled in the European Union and the United States. The decisions made from statistics from the K-12 School Shootings Database was information based on schools shootings in the United States. Thus, this projected did not focus on schools outside of the country, meaning the GDPR discussion is not as relevant for this project. There is of course still ethical aspects to using cameras in school for detection. In 2018, Joy Buolamwini published the study *Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification* [51] which investigates how skin color affects various face recognition systems' accuracy. On various large face recognition software, Microsoft had the best performance, achieving an error rate of 12.9% on darker skinned people and 0.7 % on

lighter skinned people. IBM achieved the worst result, having an error rate of 22.4 % on darker individuals which was almost 7 times higher than on lighter faces. In a case where the gun detector would try to recognize faces along with the guns to look for guilty persons, obviously problems like this could occur. In the case of this project, the detector exclusively looks for guns and have no knowledge of who is holding the gun or what gender or ethnic background the person has, and since guns should not exist inside a school environment, looking for guns and only that is justified.

6 Conclusion

6.1 Research questions

1. **How can earlier knowledge of school shootings be used to build a gun detection system?**

   Surveillance footage from school shootings is often secured from the public, which makes it hard to study video footage from such situations. To relate the implementation to school shootings, the K-12 School Shootings Database can be used for statistics and decisions can be made from it, whenever possible. In this project, the database was used to see which the most common weapons were, to know what to focus on initially.

2. **How can a neural network be trained so that objects similar to guns are not recognized as guns?**

   To help the network distinguish guns from similar handheld objects such as cell phones, a non-gun class can be implemented. This class helped with lowering the amount of false positives from 21 to 11 on a test set with 300 images containing 165 guns.

3. **How can data sent to the server be minimized to achieve greater speed without compromising too much of the accuracy of the detector?**

   The image data can be converted to grayscale to send frames with one channel instead of three. Furthermore, the frame can be scaled down. On a frame which initially has a resolution of 1920 x 1080, a downscaling of as much as 1/8 can be done before affecting the distance at which the detector can recognize a gun.

6.2 Future work

Extending with more classes

For this project, one type of firearm, the handgun, was chosen to focus on for the detector. This was based on that it it the most common weapon used in school shootings as figure 1.3 suggested. Before this detector can be considered good enough, of course it has to be extended to recognize all types of firearms, such as rifles and carbines. As discussed in section 3.5, the classification problem is easier to solve the more classes the network knows about. So, the extension with more firearms would help the already known gun class. In this implementation, a non-gun class was added which contained several different objects. This was to help the gun class, but if this project would be extended, this non-gun class could be split into several classes, as mentioned in chapter 5. Another alternative would be to replace the non-gun class with all the COCO classes instead of creating a non-gun class from scratch. This would improve the gun detector considerably.

Using 3D models to construct training- and test data

One of the main challenges was finding data to match real life scenarios. The images used were from movies with often good quality and light, which differ a lot from the images from security cameras presented in section 3.5. To improve the testing data and even the evaluation process, 3D models could be used to generate a large amount of image data. The idea would be to create some sort of a 3D application where 3D models of various persons with weapons in hand could be placed in various environments. The models and the camera could then be rotated and modified and for each modification, an image could be rendered automatically. There would be several advantages of such an application. All the parameters related to the camera could be matched to any real life security camera. This means that the angle from which the camera points could be matched as well as the lens distortion and also the quality of the rendered image. The environment in which the person is placed could always be school environments with any light settings desired. The person could be rotated and various poses could be captured, where the person can hold a weapon of any category. From this application, a large amount of image data could be generated and all the annotations to each image could automatically be generated since the exact position of the weapon is known.

Bibliography

[1] *K-12 School Shooting Database*. https://www.chds.us/ssdb/category/graphs/. Accessed: 2019-09-01.

[2] Karanbir Singh Chahal and Kuntal Dey. "A Survey of Modern Object Detection Literature using Deep Learning". In: *arXiv preprint arXiv:1808.07256* (2018).

[3] Paul Viola, Michael Jones, et al. "Rapid object detection using a boosted cascade of simple features". In: *CVPR (1)* 1 (2001).

[4] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: 2005.

[5] *ImageNet Large Scale Visual Recognition Challenge*. http://image-net.org/challenges/LSVRC/. Accessed: 2019-07-25.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012.

[7] Tom Hope, Yehezkel S. Resheff, and Itay Lieder. *Learning TensorFlow: A Guide to Building Deep Learning Systems*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491978511, 9781491978511.

[8] *Convolutional Neural Networks (CNNs / ConvNets) - CS231n Convolutional Neural Networks for Visual Recognition*. http://cs231n.github.io/convolutional-networks/#overview. Accessed: 2019-08-29.

[9] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. "Overfeat: Integrated recognition, localization and detection using convolutional networks". In: *arXiv preprint arXiv:1312.6229* (2013).

[10] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014.

[11] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. "Selective search for object recognition". In: *International journal of computer vision* 104.2 (2013).

[12] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015.

[13] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*. 2015.

[14] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016.

[15] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

[16] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[17] Roberto Olmos, Siham Tabik, and Francisco Herrera. "Automatic handgun detection alarm in videos using deep learning". In: *Neurocomputing* 275 (2018).

[18] Gyanendra K Verma and Anamika Dhillon. "A Handheld Gun Detection using Faster R-CNN Deep Learning". In: *Proceedings of the 7th International Conference on Computer and Communication Technology*. ACM. 2017.

[19] *Internet Movie Firearms Database*. http://www.imfdb.org/wiki/Main_Page. Accessed: 2019-06-12.

[20] *Transfer Learning - CS231n Convolutional Neural Networks for Visual Recognition*. http://cs231n.github.io/transfer-learning/. Accessed: 2019-08-29.

[21] *ImageNet*. http://www.image-net.org/. Accessed: 2019-08-30.

[22] *Python*. https://www.python.org/. Accessed: 2019-06-12.

[23] Henning Schulzrinne. "Real time streaming protocol (RTSP)". In: (1998).

[24] *Tensorflow*. https://www.tensorflow.org/. Accessed: 2019-06-10.

[25] *Tensorflow Object Detection API*. https://github.com/tensorflow/models/tree/master/research/object_detection. Accessed: 2019-06-10.

[26] *Amazon EC2*. https://aws.amazon.com/ec2/. Accessed: 2019-06-10.

[27] Ulrich Flegel. "The Interaction between SSH and X11". In: *Unknown* (1997).

[28] *Retina Web Graphics Explained: 1x versus 2x (Low-Res versus Hi-Res)*. https://www.danrodney.com/blog/retina-web-graphics-explained-1x-versus-2x-low-res-versus-hi-res/. Accessed: 2019-06-12.

[29] *EasyRes)*. http://easyresapp.com/. Accessed: 2019-06-12.

[30] *OpenCV*. https://opencv.org/. Accessed: 2019-06-11.

[31] *Queue - A synchronized queue class*. https://docs.python.org/2/library/queue.html. Accessed: 2019-06-10.

[32] Kevin Fiscus and D Shinburg. "Base64 Can Get You Pwned". In: *GIAC (GCIA) Gold Certification* (2011).

[33] *Tesla M60 GPU Accelerator*. https://www.nvidia.com/object/tesla-m60.html. Accessed: 2019-06-10.

[34] *Flask*. http://flask.pocoo.org/. Accessed: 2019-06-10.

[35] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López Garcıa, Ignacio Heredia, Peter Malık, and Ladislav Hluch. "Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey". In: *Artificial Intelligence Review* (2019).

[36] *CCTV VIDEO: Robber pulls out gun; man in cowboy hat tackles him to the ground*. https://indianexpress.com/article/trending/viral-videos-trending/armed-robber-steps-shop-man-cowboy-hat-tackles-him-mexico-5152558/. Accessed: 2019-06-12.

[37] *Google Chrome*. https://www.google.com/intl/sv/chrome/. Accessed: 2019-06-15.

[38] *Fatkun Batch*. https://chrome.google.com/webstore/detail/fatkun-batch-download-ima/nnjjahlikiabnchcpehcpkdeckfgnohf?hl=sv. Accessed: 2019-06-15.

[39] *LabelImg*. https://github.com/tzutalin/labelImg. Accessed: 2019-06-15.

[40] *TFRecord*. https://www.tensorflow.org/tutorials/load_data/tf_records. Accessed: 2019-07-11.

[41] *COCO Dataset*. http://cocodataset.org/. Accessed: 2019-07-21.

[42] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation". In: *CoRR* abs/1801.04381 (2018).

[43] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. "Focal Loss for Dense Object Detection". In: *CoRR* abs/1708.02002 (2017).

[44] *Tensorboard*. https://www.tensorflow.org/guide/summaries_and_tensorboard. Accessed: 2019-07-18.

[45] *mAP (mean Average Precision) for Object Detection*. https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173. Accessed: 2019-09-06.

[46] *COCO Detection Evalution*. http://cocodataset.org/#detection-eval. Accessed: 2019-08-04.

[47] *AXIS P1367 Network Camera*. https://www.axis.com/products/axis-p1367. Accessed: 2019-07-26.

[48] Rumen Kyusakov, Jens Eliasson, and Jerker Delsing. "Efficient structured data processing for web service enabled shop floor devices". In: *2011 IEEE International Symposium on Industrial Electronics*. IEEE. 2011.

[49] Salil Prabhakar, Sharath Pankanti, and Anil K Jain. "Biometric recognition: Security and privacy concerns". In: *IEEE security & privacy* 2 (2003).

[50] *GDPR*. https://gdpr-info.eu/. Accessed: 2019-08-30.

[51] Joy Buolamwini and Timnit Gebru. "Gender shades: Intersectional accuracy disparities in commercial gender classification". In: *Conference on fairness, accountability and transparency*. 2018.