Bachelor's thesis

Degree Programme in Business Information Technology

2019

Tommi Tuomola

APPLYING COMPUTER VISION TO TAILGATING DETECTION

– Case: Kupittaa Sports Hall

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology

2019 | 34 pages, 6 pages in appendices

Tommi Tuomola

APPLYING COMPUTER VISION TO TAILGATING DETECTION

- Case: Kupittaa Sports Hall

This thesis is related to a co-operation project between Turku University of Applied Sciences, the City of Turku and Fidera Ltd. The aim of the project was to develop an information system that collects usage information from the Kupittaa Sports Hall. As a part of this project, the thesis researched the possibility to apply computer vision to tailgating detection at the building entry points and evaluated which object detection algorithm best fits the use case. The thesis was commissioned by Fidera Ltd.

Neural networks and Python's OpenCV library were applied in handling the security camera footage. For tailgating detection, the most important aspect is to identify the human shapes from the images. As a result two object detection algorithms were compared in terms of performance and recognition confidence.

In the thesis Google Mobilenet SSD and YOLOv3 were used as object detection algorithms. Clear differences were recognized in both performance and recognition confidence of the algorithms. YOLOv3 reported over 90% confidence levels with the hardware used for testing. Google Mobilenet SSD reported values less than 60%. Performancewise Google Mobilenet SSD was able to handle almost three times the amount of images the YOLOv3 was capable of in the same time frame.

The results show that in the case of Kupittaa Sports Hall, the Google Mobilenet SSD gives a better outcome with the available hardware.

KEYWORDS:

Machine learning, computer vision, neural networks, pattern recognition, tailgating

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittely

2019 | 34 sivua, 6 liitesivua

Tommi Tuomola

TIETOKONENÄÖN SOVELTAMINEN TAILGATING-TAPAHTUMIEN TUNNISTAMISESSA

- Case: Kupittaan urheiluhalli

Opinnäytetyö liittyy Turun ammattikorkeakoulun, Turun kaupungin ja Fidera Oy:n yhteistyöhankkeessa toteutettavaan tietojärjestelmään, jonka päätavoitteena on kerätä tietoa Kupittaan urheiluhallin eri suorituspaikkojen ja tilojen käyttöasteesta. Osana tätä tietojärjestelmäkehitystä tutkittiin mahdollisuutta tunnistaa rakennuksen sisäänkäyntien yhteydessä tailgating-tapahtumia tietokonenäön avulla ja selvitettiin tietojärjestelmään parhaiten sopiva hahmontunnistusalgoritmi. Opinnäytetyö toteutettiin Fidera Oy:n toimeksiannosta.

Työssä sovellettiin neuroverkkoja turvakamerakuvan käsittelyssä Pythonin OpenCV -kirjaston avulla. Tapahtumien tunnistamisessa olennaista oli tunnistaa ihmishahmot kuvasta. Työn tuloksena vertailtiin kahta eri hahmontunnistukseen soveltuvaa koneoppimisalgoritmia tehokkuuden ja luotettavuuden osalta Kupittaan urheiluhallin viitekehyksessä.

Opinnäytetyössä käytettiin Google Mobilenet SSD- ja YOLOv3-hahmontunnistusalgoritmeja. Algoritmien suoritus- ja tunnistustehossa havaittiin selkeitä eroja. YOLOv3 ilmoitti testikokoonpanolla yli 90%:n tunnistusvarmuuksia. Vastaavasti Google Mobilenet SSD:n ilmoittama tunnistusvarmuus oli alle 60%. Suoritustehossa Google Mobilenet SSD pystyi käsittelemään noin kolminkertaisen määrän kuva-aineistoa samassa ajassa YOLOv3-algoritmiin verrattuna.

Tulokset osoittivat, että Kupittaan urheiluhallin tapauksessa Google Mobilenet SSD -algoritmi antaa käytettävissä olevalla laitteistolla paremman lopputuloksen.

ASIASANAT:

Koneoppiminen, tietokonenäkö, neuroverkot, hahmontunnistus

CONTENT

APPENDICES

FIGURES

TABLES

LIST OF ABBREVIATIONS (OR) SYMBOLS

AI	Artificial intelligence
CPU	Central processing unit
CyberLab	Turku University of Applied Sciences Information Security Laboratory
FPS	Frames per second
GPU	Graphics processing unit
NCS	Intel Neural Compute Stick
OpenCV	Open Source Computer Vision
SSD	Single shot multibox detector

1 INTRODUCTION

Sight is one of the five traditionally recognized senses that humans have. Sight is used to interpret the surrounding environment. The keyword for this thesis here is interpreting. Without going into details, sight allows us to recognize different objects as well as perceive motion and depth. According to Hermann von Helmholtz the human eyesight is optically rather poor. To compensate this the human neural system is believed to use Bayesian inference to interpret the statistical incomplete information gained via the visual system. (Wandell 1995, 7, 336.) As these statistical methods are well understood, they can be used to mimic the behavior of the human vision in artificially processing images with computers.

One possible application for computer vision is the tailgating event detection. While security guards could be placed at all entry points of a building to achieve this by visual detection, that would be very expensive for any sizable building. If it was possible to detect unauthorized entries automatically through computer vision, the guards could be alarmed to the gate when needed.

This thesis researches the question whether tailgating event detection is feasible by using computer vision and if it is which one of the available object detection algorithms should be used. In this thesis the choice of algorithms is limited to two commonly used algorithms, MobileNetV2 SSD and YOLOv3. Comparative research is done between these two candidates. The used research method is case study.

The subject is approached by first introducing artificial intelligence, machine learning and neural networks at a general level. Then the knowledge about neural networks is applied to object detection in computer vision context. The algorithm candidates are introduced as is the OpenCV library which will be used for running these algorithms. In the empirical part, the Kupittaa Sports Hall use case and the application developed for OpenCV are presented.

2 ARTIFICIAL INTELLIGENCE, MACHINE LEARNING AND NEURAL NETWORKS

In this chapter we'll discuss artificial intelligence, machine learning and neural networks. These concepts are closely related to each other as shown in Figure 1. Artificial intelligence is an umbrella term and will be briefly discussed before the more detailed descriptions are given for machine learning and neural networks.

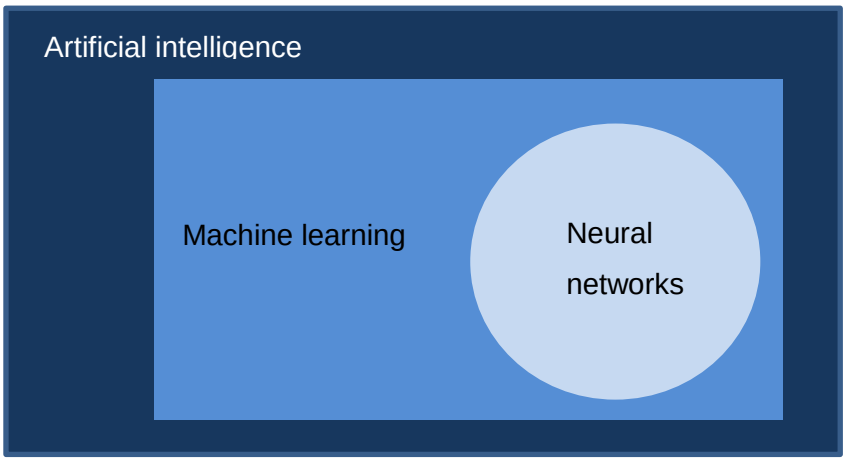

Figure 1. Relation between artificial intelligence, machine learning and neural networks.

Artificial intelligence (AI) is a part of computer science that considers machines in relation to human-like decision making. The artificial intelligence research has been an on-going academic discipline since late 1950's, however its popularity has been rising in recent years because of development in computer processing power. Artificial intelligence can be defined as an autonomous rational agent acting on an observed environment in order to achieve a preset goal. (Poole et al. 1998, 1.)

Nowadays AI research offers applications for problems in numerous different scientific disciplines: Economics, robotics, healthcare and transportation just to name a few. In this thesis we are applying a specific branch of AI called pattern recognition to automatically identify images containing humans from a stream of images.

By our definition, AI is an algorithm that given a set of observations outputs a decision. In this sense we can think of our data as a set of observations of an environment. AI maps the data to an output representing the decision we have set as the AI's goal in this situation.

2.1 Machine learning

In this thesis we are interested in a very specific subset of the AI. We want to identify human forms from a video stream. To achieve this, we handle the video stream as a stream of images and search each of those for certain kinds of patterns. That is, we have a set of observations (the data included in the images) and we need to classify each image according to whether it includes or does not include a pattern representing a human. In pattern recognition this is exactly what we are attempting to do. (Bishop 2006, vii, 1-3.) We are trying to find specific regularities, i.e. patterns, in the data. As a tool to run our pattern recognition and to classify our images we use machine learning.

Machine learning has a very long history and its research begun almost simultaneously with AI research in the late 1950's. In 1952 a computer program was taught to play checkers. (Mayo et al. 2018.) This can be seen as one of the first steps in this field. We have come quite far from those days and are now applying machine learning for various very challenging tasks. For example, machine learning is applied in stroke research in healthcare (Jiang et al. 2017).

It can be difficult to give an exact general definition for what we mean by machine learning. In this thesis we'll take the popular approach given by Mitchell in 1997: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" (Mitchell 1997, 2). We'll explain what we mean by tasks, performance measure and experience in the following paragraphs while going through machine learning related concepts.

By a *task* T we mean something that we are trying to achieve with machine learning. Learning is a prerequisite for the ability to perform the task. In our particular case, classifying an image according to if it has a human form in it or not is the task we want to accomplish. To accomplish this we first need to have an algorithm that has been taught how to classify the images. In order to teach our algorithm, we give it several *examples*. These examples consist of a set of *features* and in case of *supervised learning* also the wanted output. (Kotsiantis et al. 2007, 3-24.) In our pattern recognition example we could have a number of images with information about whether there's a human present in the image. In this case the features would be the pixel colour values of each image in the

example set. This process can be thought of as describing the machine learning task to the algorithm by providing the set of example data.

To evaluate how well our machine learning algorithm performs its given task T, we need a *performance measure* P (Mitchell 1997, 2-6). For our image classification case, it feels natural to measure the performance by considering the accuracy of the algorithm. That is, the performance of our algorithm on a given set is decided by how frequently we classify the images of the set correctly. Normally we are interested in how our algorithm performs on datasets which it has not been trained on. That is, we want our algorithm to be able to classify data outside the task description. Thus, to measure the accuracy we need a set of data for which wanted output we are familiar with, but which is not a subset of the data used for training our algorithm. A common practice is to only use a part of our known data to train the algorithm and save the remaining part to test the algorithms performance as shown in Figure 2. (Salian 2018)

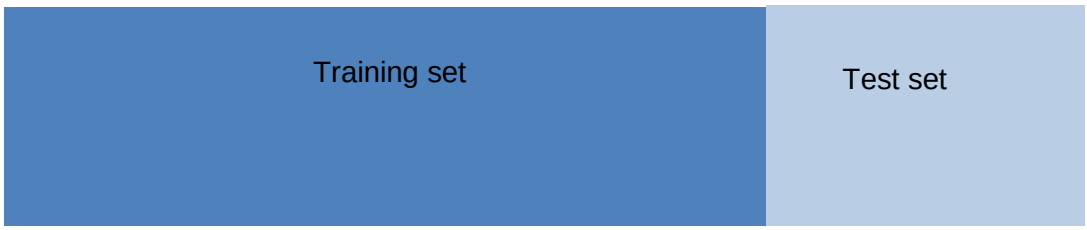

Figure 2. Splitting the dataset into training and test sets.

In order to learn, a machine learning algorithm needs to be trained by allowing it to experience a *dataset*. This dataset is a collection of many examples consisting of a set number of features each. Depending on what kind of *dataset* the algorithm is trained on, the learning can be categorized into roughly three categories: unsupervised, semi-supervised, supervised and reinforcement learning (Fumo 2017). In this thesis we are interested in supervised learning which directly applies to our case of categorizing images. In supervised learning the machine learning algorithm is trained with a dataset of input-output pairs so that each input is mapped onto a specific output.

There's a large amount of supervised learning algorithms available. Some of the more popular ones are *support vector machines* and *decision trees*. These traditional algorithms perform well for as long as the number of features in the dataset is kept moderate. (Hakala 2018, 4-7.) However as we need to categorize images with thousands

of pixels, each pixel with three colour values, these traditional algorithms become unwieldy.

Performance of a machine learning algorithm is often dependant on the amount of data we are able to use for its training. To mitigate this problem we are sometimes able to use *dataset augmentation*. The idea is to generate additional data based on our original training dataset. This can be especially effective in object recognition as the images have very high dimensionality. For example, It's often possible to make small rotations or translations to the image without changing the wanted output. Dataset augmentation is a method that helps us widen our dataset. (Hakala 2018, 14-15.) This does not help with the high dimensionality problem. Deep learning and especially neural networks as its application offers a robust solution to mitigate the challenges of handling high dimensional datasets.

2.2 Neural networks

Neural networks is a crucial machine learning algorithm used especially in supervised learning to efficiently mitigate the dimensionality problem. Its architecture is loosely based on the idea of how human brain works. In human brain there are billions of neurons connected by billions of synapses. The neurons take information from human sensory system as input and transmit it forward to the next neuron via the synapses. (Panchal 2018.) We'll get into much more detail of how the actual neural networks achieve this but for now this analogy suffices.

Neural network research begun around the same time period as artificial intelligence and machine learning research, back in the 1950's. Although some work on the mathematical basis was already done in the 1940's with threshold logic. The most well-known concept from those days is Frank Rosenblatt's perceptron which is a classic example of a binary classifier. In the early 1970's the neural network research considerably stagnated. Computers of that time didn't have enough processing power to run anything but the most simple networks. There were also theoretical challenges. For example, the used perceptrons were unable to produce exclusive-or circuits. (Wikipedia 2019)

Available computing power has risen considerably in the 2010's and progress in neural networks has been rapid. The use of graphics processing units capable of efficient parallel calculations to train and run the networks which consist of numerous simple

mathematical equations has allowed us to apply neural networks in a multitude of tasks. Possible applications vary considerably. Deep neural networks have been used to train a computer to beat even the best human players in Go (Roell 2017) and Chess, help with diagnosing cancer (Shell & Gregory 2017) and detecting botnet activity in cybersecurity context.

Neural networks approach the learning task in a multilayered manner so that layers form a network where the previous layers output works as following layers input. Inputs for each layer are modified with weight and bias parameters. Learning is achieved by adjusting these parameters. The neural networks consist of an input layer, an output layer and a number of hidden layers in between as shown in Figure 3.

The depth of the neural network is defined as the total number of hidden layers in the network plus one. A neural network is said to be fully connected if each neuron on each layer affects each neuron on the following layer. That is, the associated weights are non-zero throughout. An example of a fully connected network is shown in Figure 3. Here the colors are used to show the connections between neurons on different layers. In real cases the number of layers and especially neurons can be considerably higher. For example, for images the number of input neurons commonly is height in pixels * width in pixels * 3.

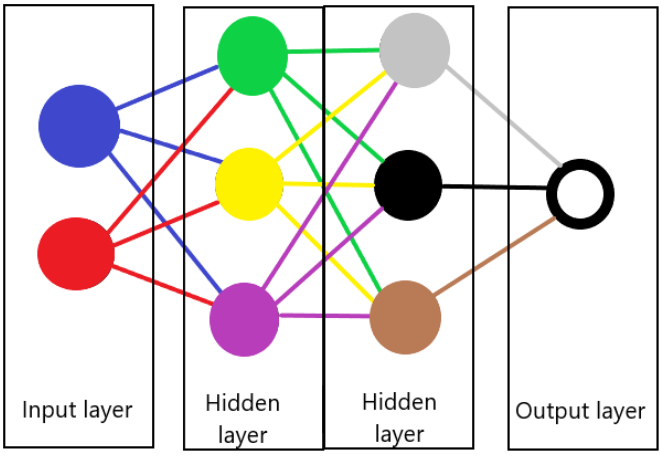

Figure 3. Neural network with two hidden layers.

The convolutional neural networks are a special case of neural networks designed for handling the type of data associated with images. The main problem with images and deep neural networks is that the number of weights we need gets very large very fast.

Convolutional neural networks use the fact that the input is an image to their advantage. Without going into any details, the idea of the convolution is to let the image points surroundings affect the neural network. For example, if there are blue pixels all around a red pixel, maybe it means something different to having green pixels all around. This kind of deduction is achieved by choosing a matrix filter (sometimes called kernel in image processing) and calculating the convolution between that filter and the matrix representing the image as shown in Figure 4. As a result we get a matrix feature map representing the image pixels and their associated surroundings. The dimensionality of the feature map depends on the chosen kernel and stride. In our example in Figure 4 the stride was one, so we moved the filter one element at a time starting from (0,0) by placing the middle of the filter there. (Karpathy 2018)

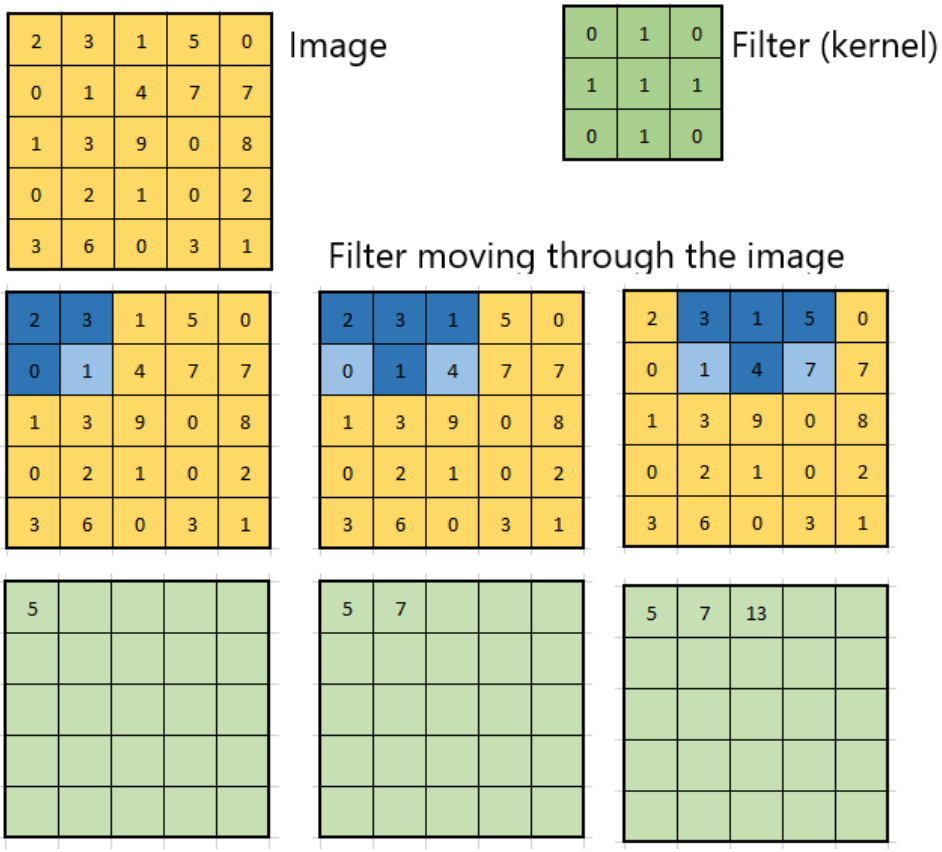

Figure 4. Convolutional neural network filter moving through image.

In addition convolutional neural networks use activation functions in the same way as regular neural networks to ensure non-linearity of the network and hence enabling learning. Another way the convolutional neural networks affect the dimensionalities of

the image matrix is the pooling process where the image is subsampled as in Figure 5. (Karpathy 2018)

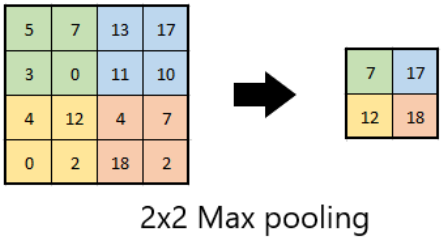

Figure 5. Example of pooling by choosing the maximum element

3 OBJECT DETECTION

Object detection, in computer vision context, is defined as an ability to find specific objects of interest in an image. Once found the object detection algorithm determines an area of the image where these object lie. This is different to classification in the sense that classification tells us something about the image as a whole whereas object detection algorithms recognize objects in the image, classify those objects and present their location. (Hollemans 2018a.)

For example, let’s say we have an image of a person holding a book. A classifier algorithm would be able to give a numerical value that indicates the likelihood of the image being about the book or about the person. That is, classifier would produce a probability distribution of the classes as an output. Object detection algorithm, on the other hand, would indicate if there are representatives of these classes in the image and if so where they are located.

Object location is commonly determined by a bounding box as shown in Figure 6. That is, the model determines coordinates within the image for a rectangular area that may contain an object of interest. As the models aren't able to recognize objects at 100% certainty, they associate with each bounding box the confidence level for an object residing within that bounding box. If the model is capable of recognizing multiple classes of objects, the detected object’s class is also associated with the bounding box. (Hollemans 2018a.)

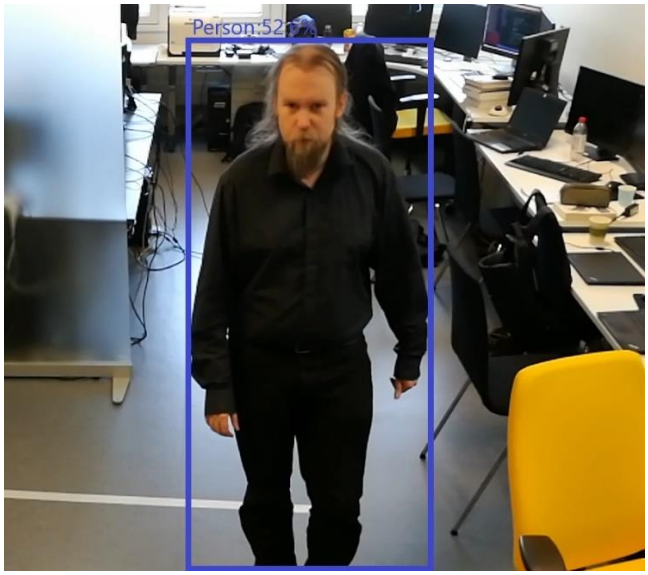

Figure 6. An image with a bounding box for the detected person object.

There's a multitude of different kinds of object detection algorithms. In this thesis we will concentrate on the single shot object detectors (SSD) and especially compare the Google MobileNetV2 SSD model to the YOLOv3 model.

SSDs are algorithms that are used in object detection and classification on images. The name comes from the idea of only running the convolutional network once on the input image in order to form a feature map. Then a small convolutional kernel is ran on the feature map to form the bounding boxes for the recognized objects as shown in Figure 7. (Liu et al. 2015.)

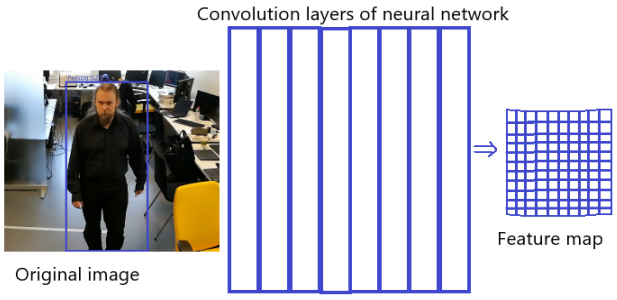

Figure 7. A SSD using a convolutional neural networks for finding a feature map from an image.

To determine the location of found objects the SSD models divide the image into a grid and use object detectors in the boxes determined by the grid. This way the object detectors in SSD models can specialize in detecting the objects at certain areas of the image.

Image files are represented by multidimensional arrays but at this point it suffices to only consider the coordinates. So, we have a two dimensional array of image coordinates. The models are often tuned for specific pixel resolutions. Thus we need to resize the image before running the algorithm. If the resolution is too high, we can reduce it. If the resolution is too low, we can add empty pixels to the sides without fear of the algorithm detecting objects there. Basically, we simply pad the matrix representing the RGB values of the image with null values. A little more problematic is the fact that the neural networks that the algorithms use work on images with equal width and height. The images we want to detect objects from aren't always squares. When resizing such an image we need to be careful not to change its aspect ratio so much that it would affect the neural network. (Hollemans 2018a.)

SSDs run images through a constant number of feature extractor layers and object detection layers. The actual number depending on the model in question. The idea here is to end up with a feature map with considerably smaller resolution than our original image. For example, YOLOv2 uses 416x416 pixel images as input and ends up with a 13x13 grid feature map. For each cell in the grid the SSD has five independent object detectors. The object detectors are needed for placing the bounding box around the object as in Figure 8. (Hollemans 2018a.)

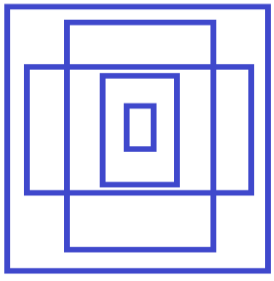

Figure 8. Representation of different ways to place the bounding box within a grid cell.

While the grid is used to let our detectors only concentrate on certain locations in the image for object detection, the preset bounding box forms in Figure 8 are used to limit the possible shape of our objects. That is, our detected objects are shaped so that they fit into these bounding boxes. The five different object detectors for each grid cell all have their own unique bounding box.

The feature map outputs probabilities for all different classes in the model, the coordinates of the bounding box relative to the grid cell center and a confidence score. The bounding box defined in previous paragraph only gives an approximation of the objects size. This is scaled with the coordinate response from the feature map. That is, as coordinates we get the relative x and y coordinates of the bounding box within the grid cell and scaling factors for the boxes width and height. This way we don't limit ourselves to objects of certain sizes.

Getting reliable information about the different models can be a bit awkward because of the way the models are referenced in different sources. Sometimes SSD refers to the concept of a single shot multibox detector and sometimes it's a direct reference to the Google MobileNetV2 SSD model. There are also different versions of the MobileNet model. YOLO on the other hand can be a reference to any of the different versions of YOLO model. These models have significant differences, so figuring out which version is in question is important.

3.1 MobileNetV2

The paper on Google's MobileNetV2 was submitted January 13th in 2018 (Sandler et al. 2018). It introduces a new architecture for the MobileNet model. The MobileNetV2 base model consists of 17 bottleneck residual blocks and 1x1 convolution block. The function of these layers is to take an image as an input and output the features recognized from the input pixels to the following neural network. That is, MobileNet can be used as a feature extractor. (Hollemans 2018b.)

The name of the models main blocks comes from the fact that it has separate convolutional layers that first expand the input by expansion factor and after the depthwise convolution layers they project the tensor dimensions back to the original. (Hollemans 2018b.) Expansion factor is one of the configurable parameters of this model, its default value being 6. The idea is to keep the input and output tensors low-dimensional

but still do the actual filtering with a high-dimensional tensors. What happens to tensor dimensions inside the bottleneck residual blocks with expansion factor of 6 is shown in Figure 9.

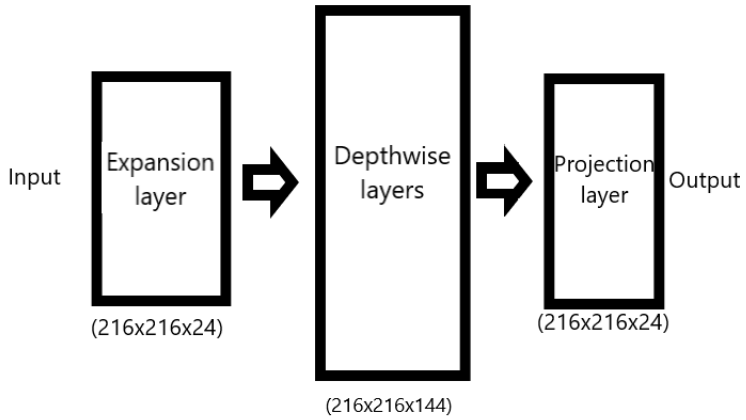

Figure 9. MobileNetV2 bottleneck residual block.

3.2 YOLOv3 model

The paper on YOLOv3 was submitted on 8th of April in 2018. In comparison to MobileNetV2 the YOLOv3 model uses a considerably deeper neural network for feature extraction. YOLOv3 moved to use Darknet-53 which has 53 convolutional layers. For detection YOLOv3 stacks another set of 53 network layers for a total of 106. (Redmon & Farhadi 2018)

YOLOv3 detects at three different scales. This is achieved by downsampling the input images dimensions. The first detection layer is the 82nd. At this point the image has been downsampled by 32. For our previous example image of dimensions 416x416 pixels this would result in 416/32 x 416/32 x 255 = 13 x 13 x 255 detection feature map. In addition to this YOLOv3 produces detection feature maps of 26 x 26 x 255 and 52 x 52 x 255. All in all, the image is downsampled by 32, 16 and 8. All these different layers and scales helps the model in detecting small objects more accurately, but makes it dependant on efficient concurrent computation. (Kathuria 2018)

3.3 OpenCV

Open Source Computer Vision Library (OpenCV) is a software library developed with the idea of offering a common platform for computer vision and related machine learning applications to the public. To achieve this goal the library is published under Berkeley Software Distribution license. (OpenCV team 2019a.)

OpenCV has been developed since its first alpha release in 2000 at the IEEE Conference on Computer Vision and Pattern Recognition until the latest OpenCV 4.1 release in April 2019. While the development was initiated by Intel Research Labs in 1999, there's currently a community of over 47000 developers working on OpenCV through a non-profit foundation OpenCV.org. (OpenCV team 2019a.)

The library itself was originally written in C but has since been converted to use C++ as its primary programming language for which new algorithms are developed. There are a number of APIs available for other programming languages. For our purposes the important one is the binding with Python language which we will be using in this thesis because of the ease of integrating the needed machine learning and security camera interfaces.

OpenCV library constitutes of a large number of algorithms which can be used in various applications ranging from simply manipulating images by rotating, resizing or recoloring them to tracking motion or identifying objects in a video stream. These algorithms have been used in various applications to achieve driver-less car navigation, medical image analysis and automated surveillance just to name a few.

OpenCV has a specific deep neural networks module which can be used to run object detection with different neural network models. We'll be applying both YOLO and Caffe model support to run our object detection algorithms with pre-trained weights in chapter 4.2.

4 TAILGATING USE CASE

The Kupittaa Sports Hall offers track and field facilities as well as the possibility to use gyms and saunas for the interested clubs and individuals. Its main user groups are athletics, rowing and canoeing, orienteering, ring gymnastics and triathlon clubs based in Turku. (City of Turku 2019)

Turku University of Applied Sciences, the city of Turku and Fidera Ltd are working in co-operation to produce an intelligent information system that monitors the usage level of the Kupittaa Sports Hall. The city is interested in how well the sports hall is occupied and if there are specific times when the hall is used less. In relation to keeping track of the number of people using the hall at any given time, the question of tailgating arose.

4.1 The Kupittaa Sports Hall

Currently the city has information about the users who authenticate themselves at the gate with a badge. We are interested in the accuracy of this information. The assumption is that the real number of users is not lower than the amount of badge authentications at the gate. That is, the accuracy is determined by possibly unauthorized people entering the building without authenticating. One way of entering like this is to follow an authorized person through the gate. This is called tailgating. We will limit our focus to recognizing tailgating events at the entrance most commonly used at the Kupittaa Sports Hall.

The physical gate at the entrance, as shown in Figure 10, is there to prevent unauthorized access. It also gives a level of security against tailgating. Now, in order to detect tailgating events, as per our definition, it suffices for us to detect situations when there are multiple persons entering the gate area simultaneously.

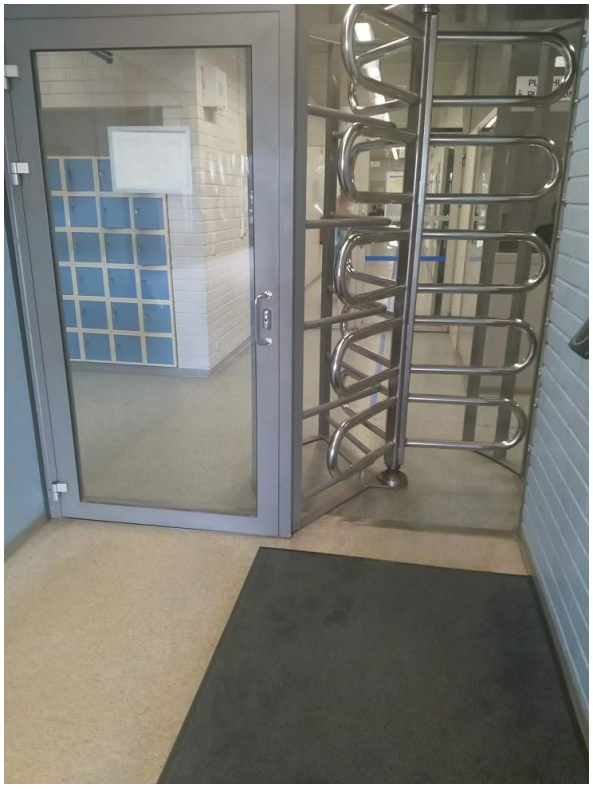

Figure 10. Kupittaa Sports Hall entrance.

Detecting multiple people at the gate depends on actually being able to define the gate area somehow in the image. There are different ways to approach this issue. The most direct way to achieve this would be to limit the area of the image we run through the object detecting neural network model by cropping everything outside the gate area off the image. Another possible approach is to create virtual tripwires in the image around the gate and run object detection between them. The tripwire approach has the benefit of giving enough space around the gate for motion detection.

4.2 OpenCV Application

As there, for obvious reasons, was no access to the video stream from the security cameras at the sports hall, simulations were ran on a different security camera at Turku University of Applied Sciences CyberLab. For this purpose a Python application, which used OpenCV to apply different neural network models on the video stream, was developed. Source code of an application running the YOLOv3 model for object detection is given in Appendix 1.

The video stream processing unit in Kupittaa Sports Hall has limited processing power. As is shown in 4.3, analyzing the image with the neural network is the most time consuming part of the process. The time used for running the neural networks can be minimized by a hybrid approach of first detecting motion and only then running object detection. The motion detection is also the reason why there is a need for a bit of extra space around the gate for the tripwire approach. While we still need to run the neural networks for object detection, we will not need to place a huge processor load on the hardware constantly with the hybrid approach.

Neural networks are often ran on powerful GPUs. This is because of the relatively simple computations that can be ran concurrently. However on edge devices such as the security camera setup in Kupittaa Sports Hall, this isn't a feasible option. Hence, another optimization that was made to ease the difficulty of running the neural network on potentially weak CPU, was to separate reading the frames from the video stream from the part of the application that runs the neural network. This was done by reading the video stream in a separate thread and having the main thread, which also ran the neural network, always request the newest frame from the video stream in the beginning of its neural network loop. Source code for the written threaded approach is given in Appendix 2.

4.3 Simulations

Simulations were ran using a desktop computer (Intel Core i7-4790 CPU @ 3.60GHz) in CyberLab. HikVision DS-2CD6362F-I security camera was used to simulate the Kupittaa Sports Hall camera. The maximum framerate of the HikVision camera is 25 frames per second (FPS) with resolutions ranging from 1280x1280 to 3027x2048. Camera stream was connected using the Real Time Streaming Protocol. Simulations were ran on an Ubuntu 18.04 virtual machine using Python 3.7 with OpenCV version 4.0.1.

As the result of our simulations we stored FPS values together with collected minimum and maximum confidence levels for person detection. For these values the average and the average for absolute deviation were calculated. To mitigate the statistical effects the simulations were ran for 600 seconds each time. The video stream was taken in CyberLab and as such there was various amount of movement and recognized objects in different simulations but there was no detected effect of this in the results. The Python

code used for running the different models was very similar and thus only YOLOv3 version is shown here in Appendix 1. The simulation results are presented in Table 1.

Table 1. The simulation results

	MobileNet SSD			**YOLOv3**		
		Confidence			**Confidence**	
	FPS	**min**	**max**	**FPS**	**min**	**max**
	3,77	0,31	0,60	0,86	0,65	0,99
	5,31	0,30	0,35	0,85	0,58	0,97
	5,50	0,30	0,41	0,85	0,54	0,98
	5,73	0,31	0,52	0,86	0,61	0,98
	5,60	0,32	0,45	0,86	0,75	0,98
	5,80	0,30	0,43	0,84	0,51	0,99
	5,72	0,33	0,48	0,84	0,62	0,97
	5,28	0,31	0,49	0,85	0,53	0,99
	5,28	0,33	0,57	0,84	0,57	0,98
	5,72	0,32	0,39	0,86	0,60	0,98
Average	5,37	0,31	0,47	0,85	0,60	0,98
Average deviation	0,37	0,01	0,06	0,01	0,05	0,01

The results give a clear implication of the differences of the models. Higher FPS indicates better performance as the application only shows video frames that have gone through the object detection neural network. MobileNet SSD performs with considerably higher FPS but its reported confidence values are lower. The reported confidence values are not directly comparable as they measure the detection confidence from the algorithms own point of view. The reported minimum confidence values are affected by the detection thresholds set in simulation application. The used minimum thresholds were 0.30 for MobileNet SSD and 0.50 for YOLOv3.

To ensure efficiency Linux application *htop* was used to check that the processor load was as high as possible while running the application. OpenCV, unlike the rest of the simulation application, uses all available processor cores to run the neural network. Because the processor load was high during runtime this was a clear indication that the it was indeed the neural networks we spent most of the processor time on. This was verified by running the Python profiler cProfile to check the relative time used in different functions of the application. The cProfile reported values of over 95% processor time

spent running the neural networks depending on the total runtime of the application. The rest of the processor time was spent on start-up processes like importing libraries and connecting to the video stream. This further verifies that the choice of chosen object detection algorithm is relevant.

5 CONCLUSION

The aim of the thesis was to compare MobileNet SSD model with YOLOv3 model and consider which one of these neural network models would better fit a use case of tailgating event detection with computer vision in Kupittaa Sports Hall. As it wasn't possible to connect to a real video stream from the sports hall, the neural network model evaluation was carried out by simulating the video stream from the security camera in CyberLab.

With the hardware configuration used in simulations the higher confidence YOLOv3 model seems problematic to use because of the low performance. For tailgating detection it would be important to get more than one or two frames per person entering the building. MobileNet SSD's higher FPS values let's us analyze more frames while the person is walking through the gate. This results in better probability of recognizing tailgating events. YOLOv3 simply doesn't allow us to process through enough frames to be useful even if it recognizes persons with high confidence on the frames it processes.

However, there are better choices for hardware configuration available. For example, one option would be to use an Intel Neural Compute Stick (NCS) or similar to help with the performance issues of running the neural networks on CPU. While NCS cannot compete with running the neural network on GPU, it's still better than running the network on old desktop processor and relatively inexpensive. (Intel 2019) The main point here is that if reasonable levels of FPS can be reached with YOLOv3, it can be used for the better confidence levels.

Another thing to consider is the new version of OpenCV that was released on 8.4.2019. The new version has performance improvements and better support for the NCS. This is exactly what would have been needed but it was decided to be out of scope for this thesis because of time constraints. (OpenCV team 2019b) In addition, from a strictly performance point of view, running OpenCV with C++ rather than Python would give a performance boost.

Because of the simulation setup in CyberLab it was not possible to vary camera placement. So the effect of camera position was ruled out of the scope of the thesis. It is however a significant factor in the real case and should be carefully considered. The dependency on camera placement as well as the hardware configuration puts some

doubt on the reliability of the results. The presented neural network models can be used but there's still open issues with the actual on-site hardware configuration before final decisions on the used models can be made. More information is needed.

In conclusion, computer vision can be used for detecting tailgating events. Furthermore the choice of neural networking model depends highly on hardware configuration of the use case. On lower end processing power the MobileNet SSD gives more reliable results but with more processing power it's possible to use YOLOv3 which in the simulations gave better confidence levels and object detection.

The thesis work was quite challenging. The subject matter is vast and there has been research done in both computer vision and neural networks for decades. Thus a good amount of time was spent reading through articles and related theory. The amount of material used in the thesis had to be limited. This was very difficult as there were so many aspects of the subject matter that would have been interesting to work with. Then there was the question of time constraints. The work needed to be completed in a matter of months. In a co-operation project this proved to be challenging. All in all, it was an excellent project to learn computer vision and neural networks in.

REFERENCES

Bishop, C. 2006. Pattern Recognition and Machine Learning. New York: Springer-Verlag.

City of Turku, 2019. Kupittaa Sports Hall. Accessed 23.5.2019 https://www.turku.fi/en/culture-and-sports/sports/sports-facilities/indoor-sports-and-exercise-halls/kupittaa-sports-hall

Fumo, D. 2017. Types of Machine Learning Algorithms You Should Know. Accessed 23.5.2019 https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861

Hakala, J. 2018. Object Recognition for Maritime Application Using Deep Neural Networks. Master of Science Thesis. Master's Degree Programme in Information Technology. Tampere: Tampere University of Technology.

Hollemans, M. 2018a. One-shot object detection. Accessed 23.5.2019 https://machinethink.net/blog/object-detection/

Hollemans, M. 2018b. MobileNet version 2. Accessed 23.5.2019 https://machinethink.net/blog/mobilenet-v2/

Intel, 2019. Introducing the Intel Neural Compute Stick 2 (Intel NCS 2). Accessed 23.5.2019 https://software.intel.com/en-us/neural-compute-stick

Jiang, F;Jiang, Y.;Zhi, H.;Dong, Y.;Li, H.;Ma, S;Wang, Y;Dong, Q;Shen, H & Wang, Y 2017. Artificial intelligence in healthcare: past, present and future. Accessed 23.5.2019 https://svn.bmj.com/content/svnbmj/2/4/230.full.pdf

Karpathy, A. 2018. Convolutional Neural Networks (CNNs / ConvNets). Accessed 23.5.2019 https://github.com/cs231n/cs231n.github.io/blob/master/convolutional-networks.md

Kathuria, A. 2018. What's new in YOLO v3? Accessed 23.5.2019 https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b

Kotsiantis, S.B.;Zaharakis, I. & Pintelas, P. 2007. Supervised machine learning: A review of classification techniques, Emerging artificial intelligence applications in computer engineering, Vol. 160.

Liu, W.;Anguelov, D.;Erhan, D.;Szegedy, C.;Reed, C;Fu, C & Berg, A. 2015. Single Shot MultiBox Detector. Accessed 23.5.2019 https://arxiv.org/abs/1512.02325

Mayo, H.;Punchihewa, H.;Emile, J. & Morrison, J. 2018. History of machine learning. Accessed 23.5.2019 https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html

Mitchell, T. 1997. Machine Learning. New York:McGraw-Hill.

OpenCV team 2019a. About. Accessed 23.5.2019 https://opencv.org/about/

OpenCV team 2019b. OpenCV Change Logs. Accessed 23.5.2019 https://github.com/opencv/opencv/wiki/ChangeLog#version410

Panchal, S. 2018. Artificial Neural Networks – Mapping the Human Brain. Accessed 23.5.2019 https://medium.com/predict/artificial-neural-networks-mapping-the-human-brain-2e0bd4a93160

Poole, D.;Mackworth, A. & Goebel, R. 1998. Computational Intelligence: A Logical Approach. New York: Oxford University Press.

Redmon, J. & Farhadi, A. 2018. YOLOv3: An Incremental Improvement. Accessed 23.5.2019 https://pjreddie.com/media/files/papers/YOLOv3.pdf

Roell, J. 2017. Why AlphaGo is a bigger game changer for Artificial Intelligence than many realize. Accessed 23.5.2019 https://medium.com/@roelljr/why-alpha-go-is-a-bigger-game-changer-for-artificial-intelligence-than-many-realize-64b00f54a0

Salian, I. 2018. SuperVize Me: What's the Difference Between Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning? Accessed 23.5.2019 https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/

Sandler, M.;Howard, A.;Zhu, M.;Zhmoginov, A. & Chen, L. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. Google Inc. Accessed 23.5.2019 https://arxiv.org/pdf/1801.04381.pdf

Shell, J. & Gregory, W. 2017. Efficient Cancer Detection Using Multiple Neural Networks. IEEE journal of translational engineering in health and medicine. Vol. 5.

Wandell, B.A. 1995. Foundations of vision. Sunderland, Mass.: Sinauer Associates.

Wikipedia, 2019. Accessed 23.5.2019 https://en.wikipedia.org/wiki/Artificial_neural_network

Python example of an object detection with YOLOv3 model

```
import numpy
import argparse
import imutils
import time
import cv2
import os
from stream import TestStream
from imutils import rotate
from imutils.video import FPS

# Parse arguments from command line parameters.
ap = argparse.ArgumentParser()
ap.add_argument("-y", "--yolo", type=str, default="yolo",
                help="base path to YOLO directory")
ap.add_argument("-c", "--confidence", type=float, default=0.5,
                help="minimum confidence threshold for detection")
ap.add_argument("-t", "--threshold", type=float, default=0.3,
                help="threshold for non-maxima suppression")
args = vars(ap.parse_args())

# Load the object classifier label names and associate the classes
# with colours for the bounding box.
labelsFile = os.path.sep.join([args["yolo"], "coco.names"])
labels = open(labelsFile).read().strip().split("\n")
numpy.random.seed(100)
colours = numpy.random.randint(0, 255, size=(len(labels), 3),
                               dtype="uint8")

# Define location of cfg and weight files for our model.
weightsPath = os.path.sep.join([args["yolo"], "yolov3.weights"])
configPath = os.path.sep.join([args["yolo"], "yolov3.cfg"])

# Load our network model with the given configs and determine names
# of the output layers.
neuralNetwork = cv2.dnn.readNetFromDarknet(configPath, weightsPath)
layerNames = neuralNetwork.getLayerNames()
layerNames = [layerNames[i[0] - 1]
              for i in neuralNetwork.getUnconnectedOutLayers()]

# Load the videostream and set dimensions to None.
videoStream = TestStream().start()
(H, W) = (None, None)
```

```
# Sleep for 2sec to make sure the stream connection is ok and start
# FPS count.
time.sleep(2.0)
fps = FPS().start()

# Initiate a fullscreen window for our stream.
cv2.namedWindow('Live', cv2.WND_PROP_FULLSCREEN)
cv2.setWindowProperty("Live", cv2.WND_PROP_FULLSCREEN,
                      cv2.WINDOW_FULLSCREEN)

# Loop until interrupted by pressing q.
while True:
    # On every pass we read the latest frame from the video stream.
    frame = videoStream.read()

    # This is only needed in our test case where we had to mount
    # the camera upside down on the wall. 180 degree rotation fixes
    # the issue.
    frame = rotate(frame, 180)
    fps.update()

    # If we are still missing frame dimensions, grab them. Should
    # only happen on first pass. Notice the order here.
    if W is None or H is None:
        (H, W) = frame.shape[:2]

    # Construct a blob from the input frame with our network models
    # settings and perform a forward pass of the neural network,
    # resulting in our bounding box coordinates and associated
    # confidence levels.
    blob = cv2.dnn.blobFromImage(
        frame, 1 / 255.0, (416, 416), swapRB=True, crop=False)
    neuralNetwork.setInput(blob)
    layerOutputs = neuralNetwork.forward(layerNames)

    boxes = []
    confidences = []
    classIDs = []

    # Loop through the output layers.
    for output in layerOutputs:
        # Loop through each of the detections on a given output.
        for detection in output:
            # Each detection has a list of classes and related
            # confidence level. We only want the one with the best
```

```
            # confidence level, ie the most likely class for this
            # detection.
            confidenceScores = detection[5:]
            classID = numpy.argmax(confidenceScores)
            confidence = confidenceScores[classID]

            # Make sure the best confidence is above our set
            # confidence threshold.
            if confidence > args["confidence"]:
                # Deduce the related bounding box coordinates in
                # the video frame. Scaling is needed here as our
                # camera resolution differs from model resolution.
                # We use top-left coordinates of the bouding box
                # to determine its location.
                boundingBox = detection[0:4] * numpy.array([W, H,
                                                            W, H])
                (boundingBoxCenterX, boundingBoxCenterY,
                 width, height) = boundingBox.astype("int")
                x = int(boundingBoxCenterX - (width / 2))
                y = int(boundingBoxCenterY - (height / 2))

                # Add the bounding box coordinates, confidence and
                # classID to the related data structures.
                boxes.append([x, y, int(width), int(height)])
                confidences.append(float(confidence))
                classIDs.append(classID)

    # We use non-maxima suppression to select most relevant found
    # objects.
    indices = cv2.dnn.NMSBoxes(boxes, confidences,
                               args["confidence"],
                                args["threshold"])

    # Ensure at least one detection exists.
    if len(indices) > 0:
        # Loop through the indices corresponding to relevant
        # bounding boxes.
        for i in indices.flatten():
            # Grab the bounding box coordinates and dimensions.
            (x, y) = (boxes[i][0], boxes[i][1])
            (w, h) = (boxes[i][2], boxes[i][3])

            # Draw the bounding box rectangle and label onto the
            # frame.
            colour = [int(c) for c in colours[classIDs[i]]]
            cv2.rectangle(frame, (x, y), (x + w, y + h), colour, 2)
            text = "{}: {:.4f}".format(labels[classIDs[i]],
```

```
                                        confidences[i])
            cv2.putText(frame, text, (x, y - 5),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, colour, 2)

    # Show the image on screen and check if abort key has been
    # pressed.
    cv2.imshow('Live', frame)
    k = cv2.waitKey(10) & 0xff
    if k == ord('q'):
        break

# Stop FPS counter and videostream, destroy the image window and
# print out gained FPS values.
fps.stop()
videoStream.stop()
cv2.destroyAllWindows()
print("FPS: elapsed time: {:.2f}".format(fps.elapsed()))
print("FPS: approx. FPS: {:.2f}".format(fps.fps()))
```

Python example of a threaded video stream frame reading

```
from threading import Thread
import cv2
import time

video_source = 'rtsp://user:pw@rtsp-address_of_security_camera'

class TestStream:
    def __init__(self, src=video_source, name="TestStream"):
        # initializing the stream
        self.stream = cv2.VideoCapture(video_source)
        time.sleep(2.0)
        (self.grabbed, self.frame) = self.stream.read()

        self.name = name
        self.stopped = False

    def start(self):
        # Start reading frames from the stream
        reading_thread = Thread(target=self.update, name=self.name,
                                args=())
        reading_thread.daemon = True
        reading_thread.start()
        print("Video stream: " + str(reading_thread.is_alive()))
        return self

    def update(self):
        # stay here until stopped flag is up
        while True:
            # if stopped flag is up, kill the thread
            if self.stopped:
                print("Shutting down video stream.")
                return
            # read the newest frame
            (self.grabbed, self.frame) = self.stream.read()

    def read(self):
        return self.frame

    def stop(self):
        self.stopped = True
```