IT UNIVERSITY OF COPENHAGEN

Exploring AI's Potential for Surveillance Camera Detection in Denmark

AUTHORS:
Claudio Sotillos Peceroso & Haidar Imad Al-Seelawi

CODE: KIREPRO1PE

Research Project
Claudio Sotillos Peceroso (clap@itu.dk) – MSc in Computer Science
Haidar Imad Al-Seelawi (haia@itu.dk) – MSc in Software Design

December 13, 2023
Supervisor:
Vedran Sekara

ABSTRACT

With the recent growth of various Deep Learning fields, we are witnessing more and more possible implementations of Deep Learning to solve various social concerns. In this research project specifically, we explore the application of Computer Vision techniques with the main objective of localizing and mapping the location of surveillance cameras on the streets of Copenhagen. We take advantage of the fact that a similar study has been successfully conducted in the United States, which we will use as a starting point. Here we show how a Computer Vision model trained in a different context (US) can be adapted to successfully perform the same task in another context (DK) by using transfer learning. We have been able to fine-tune the original model, obtaining a new model that shows higher prediction accuracy than the original model on the streets of Copenhagen, as well as creating a dataset of images of the Copenhagen facades that can be used for future studies. Our model correctly predicted 57% of the images while the US model predicted 52.5% of the images. In addition, it almost doubled the f1-score, meaning that it is making fewer mistakes in both falsely claiming the presence of cameras and missing images with cameras. Despite the evident improvement, we are aware that we can still achieve a more powerful model in terms of its predictive capacity, which we will continue to pursue in our future master's thesis.

CONTENTS

1 INTRODUCTION

In today's digital landscape, the widespread presence of surveillance cameras in public spaces has become a critical social concern that raises questions about privacy, transparency, and civil liberties.

This concern is particularly highlighted in the article *"30 to 100 meters: this is the distance at which cameras in shops and nightclubs can film on the street"* [1]. The expansion of the monitoring range in public spaces could raise surveillance levels in Denmark, claims *Birgitte Arent Eiriksson*, the Director of Justitia[1] [2]. Furthermore, she claims that it will have profound implications for our privacy, influencing our behavior in public space, shaping our conversations, and determining our social interactions [1].

With this in mind, this research project dives into an exploration, employing Computer Vision methodologies to unravel the details of surveillance on the streets of Denmark, starting with its capital, Copenhagen.

The main driving force for this research project is to eventually be able to answer the following questions:

- How many surveillance cameras are on the streets of Copenhagen and where are they located?
- Are citizens inadvertently subjected to excessive surveillance in public spaces?
- Do certain neighborhoods or demographic groups experience higher levels of scrutiny?
- Does the historical context of crime influence the density of surveillance?

However, to address these questions, we need to map the locations of cameras in Copenhagen.

[1] **Justitia:** Denmark's first independent legal policy institute, their aim is to strengthen the focus on and respect for basic principles of the rule of law and freedoms in the public.

Inspired by a similar study carried out in the United States (will be referred to as the project by Sheng et al.) entitled "*Surveilling Surveillance: Estimating the Prevalence of Surveillance Cameras with Street View Data*" [3], our project aims to adapt the already trained Computer Vision model by Sheng et al. using transfer learning and apply it to the streets of Copenhagen. Having a trained model will save us time and computational costs since training a robust object detection model from scratch requires a large amount of images and computational power.

To perform transfer learning, we still need a considerable amount of images. Therefore, one of our first objectives is to curate an extensive database of street-view images meticulously gathered from the facades of Copenhagen. This image database not only forms the foundation of our analysis but also stands as a valuable resource for future studies. Furthermore, we aim to develop a simplified pipeline to retrain the pre-existing model, to use it in our future master's thesis. The idea behind this pipeline is *to make the model capable of generalizing and detecting Danish surveillance cameras*. By fine-tuning[2] the model using carefully curated training data, we aspire to enhance its adaptability to the context of Danish urban spaces.

Throughout the report, we will refer to the model already trained by Sheng et al. as the **US Model** and to our fine-tuned model as the **DK Model**.

This research project focuses on the following questions:

- Can the *US Model* effectively identify cameras on Danish streets, and to what extent does it demonstrate proficiency in this task?
- Is the predictive capability of the model improved on Danish streets after the fine-tuning process?

Our work is organized into several key sections. *Section* 2 provides an overview of the work conducted by Sheng et al. and offers a technical review of the architecture **Faster R-CNN R**50 **FPN** proposed by them, which we will also make use of. Then, in *Section* 3 we detail how the images were gathered for our study. *Section* 4 is devoted to a detailed explanation of the setup we used to perform the model fine-tuning and evaluation, in addition to how we created the train/validation and testing datasets. Following this, *Section* 5 explains how we evaluated the *US Model* and how we fine-tuned it. The last sections present the results of the experiment and analysis of these followed by a Conclusion and Further work.

[2] **Fine-Tune:** Adapt a pre-trained machine learning model to a specific task or new scenario, leveraging its existing knowledge. It helps optimize performance for domain-specific applications without starting training from scratch.

2 BACKGROUND & MODEL TECHNICAL DETAILS

The paper of Sheng et al. aimed to provide a systematic and scalable method to estimate the prevalence and location of surveillance cameras in urban areas, as well as to determine the distribution of cameras and what factors influence the increased surveillance in given areas.

To this end, they collected data from 16 major cities, focusing on the 10 cities with the highest density (each with at least 500.000 residents) and six major cities in Asia and Europe (see Figure 1). They used a 3-step statistical estimation procedure to perform the estimation [3].

First, they collected a dataset of around 13.000 Google Street View Static images, some of which contained cameras and some of which did not. These images were meticulously labeled using segmentation masks. They managed to collect 861 images containing a total of 977 cameras, which were distributed to form the training, validation, and test sets. After that, they trained a model capable of segmenting cameras in the images, evaluating its accuracy by calculating the recall in a validation dataset [3].

City	Population	Area (sq. km)	Road length (km)
Los Angeles	3,793,000	1,213	21,095
New York City	8,175,000	783	16,362
Chicago	2,696,000	589	10,449
Philadelphia	1,526,000	347	6,759
Seattle	609,000	217	5,569
Milwaukee	595,000	248	4,899
Baltimore	621,000	209	3,746
Washington, D.C.	602,000	158	3,262
San Francisco	805,000	121	3,101
Boston	618,000	125	2,589
Tokyo	13,159,000	2,194	46,688
Bangkok	8,305,000	1,569	34,692
London	8,174,000	1,572	28,907
Seoul	9,630,000	605	14,748
Singapore	3,772,000	728	5,794
Paris	2,244,000	106	1,853

Figure 1: Table from Sheng et al.'s paper listing the 10 large U.S. cities and 6 other major cities along with their respective populations, the number of inhabitants per square kilometer, and road lengths [3].

Once they obtained a Computer Vision model with a good performance, the next step was to use it on a random sample of images. They sampled 100.000 points (coordinates) uniformly at random. For each of the coordinates, they queried the Google Street View Static API to get the 360° panorama. The computer vision models were run

on these images identifying 6.281 images that contained 6.469 cameras in total. All these images were manually labeled to eliminate false positives (detecting a camera when there isn't one) [3].

Finally, combining the location of the detected cameras with the geometry of the camera angles, the road network, and building footprints, they calculated an estimate of the sample's camera coverage of the road network. The road network data and building footprints were taken from OpenStreetMap [4].

Their study revealed a notable concentration of cameras in commercial, industrial, and mixed-use areas, as well as neighborhoods with higher non-white populations.

The Computer Vision model that they initially used was a combination of the DeepLab V3+ architecture with an EfficientNet-b3 backbone. Eventually, they decided to change this architecture to a *Faster Region-Based Convolutional Neural Network* (**R-**50**-FPN**) architecture as a suggestion by Turtiainen et al. [5]. Faster R-CNN (R-50-FPN) is more efficient and faster to train since it has been specifically designed for object detection tasks. Since the architecture used in our study is Faster R-CNN (R-50-FPN), we will focus on explaining it, starting with a previous explanation of its predecessor architectures.

2.1 FROM R-CNNS TO FASTER R-CNNS

Traditional object detection techniques are composed of 3 major steps. First, start by generating several region proposals for the input image. These region proposals are candidates that might have objects within them. The number of these regions is usually in the order of several thousands (i.e. 2.000 or more).

Then, from each region proposal, a fixed-length feature vector is extracted using various image descriptors such as the histogram of oriented gradients (HOG) which can be observed in Figure 2. Finally, the feature vector is used to assign its respective region proposal either to the background class or to one of the object classes.

2.1.1 R-CNNs

In 2014 the R-CNN architecture was published [7]. The innovation presented here is that the extraction of features is carried out by means of a *Convolutional neural network* (**CNN**). The pipeline that images followed for their object detection was to first generate thousands of

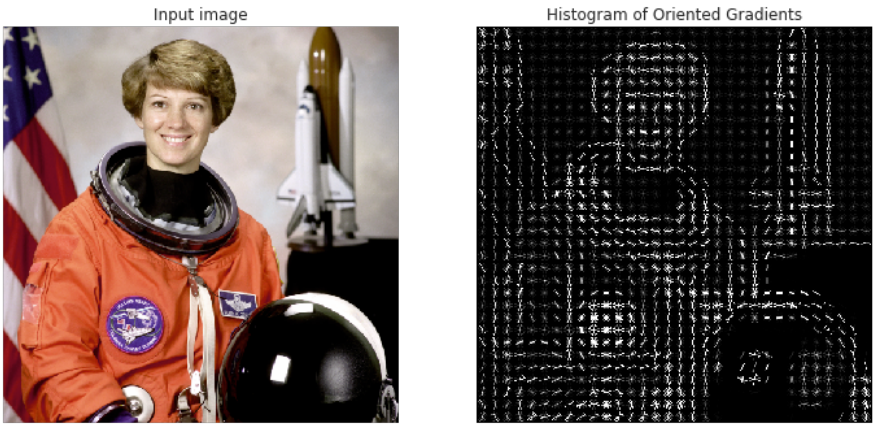

Figure 2: Original Image vs HOG of the image [6].

region proposals using the **Selective Search**[3] algorithm. Then, the second module consisted of a CNN which extracts the features from each region proposal in the form of a fixed-length feature vector (this CNN is commonly referred to as the Backbone of the R-CNN). Finally, the third module uses a pre-trained *Support vector Machine*[4] (**SVM**) which classifies the region into one of the possible classes or into "background". To have a clearer picture of how R-CNN function check Figure 3.

The performance of this new architecture compared to the architectures proposed so far made it state-of-the-art at the time. However, this architecture had several drawbacks. The biggest of all is that it is a very expensive architecture to use, both in terms of memory required as it cached all the feature vectors on disk and also in terms of time as it took a long time to train and to be evaluated mainly because of the slowness of the Selective Search algorithm.

Moreover, since it is a modular model, which consists of 3 independent components, it cannot be trained as a whole system (end-to-end). In order to make this architecture more efficient, Fast R-CNN was proposed in 2015, which addresses some of the previously mentioned problems.

2.1.2 Fast R-CNNs

This architecture closely resembles the R-CNN approach except for one key difference. The CNN is fed with the input image to generate a convolutional feature map, instead of being fed region proposals.

[3] **Selective Search:** Combines multiple low-level image segmentation techniques and groups similar segments into larger regions which are then merged based on similarity measures, creating a hierarchy of different-sized image segments.

[4] **SVM:** Machine learning algorithm used for classification and regression tasks. It works by finding the hyperplane that best separates data points of different classes in a high-dimensional space.

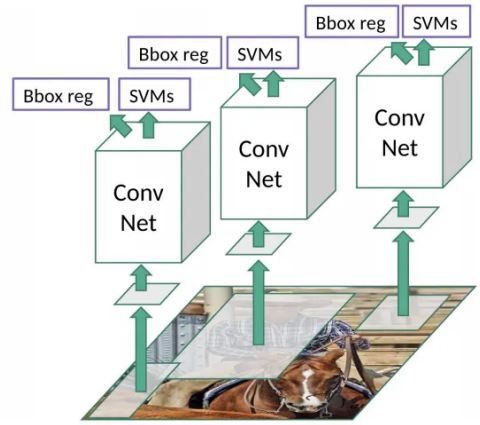

Figure 3: Overview of how a R-CNN Architecture performs regional object predictions [8].

Then, region proposals are extracted from the convolutional feature map via Selective Search, transformed into squares, and resized into fixed-size feature vectors via RoI Pooling to feed them into several *fully connected* (FC) layers[5] (check Figure 5 for understanding the concept of RoI pooling). In this scenario, these layers learn to map the extracted features to specific classes and bounding box regression values. The output of the last FC layer is fed at the same time into a *Softmax layer*[6] to anticipate the category of the proposed region and to a final FC layer used to predict the bounding boxes. Indeed, the percentage shown in the output bounding box is the probability (computed by the Softmax Layer) of being a given object. Check Figure 4 to have a clearer overview of how Fast R-CNN processes a given image.

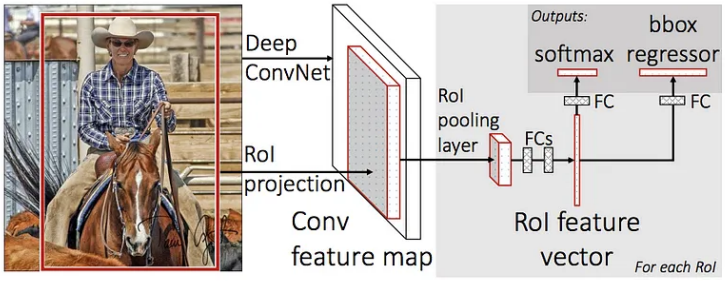

Figure 4: ***Fast R-CNN pipeline*** – Firstly, CNN extracts the Feature Map. Then, region proposals with Selective Search from the Feature Map. Finally, RoI pooling is applied over each region proposal generating feature vectors which will be fed to some FC layers to get the Class & Bounding Box predictions [7].

[5] **FC Layers:** A fully connected layer is a neural network layer where each neuron in the layer is connected to every neuron in the previous layer.

[6] **Softmax Layer:** Layer used for multi-class classification. It takes the output produced by the preceding layers and maps these into probabilities of each class, summing up to one.

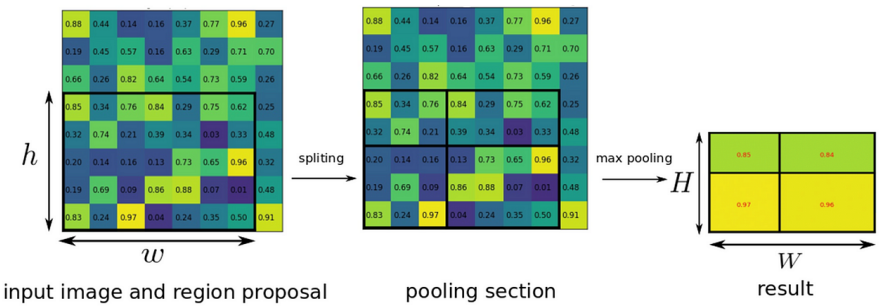

Figure 5: RoI pooling takes a region proposal from an input feature map and divides it into fixed-size bins. Each bin is independently max-pooled, resulting in a set of fixed-size sub-regions. These sub-region features are then flattened into a one-dimensional vector, which is fed into subsequent fully connected layers [9].

Despite the advantages of the Fast R-CNN model, this architecture still relies on Selective Search for the generation of region-proposals. So in order to tackle this drawback the Faster R-CNN architecture is proposed.

2.1.3 Faster R-CNNs

As already mentioned in 2.1.1, Selective Search is a slow and time-consuming process that affects the overall architecture performance. To get rid of this bottleneck, the Selective Search algorithm is changed by an object detection algorithm which lets the network learn the region proposals. It is a **Region Proposal Network (RPN)** that generates regions with variable scales and aspect ratios (width-to-height ratio), implementing what is known as *neural network attention*, telling the network where to focus [8].

The RPN shares its convolutional layers with the backbone network. This implies that both Networks can be unified into a single network which can be trained as a hole (end-to-end) and it has to be done only once. Also, since it can be fine-tuned for a specific detection task this allows to generate better region proposals than Selective Search does. This aspect is key in making the Network much faster and more efficient compared to previous architectures.

The feature map of the last Convolutional layer (shared with the Fast R-CNN) is processed using a sliding window (size NxN). From each window, K regions are proposed (each with a different scale or aspect ratio). These regions are also known as **Anchor Boxes** which are key to sharing features among the RPN and the Fast R-CNN detection net.

Each region proposal/anchor is transformed into a fixed-length feature vector and processed by some FC layers which compute scores (also called **Objectness Scores**) indicating how likely these boxes con-

tain objects of interest as well as a more adjusted/precise bounding box. Then those regions that have a low Objectness Score are discarded (this is normally done by setting a threshold as a hyperparameter). In Figure 6 you can observe a sketch of how Faster R-CNNs process an Image.

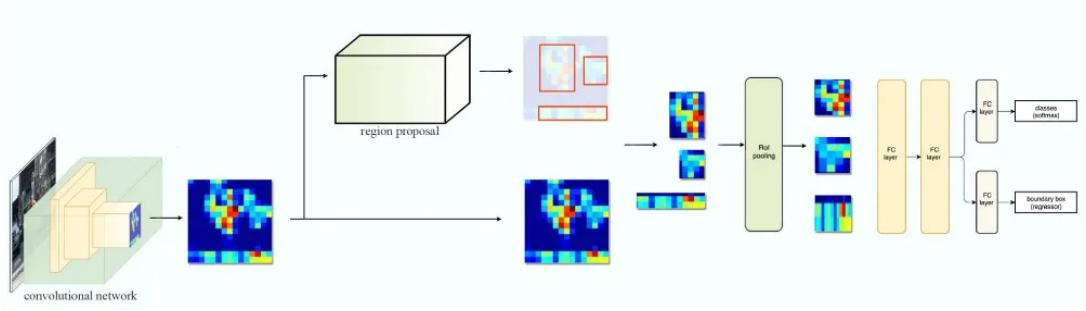

Figure 6: Stages that an image goes through when processed by a Faster R-CNN [10].

2.2 FASTER R-CNNS (R-50-FPN)

This architecture is practically the same as the previously described Faster R-CNN, differing only in the backbone used. It uses a **ResNet-50** as Backbone which is a CNN known for being able to extract intricate hierarchical features within input images. In addition, the backbone has what is known as a **Feature Pyramid Network** (FPN) which combines the feature maps of different levels of the Resnet 50 into a pyramid of feature maps, or in other words, feature maps at different scales (feature Pyramid). Clarify that the FPN is not an object detector but rather a feature extractor that works together with object detectors (i.e. RPN) [10].

In Figure 7 you can observe the stages that an image goes by when processed by this architecture. As can be seen, it is very similar to the stages shown in Figure 6 with the difference that instead of generating a feature map, a pyramid of feature maps is generated (see orange box) on which the RPN is applied to generate RoIs. Depending on the size of the RoI, the feature map with the most appropriate scale is selected and then RoI pooling is applied to it (see green box). Aside from this modification of the backbone, everything else works the same as in the Faster R-CNN architecture.

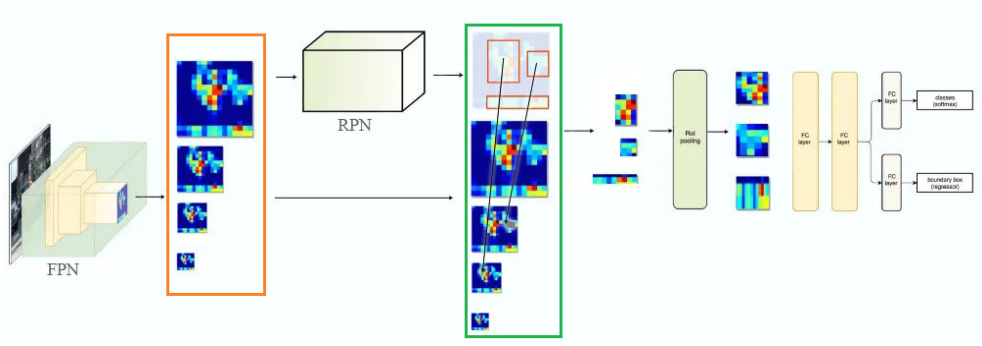

Figure 7: Stages that an image goes through when processed by a Faster R-CNN (R50 FPN) [10].

2.3 OBJECT DETECTION ESSENTIALS

2.3.1 Annotations

Annotations are important for training or fine-tuning an object detection model because they provide ground-truth information. The objects that the model has to detect must be marked with a bounding box.

The project by Sheng et al. uses the pascal_voc format with four-pixel values that define the coordinates of the bounding box. The format is represented as [x_min, y_min, x_max, y_max] [11], where x_min and y_min indicate the coordinates of the top-left corner of the bounding box (left arrow), and x_max and y_max represent the coordinates of the bottom-right corner of the bounding box (right arrow) in Figure 8. An annotation has the following format:

$$\texttt{category\_id} : 1, \texttt{bbox} : [98, 345, 420, 462], \texttt{bbox\_mode} : 0$$

The bounding box with the pascal_voc format is passed as "bbox_mode: 0" in the annotation. "Category_id: 1" is for an image that has a camera. An image that does not have a camera has an empty annotation, which is just an empty bracket [ ]. We will use this same annotation format.

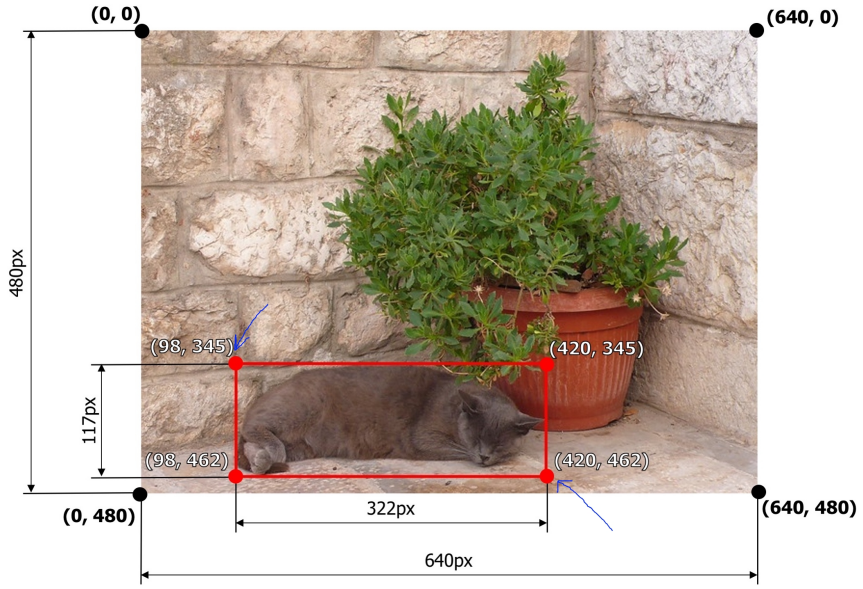

Figure 8: An illustration of a bounding box is provided with blue arrows indicating the coordinates that should be included in the annotation [11].

2.3.2 Confidence Score & Intersection over Union (IoU)

A confidence score threshold is chosen to filter out false positives and ensure that a predicted bounding box has a certain minimum score [12]. Furthermore, the confidence score indicates how likely the bounding box is to contain an object. The confidence threshold is a hyperparameter that can range from 0.0 to 1.0. 0.0 indicates a low-confidence detection, whereas 1.0 indicates high-confidence detection. Sheng et al. has set the confidence threshold to 0.0 but we changed it to 0.5 to decrease false positive detection.

The Intersection over Union is a metric to measure the accuracy of the localization (the overlap between the predicted bounding box and the ground-truth bounding box) and compute the localization errors in object detection models. During the validation or testing phase, the images and their respective annotations are passed to the object detection model. The model makes a prediction about where that specific object is located, and the prediction is given as a bounding box, as illustrated in Figure 9. The IoU ranges from 0.0 to 1.0, where 0 means no overlapping between the boxes and 1 indicates perfectly overlap between the boxes. To calculate the IoU, the calculation is performed by the amount of overlap between the predicted bounding box and the ground-truth bounding box, as illustrated in Figure 10. The IoU threshold is a hyperparameter that can be set before the validation or the testing phase [13].

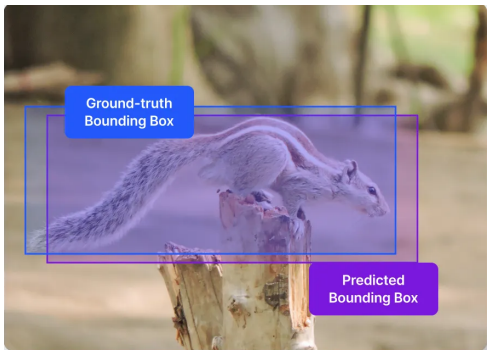

Figure 9: Prediction overlap Ground-truth [13].

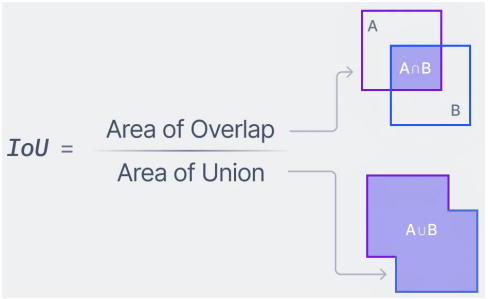

Figure 10: IoU Calculation [13].

The source code by Sheng et al. has set the IoU threshold at 0.5, which means that if the model outputs a prediction bounding box around an object overlapping the ground-truth bounding box with an IoU less than 0.5 the image will be considered as a false positive detection as illustrated in Figure 11.

A prediction is true positive if the confidence score and the IoU are greater than or equal to the threshold 0.5. A prediction is false positive if the confidence score is greater than or equal to 0.5 but the IoU is the less than 0.5.

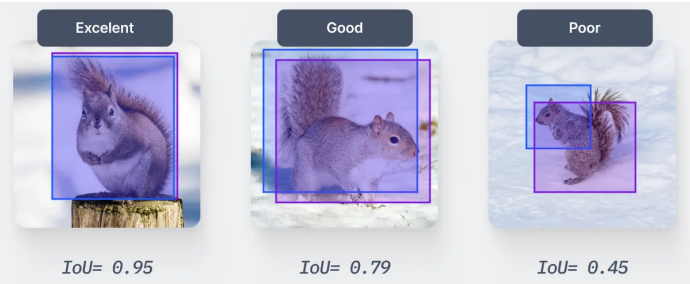

Figure 11: IoU comparative performance [13].

2.3.3 Image augmentation

In order to reduce model overfitting[7] and increase the diversity of the dataset, and accommodate for the various scenarios in which a camera might be situated. We applied image augmentation (such as random adjustments in brightness, flip, contrast, saturation, etc., as illustrated in Figure 12)
Note that the applied image augmentations are randomly applied. It is beyond our control what images will be augmented.

Image augmentations such as *cutout* and *crop* are avoided due to the risk of removing a camera when it is placed at the edge of the image. *Grayscale*, *hue*, or *mosaic* are not chosen because they lack realism.

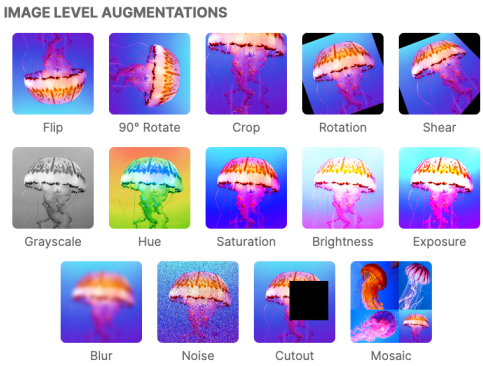

Figure 12: An illustration of different image augmentations [14].

[7] **Overfitting:** Occurs when a statistical model fits its training data exactly. When this happens, the algorithm cannot perform accurately against unseen data, is unable to generalize, and therefore does not fulfill its purpose.

3 DATA ACQUISITION

To achieve our research goals, we first needed to gather a dataset with enough images to cover all the streets of Copenhagen and the Frederiksberg District.

3.1 IMAGE GATHERING

3.1.1 Coordinate Sampling

The first step was to obtain coordinates spaced far enough in order to have as much ground coverage as possible, while also being able to identify the direction of the road (which will be crucial for later capturing photos of the facades on either side of the road). We considered that a good starting point is to get 150.000 coordinates. With this amount of coordinates, it is more than enough to have a complete coverage of the streets of Copenhagen, which can be seen in Figure 13. This amount of coordinates is equivalent to around 300.000 images since per coordinate the idea is to get the photo of the right and left facade.

We use the Python Library **OSMnx** [15] to create a graph that represents the street network of Copenhagen and the Frederiksberg District, where nodes represent intersections or key points, and edges represent the streets connecting these nodes as illustrated in Figure 13. The graph is created from a *geojson* file (both the file and the code can be found in our repository [16]).

To achieve a representative sample, a normalization technique is applied. Each street segment's length is divided by the total road length in the network, resulting in a normalized length measure. This normalization process is performed so that street segments are selected proportionally to their actual length. Longer segments, being the backbone of the road network, carry a higher probability of being chosen during the random sampling.

After the normalization, as many street segments as coordinates to be sampled are selected according to their normalized lengths, which means that the same segment can be selected more than once (those streets that are longer will be selected several times). For each segment, a random value between 0 and 1 is sampled and multiplied by the length of the road segment. Thus, if the random value is 0, we

would get the coordinates for the beginning of the street, and vice-versa for 1.

Finally, we interpolate this random position of the segment to find the coordinates it corresponds to. In addition, the bearing or direction of each street is determined by using two slightly distant points which are obtained from the randomly sampled point. This direction is calculated using the `geodesic.inv` function, a method that computes the shortest distance between two points on the Earth's surface. The importance of the road heading will be explained in the next subsection.

Before obtaining the images of these coordinates, we preprocess them, deleting those coordinates that are nearby (less than 7 meters). This is done in order to avoid the possibility of having duplicated street view images, because if two coordinates are very close to each other, the Street View API may consider that these are the same and therefore return duplicate images.

We transform the GPS coordinates (latitude and longitude) to UTM (Universal Transverse Mercator) expressed in meters. Then we index the transformed UTM coordinates with a KDTree[8] (*k-dimensional tree*) which organizes these in space. We iterate over all the coordinates and query the KDTree to identify the nearby points to each coordinate, deleting those with one or more points in the 7-meter radius.

After preprocessing we end up with a total of 113.602 coordinates as illustrated in Figure 13. In the next subsection, we will explain how to extract images from these.

Figure 13: Road map filled with red dots resembling coordinates placed over Copenhagen and Frederiksberg.

[8] **K-D Tree:** Binary search tree where data in each node is a K-Dimensional point in space [17].

3.1.2 Image Acquisition Process

When it comes to obtaining Google Street View Static Images, there are two main options. The first and most commonly used option is the ***Google Street View Static API*** [18]. The only downside is that making requests has a cost. However, Google gives 300$ when you create a new account, and then every month gives 200$. For each 7$, Google allows to make 1.000 requests. Therefore with an account, with the initial 300$ we could make around 43.000 requests, plus 29.000 requests each month [19].

The second option is using ***Mapillary*** [20], an API that allows us to get images for free. Like Google, they also have cars that go through the streets scanning them as they pass by. However, for locations/coordinates that they have not yet been able to cover, they allow people to upload photos of what is near that coordinate, resulting in images with totally different formats.

In the end, we opted to use the Google API, creating a total of three accounts (each of these associated with a different bank account and a different email address to comply with the regulations imposed by Google). In Figure 14 can be observed the total amount of requests we have available with the budget provided by Google.

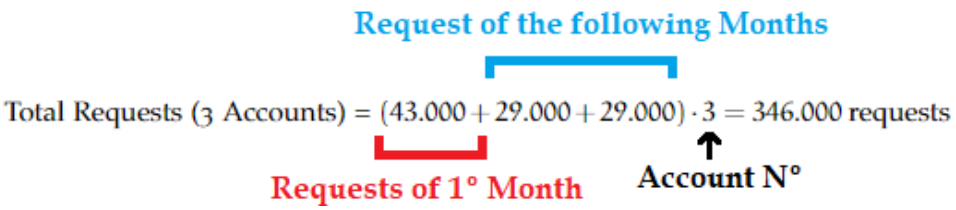

Figure 14: Calculation of the number of total requests for three accounts according to the budget provided by Google.

To get the images, you need to fill in a URL that contains the details of the image you want to get from the API (check Figure 15 to see the request URL). These fields are :

- **size**: Image Resolution (in our case we set it to 640x640).
- **location**: Concatenation of Latitude and Longitude (i.e. 55.706, 12.455).
- **fov**: Horizontal field of View of the image expressed in degrees (we chose 80 since it was a relatively good *fov* for our purpose).
- **heading**: Orientation of the camera according to the compass (where 90°indicates east and 180°indicates south).
- **pitch**: Specifies the upward or downward angle of the camera relative to the Street View vehicle (we set it to 8).

- **API Key**
- **API´s Signature** (optional)

It is precisely because of the heading variable that it is important to know the direction of the street in which a certain coordinate is located. What we do is, knowing this direction, we add or subtract 90°to obtain images in the directions perpendicular to the street (which correspond to the right and left facades of this coordinate). Thus, per coordinate, we end up getting two images.

```
https://maps.googleapis.com/maps/api/streetview?size=400x400&location=47.5763831,-122.4211769
&fov=80&heading=70&pitch=0&key=YOUR_API_KEY&signature=YOUR_SIGNATURE
```

Figure 15: Example of how the Request URL looks like. The Metadata URL is the same, but adding "/metadata" just after "streetview".

However, it should be noted that not all requests have a corresponding Google Street View Image. This is why in case you do not want to waste image requests, you can first make a free metadata request, which will tell you if there is an image for the given coordinate, and then perform the image request if the metadata request was successful.

After collecting images for the coordinates over these months, we have gathered a total of 226.150 Street View images.

4 SOURCE CODE SET UP & DATASET CREATION

In this section, our goal is to introduce the source code and checkpoint given by Sheng et al. We then introduce the details of the operating system and hardware we used. In addition, we explain how we set up our development environment to make the code work properly. Lastly, we provide an overview of how the source code operates with a closer look at the Python scripts that make training and testing possible.

4.1 DEVELOPMENT ENVIRONMENT

4.1.1 Introduction

The source code for the project by Sheng et al. is available on GitHub [21]. It includes a PyTorch Lightning[9] checkpoint [22] named "best.ckpt". A checkpoint makes it possible to fine-tune a model or use a pre-trained model for evaluation without the need to retrain the model from its original state.

It should be noted that a checkpoint must be loaded into the same model (Faster R-CNN R50 FPN) from which it was created before it can be used for fine-tuning or evaluation.

A checkpoint consists of stored weights and parameters in a structured way (following the architecture of the model) that have been achieved by training on images with and without cameras from 10 major US cities and six other cities around the world, resulting in a checkpoint size of 494MB.

4.1.2 Operating System & Hardware

The project by Sheng et al. utilizes Detectron2[10] version 0.4 [23]. Detectron2 runs on Linux or Mac OS.

The following hardware will be used for fine-tuning and evaluation: an 8 GB NVIDIA GeForce RTX 3070 laptop GPU with driver version

[9] **PyTorch Lightning:** PyTorch Lightning is an open-source Python library that simplifies and standardizes the training and research process for deep learning models built with PyTorch.

[10] **Detectron2:** A Facebook AI Research's next-generation library that provides state-of-the-art detection and segmentation algorithms.

470.199.02, having 5120 CUDA cores, an 8-core Ryzen 7 5800H CPU, and a system equipped with 32GB of RAM.

4.1.3 Environment

The version 20.04 of Ubuntu Linux is chosen in order to be compatible with the CUDA toolkit[11] version 11.1, PyTorch[12] version 1.10.1, Torchvision 0.11.2 and PyTorch Lightning 1.1.4 [26] which is required for making Detectron2 version 0.4 work properly. We used Anaconda[13] and Python version 3.8. These dependencies are important for the source code to work.

The project by Sheng et al. provides a requirement text file that contains all the dependencies, with their respective versions, that an environment needs in order to be able to run the code. We will leave our `requirements.txt` with the dependency versions that worked for us in our repository [16].

4.2 SOURCE CODE

In this section, we provide an overview of the source code by Sheng et al. The overview will mainly focus on the Python scripts and how they set up the training/validation or test process.

4.2.1 Source code overview

The source code has four important Python scripts that we want to elaborate on in the following. The four scripts are *main*, *detection*, *lightning*, and *data*.

The *main* Python script creates a train/validation task or a test task. The train/validation task consists of user-specified hyperparameters, a train/validation dataloader including augmentation, and a model that will be used for training or fine-tuning. The test task consists of a test dataloader and a model that will be used for evaluation.

The *detection* script is responsible for downloading the Faster R-CNN R50 FPN model from Detectron2.

[11] **CUDA:** The NVIDIA® CUDA® Toolkit provides a development environment for creating high-performance GPU-accelerated applications [24].

[12] **PyTorch:** An open source machine learning framework that accelerates the path from research prototyping to production deployment [25].

[13] **Anaconda:** Anaconda is an open-source distribution of Python for data science that aims to simplify package management and deployment [27].

The *data* script stores the path of the CSV file that contains the annotations and paths of the images that will be used for training/validation or evaluation.

The *lighting* script initializes the model passed from the *detection* script, the hyperparameters from the *main* script and the train/validation or evaluation dataset coming from the *data* script and then passes the task back to the *main* script to run the train or test task.

4.3 TRAINING, VALIDATION, AND EVALUATION

In the following section, we explain our process for selecting images and creating the dataset used for training, validation, and evaluation.

Before moving on, we want to clarify how we deal with images that contain two or more cameras. In this context, if an image has two cameras, we consider it as two distinct detection images (detection tasks) rather than a single image. In case of having two cameras in a single image, detecting only one will result in one true positive and one false negative instance.

4.3.1 Methods: Creating dataset images for training and validation

To create a train/validation dataset, we identified two methods based on our understanding of how the model trains and evaluates on images and what it returns as results.

The first method: We manually go through a subset of the images we had gathered with the intention of making sure that we get all the images with a camera. The advantage of this method is to reliably obtain a dataset with cameras, with the drawback of having to create an annotation for each image that has a camera manually, which is a time-inefficient method and a lot of work.

The second method: Involves running over a set of images through the model in the *test mode* with a confidence score threshold of 0.50. This means that any image having a camera with a model certainty less than 0.50 would be ignored, while the rest would be included. We chose this method because it saved us a lot of time on labeling compared to the first method. Using the second method led to a low number of images with cameras.

The result of using the second method was a low number of images with cameras, due to the reduced predictive capabilities of the US model over the Danish street environment.

4.3.2 Selection of images for train/validation dataset

As mentioned in Section 3.1.2, we gathered 226.150 images in total. We used 10% resulting in a small subset of 22.615 images. This subset was processed by the US model in just 14 minutes (on GPU) during the test mode, which outputted images along with their corresponding annotations. Of the 22.615 images, only 1.387 images were classified as having a camera. We manually inspected the images, and found 137 true positive images and 1.250 false positive images.

In total we used 40% of the 226.150 collected images to ensure a sufficient number of true positive images. The end result was a train/-validation dataset consisting of 501 training images, each accompanied by its corresponding annotations, and 88 validation images with their respective annotations. Furthermore, to ensure that we had a balanced dataset we added 501 training images and 88 validation images that did not have cameras/annotations.

4.3.3 Selection of images for evaluation dataset

To evaluate whether the model's capability to identify cameras has improved, we generated an evaluation dataset comprising images that the model had not previously encountered.

Our method of selecting images containing cameras involved physically exploring the streets of Copenhagen and noting the location of 100 cameras.

For the remaining 100 images without cameras, we manually selected from our collection. The images we chose contained elements such as street lights, traffic lights, wall holes, and road signs, aiming to resemble features that could be mistaken for a camera.

5 ESTABLISHING A BASELINE & MODEL FINE-TUNING

5.1 EVALUATION OF THE MODEL BY SHENG ET AL.

The idea was to use the evaluation dataset to evaluate the US model, to check to what extent it was able to generalize to Danish streets.

Our evaluation process for the US model involved feeding it 200 images that had not been encountered before (100 with cameras and 100 without cameras).

We used these predictions as a baseline and tried to find a model that performed better by fine-tuning the US Model. We state its performance in the results section.

5.2 FINE TUNING THE MODEL

We fine-tuned the US model by freezing the backbone layers of the architecture while leaving the remaining layers unfrozen. It is a common practice in transfer learning for object detection to freeze the backbone layers. These layers are responsible for learning generic features from images, and by freezing them, we leverage the pre-trained weights to capture general image features. The last layers are responsible for learning the characteristics of the objects the model needs to classify [28].

Subsequently, we used the train/validation dataset mentioned in Section 4.3.2, we adopted the pre-defined hyperparameters established by Sheng et al.'s project with minor changes: These included a learning rate of $1e-05$, patience[14] of 30, a batch size of 8, and a list of image augmentations as mentioned in section 2.3.3. We reduced the maximum number of training epochs from 100 to 45 because we found that 100 was excessive for our small fine-tuning dataset.

[14] **Patience:** The "patience" parameter determines when to stop if there is no improvement on the validation loss.

5.2.1 Evaluating the fine-tuned model

After completing the fine-tuning process, we acquired a new model, which we refer to as the DK model.

It shares the same architecture as the US model but has different weights because it has trained on Danish street images.

We evaluated the DK model by testing it on the same set of 200 images that were previously used to evaluate the US model.

5.2.2 Experiment details

We fine-tuned and evaluated both models on a single GPU. In both instances, the evaluation phase lasted under a minute running through 200 images, and the fine-tuning process took approximately 30 minutes over 39 epochs.

Figure 16 shows the train and validation loss over 39 epochs. As mentioned in Section 5.2 we set the patience to 30. At epoch 8 the validation loss was the lowest hence after 30 epochs which is 38 the fine-tuned session stopped because there was no improvement in the validation loss.

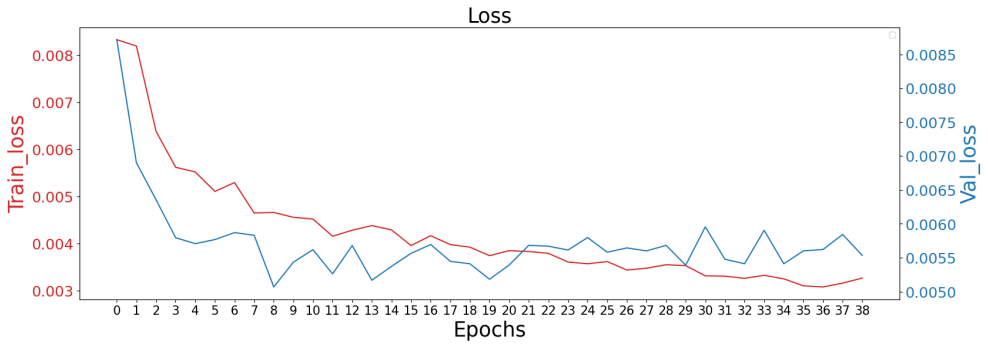

Figure 16: Train loss and Validation loss.

6 RESULTS

In this section, we present the results that we obtained from transfer learning and compare the US model and the fine-tuned model with each other. However, before we proceed, we would like to clarify the selected metrics used to quantify the performance of both models.

Precision indicates how well a model is at correctly identifying objects. Thus, high precision means that the model makes fewer mistakes, in other words, a low number of false positives [29].

Recall describes whether the model is able to detect the presence of a camera in the image. A high recall means that the model is good at not missing any camera, in other words, a low number of false negatives [29].

F1-score describes the trade-off between a model's precision (minimizing false positives) and recall (minimizing false negatives), and provides a single metric that quantifies the overall effectiveness of a model's performance in classification tasks [29].

Accuracy is a metric that measures the overall correctness of a model in all classes. Based on our balanced test dataset which has a size of 200 images, 100 images with camera, and 100 without, we did not choose the balanced accuracy metric [29].

Average Precision or the AP value is a scalar that is calculated on the basis of the area under the precision-recall curve. The AP value indicates how well a model performs in a specific class with respect to precision and recall. Taking into account its performance across different threshold values [29].

Mean Average Precision or mAP for short, involves calculating the AP for each class separately and then taking the average of these AP values. The mAP is a comprehensive metric to assess the overall performance of the model in all classes. Since we only have one class, the mAP value is equivalent to the AP value [29].

6.1 MODEL RESULTS

Based on the results collected from both models, we created two confusion matrices, which we will explain in detail below.

The US model had in total identified 33 images out of 200 images as having a camera, whereas the DK model had in total identified 60 out of 200 images as having a camera.

We manually checked the images identified by each model. Figure 17 shows that the US model had 19 images with a camera of the 33 images and misclassified 14 images. Figure 18 shows that the DK model had 37 images with a camera of the 60 images and misclassified 23.

TARGET / OUTPUT	Camera	No_Camera	SUM
Camera	19 9.50%	14 7.00%	33
No_Camera	81 40.50%	86 43.00%	167
SUM	100	100	105 / 200

Figure 17: US Model

TARGET / OUTPUT	Camera	No_Camera	SUM
Camera	37 18.50%	23 11.50%	60
No_Camera	63 31.50%	77 38.50%	140
SUM	100	100	114 / 200

Figure 18: DK Model

Based on the results we obtained, as illustrated in Figure 19, the US model was able to correctly detect 19 images with a camera while the DK model correctly detected 37 images indicating a 94.736% recall improvement. In terms of precision, the US model achieved 57.57%, while the DK model exhibited a precision of 61.6%, signifying a 7.105% precision enhancement. Regarding the F1-score, the US model achieved a score of 28.57%, while the DK model outperformed with a score of 46.25%, representing an impressive 61.875% improvement in the F1-score. Lastly, regarding the accuracy, the US model achieved a 52.5% accuracy rate, while the DK model attained a 57% accuracy rate, marking an 8.571% accuracy gain.

	US model	DK model	Increase in percentage	
Recall	19	37	94,74	%
Precision	57,58	61,67	7,11	%
F1-score	28,57	46,25	61,88	%
Accuracy	52,5	57	8,57	%

Figure 19: Calculation: Recall, Precision, F1-score, Accuracy, difference increase in %

Figure 20 shows the precision - recall curve. The area under the curve is the average precision for our camera class which was 0.79. This indicates that the DK model is effective in detecting the presence of a camera (precision) and not missing out on cameras (recall). At a recall of 0.8, the precision is 0.7, suggesting that when the model identifies 80% of actual positives, 70% of the identified instances are true positives.

Figure 21 illustrates the mean average precision progress along the 39 epochs. The highest mean average precision that the DK model yielded was 0.79 on epoch 23.

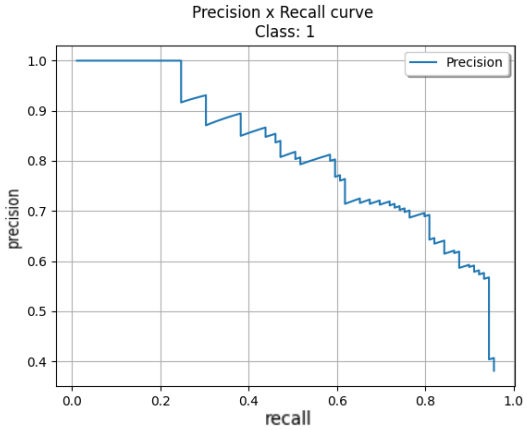

Figure 20: Precision-Recall Curve

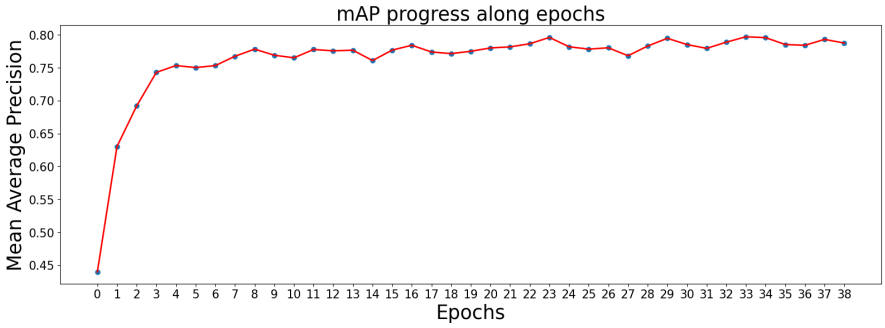

Figure 21: Mean Average Precision progress along the epochs.

6.2 MODEL COMPARISON

In Figure 22 all images that were classified as having a camera by the US model and the DK model are plotted. The plot illustrates the confidence level of each model in predicting the presence of a camera in the images. The DK model has a higher confidence score when detecting a camera compared to the US model for the same image.

To illustrate the performance gain by the DK model, an image the US model detected as having a camera with a confidence score of 0.52, detected by the DK model with a confidence score of 0.95. The DK model is not flawless, an image detected by the US model with a confidence score of 0.97 were detected by the DK model with a confidence score of 0.89. Overall, the DK model had more images with a high confidence score compared to the US model.

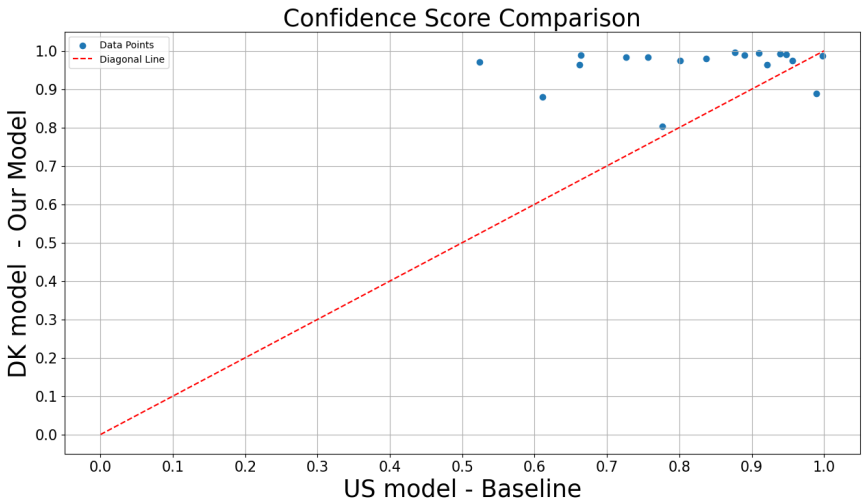

Figure 22: Confidence Score comparison

Figures 23 and 24 show the same image correctly identified by the DK model and the US model. These images have a camera and their respective predicted bounding boxes. The primary emphasis here is on the increased certainty of the DK model compared to the US model, which is an indication of improvement.

Figures 25 and 26 show two different images that the DK model misclassified as having a camera. This indicates that the DK model is not flawless.

Figure 23: Prediction by DK Model.

Figure 24: Prediction by US Model.

Figure 25: Prediction by DK Model.

Figure 26: Prediction by DK Model.

6.3 ANSWERING RESEARCH QUESTIONS

In this section, we answer our research questions introduced in Section 1.

Can the US model effectively identify cameras on Danish streets, and to what extent does it demonstrate proficiency in this task?
Overall, the US model demonstrated proficiency in detecting cameras, but has not reached its capability to generalize on Danish streets.

Is the predictive capability of the model improved on Danish streets after the fine-tuning process?
Based on the information presented in Figure 19 & 22, we observed improvements in all metrics and the DK model's prediction certainty. This indicates that fine-tuning the US model improved the model's object detection capability (detecting cameras), although some misclassifications persist, albeit at a reduced rate compared to the US model.

7 CONCLUSION & FUTURE WORK

The US model by Sheng et al., although proficient in detecting cameras, did not perform as well on Danish streets. This is not due to inadequate or insufficient training (indeed it has been trained on a larger dataset than ours), but because it is not tuned to the unique characteristics of Danish roads, such as the positioning of traffic lights and streetlights, architectural elements that can resemble small cameras. Furthermore, variations in camera shape, size, and color; the distance between the Google Street View car; the angle at which the images were captured; and the quality of the image all play substantial roles in the model's detection capabilities.

In *conclusion*, fine-tuning the US model produced a better model. The DK model is more proficient at detecting cameras on Danish streets than the US model, based on the analysis.

We will rely on the DK model in future attempts to identify the cameras in our collected images.

In our upcoming work, we plan to extend this project to a master's thesis project.

For the master's thesis, we aim to gather more images to add to our database. Furthermore, we want to create a larger train/validation dataset with the hope of improving the DK model's performance in detecting images with a camera, and we also aim for better metrics scores than those achieved with the DK model.
Once we attain a model that performs more effectively on Danish streets, we will use it to address the questions posed in section 1, which remains the primary motivation for this research project and the master's thesis.

- How many surveillance cameras are on the streets of Copenhagen and where are they located?
- Are citizens inadvertently subjected to excessive surveillance in public spaces?
- Do certain neighborhoods or demographic groups experience higher levels of scrutiny?
- Does the historical context of crime influence the density of surveillance?

BIBLIOGRAPHY

[1] DR News by Karen Klærke. From 30 to 100 meters - so far shops and nightclubs can film out on the road. https://www.dr.dk/nyheder/indland/fra-30-til-100-meter-saa-langt-kan-butikker-og-natklubbers-kameraer-komme-til-filme.

[2] Jacob Mchangama. Justicia main page. https://justitia-int.org/.

[3] Hao Sheng, Keniel Yao, and Sharad Goel. Surveilling surveillance: Estimating the prevalence of surveillance cameras with street view data. 2021. https://doi.org/10.48550/arXiv.2105.01764.

[4] Steve Coast. Open street map. https://www.openstreetmap.org/#map=7/56.188/11.617.

[5] Tuomo Lahtinen Lauri Sintonen Timo Hamalainen Hannu Turtiainen, Andrei Costin. Towards large-scale, automated, accurate detection of cctv camera objects using computer vision. applications and implications for privacy, safety, and cybersecurity. 2021. https://arxiv.org/abs/2006.03870v3.

[6] Karthik Mittal. A Gentle Introduction Into The Histogram Of Oriented Gradients. https://medium.com/analytics-vidhya/a-gentle-introduction-into-the-histogram-of-oriented-gradients-fdee9ed8f2aa.

[7] Ahmed Fawzy Gad. Faster r-cnn explained for object detection, 2020. https://blog.paperspace.com/faster-r-cnn-explained-object-detection/.

[8] Rohith Gandhi. R-cnn, fast r-cnn, faster r-cnn, yolo: Object detection algorithms, 2018. https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e.

[9] Sik-Ho Tsang. Review: Fast R-CNN (Object Detection). https://medium.com/coinmonks/review-fast-r-cnn-object-detection-a82e172e87ba.

[10] Jonathan Hui. Understanding feature pyramid networks for object detection (fpn). https://jonathan-hui.medium.com/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c.

[11] Albumentations Documentation. Bounding boxes augmentation for object detection. https://albumentations.ai/docs/getting_started/bounding_boxes_augmentation/.

[12] Simon Wenke. Confidence Score: The Forgotten Dimension of Object Detection Performance Evaluation. https://www.mdpi.com/1424-8220/21/13/4350.

[13] Deval Shah. Intersection over Union (IoU): Definition, Calculation, Code. https://www.v7labs.com/blog/intersection-over-union-guide.

[14] James Gallagher. Generate image augmentations with roboflow. https://blog.roboflow.com/image-augmentation/.

[15] Geoff Boeing. Osmnx: Python package for street networks, Year of the latest version or last update. https://osmnx.readthedocs.io/en/stable/.

[16] Claudio Sotillos and Haidar Imad. Github repo "camera-detection", 2023. https://github.com/claudio-sotillos/Camera-Detection.

[17] GeeksforGeeks. Search and Insertion in K Dimensional tree. https://www.geeksforgeeks.org/search-and-insertion-in-k-dimensional-tree/.

[18] Google. Google maps street view documentation. https://developers.google.com/maps/documentation/streetview/overview?hl=es-419.

[19] Google. Google maps platform pricing. https://mapsplatform.google.com/pricing/.

[20] Mapillary: Access street-level imagery and map data from around the world. https://www.mapillary.com/?locale=es_ES.

[21] Paper´s GitHub Repository: Sharad Goel, Hao Sheng, stanford-policylab . Surveilling Surveillance: Estimating the Prevalence of Surveillance Cameras with Street View Data, 2021. https://github.com/stanford-policylab/surveilling-surveillance.

[22] PyTorch Lightning. Pytorch lightning documentation - checkpointing. https://pytorch-lightning.readthedocs.io/en/1.6.4/common/checkpointing.html.

[23] Detectron´s GitHub Repository: Facebook Research. Detectron2: Object Detection with PyTorch. https://github.com/facebookresearch/detectron2.

[24] NVIDIA. Cuda toolkit. https://developer.nvidia.com/cuda-toolkit.

[25] PyTorch. https://pytorch.org/.

[26] PyTorch Lightning. https://lightning.ai/.

[27] Anaconda. https://www.anaconda.com/.

[28] François Chollet. Transfer learning fine-tuning. https://www.tensorflow.org/guide/keras/transfer_learning.

[29] Deval Shah. Mean Average Precision (mAP) Explained: Everything You Need to Know. https://www.v7labs.com/blog/mean-average-precision.