Road Traffic Analysis with Surveillance Cameras using Synthetic Training Data

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Evgeny Toropov

M.S. Applied Mathematics and Physics, Moscow institute of Physics and Technology
B.S. Applied Mathematics and Physics, Moscow institute of Physics and Technology

Carnegie Mellon University
Pittsburgh, PA

May, 2019

Acknowledgments

The pursuit of the PhD degree has been one of the most memorable experiences in my life. It allowed me to grow both professionally and personally. I have the most sincere appreciation of the people that surrounded me during this trip, and helped me along the way. First of all, I want to express my gratitude to my advisor Prof. José M. F. Moura, who guided my research, encouraged me to work on unsolved problems, supported me in those times when the work seemed frustrating, and navigated me towards the graduation. Any obstacle, whatever its nature, always turned out solvable after addressing it together with Prof. Moura. I want to take this opportunity to express my sincere gratitude for providing me a role model and illustrating how good judgement and persistence can lead one to success.

I am grateful to the other member of my PhD committee, Prof. Aswin Sankaranarayanan, Prof. Abhinav Gupta, and Prof. Deva Ramanan, for serving on my defense. I highly appreciate their valuable suggestions on my proposal, that allowed me to concentrate on the meaningful parts of the project, their inspiring questions, that allowed me to see the thesis from different angles, and their insightful remarks that made the dissertation better. I also highly appreciate the help of the program academic advisor, Nathan Snizaski. His invariably professional and fast responses to my numerous questions throughout the program made the PhD experience easier in so many aspects. I would also like to acknowledge the department funding that made this research possible.

I would like to express my great gratitude to my colleagues, who I was fortunate to work with. I am grateful to Satwik Kottur, Rajshekhar Das, Jonathan Mei, Yuan Chen, Shanghang Zhang, Mark Cheung, Umang Bhatt, Yang Li, John Shi, Liangyan Gui, Beatriz Ferreira, Stephen Kruzick, João Domingos, Joya Deri, Aurora Schmidt, Kyle Anderson, Subhro Das, June Zhang, and Augusto Santos, among others. Their friendship supported me from the first day in the office, and I am convinced that their willingness to help will be with me throughout my life. I thank my colleagues and friends for the discussion of numerous work aspects and for their feedback, that allowed me to look at

many problems from a different angle.

I was very fortunate to have an excellent team of students that helped me in my work on the datasets presented in this thesis. In particular, I thank Spandan Gandhi, Eileen You, Yoonju Pak, Michelle Hsieh, Phoebe Soong, and Laura Chen. Their responsible approach to work, their pro-activeness, and their patience towards the software issues were the factors that made the publication of the datasets possible.

Finally and most importantly, I want to thank my partner and my family for the enormous help and support. My partner Paola Buitrago is the kindest and the most patient person that I know. I feel extremely lucky to have her in my life for the past three years. She was always there with me to share my joy from work and from learning, as well as in times when I needed her support. It is hard to say if without Paola's help this thesis would ever be finished. But one thing I am certain about is that her insights on the problems I worked on and her opinions on the way I approached some of them, contributed a lot to this dissertation. I want to thank my parents Victor and Tatyana, and my sister Ekaterina for creating a place where I can draw my strength from. Though we are a half of the world away from each other, I know that the sensation of being at home is only a phone call away. Their support was always there during this long journey, starting from the decision of enrolling into the PhD program and all the way to choosing my next steps.

Abstract

Municipalities around the world provide public access to their traffic surveillance cameras via a web-interface. Though such web-interfaces usually provide only low frame-rate and low resolution video, the diversity and the low cost of this data is an opportunity for researchers to work on problems ranging from finding empty parking spots in a street to traffic rules enforcement. Modern machine learning algorithms based on Convolutional Neural Networks (CNNs) are capable of accurately locating and analyzing each car in an image. However, CNNs are notoriously hungry for expensive labelled training data, which is only available to large companies. We aim to close this gap by incorporating synthetic computer-generated data into the training process. The purpose is to significantly reduce the required amount of labelled real data and reduce the cost of annotation per image, thus democratizing the practical usage of CNNs for the case of surveillance cameras. The contributions of this thesis are 1) a dataset of 8,000 labelled low-resolution images of individual cars and a dataset of 80,000 synthetic car images tailored for surveillance cameras. The images are annotated with pixel-level background mask, car type, color, and orientation; 2) a dataset of over 1,000 high-quality 3D CAD models of cars; 3) experiments that demonstrate the usefulness of synthetic images in detection and semantic segmentation tasks for surveillance camera scenarios; and 4) a dataset management tool that facilitates the work with annotations in computer vision. To sum up, the dissertation aims to promote synthetic images as a tool for machine learning and outline its advantages and limitations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Municipalities of many cities around the world have installed traffic surveillance cameras to monitor city roads. For example, hundreds of cameras are installed in New York City [1], thousands in Moscow [2] and Beijing [3]. These cameras may provide real-time streaming video used by automated analytics systems [1, 2]. Such systems can be used to switch the traffic light at the best moment [4], to notify drivers of a traffic jam [5], or to spot traffic rule violations [2]. Video information also has the potential for more advanced uses following the concept of smart cities and ranging from helping the police to track a car across the city to warning nearby self-driving cars of a vehicle turning from a blind spot around the corner.

Existing automated traffic surveillance systems use various methods for analyzing video streams, including detecting cars in images. Standard methods include background subtraction or using infrared sensors [6, 7, 8]. More advanced algorithms take advantage of machine learning methods boosted with the growing popularity of Deep Convolutional Neural Networks (CNNs) [9]. Such methods are not only able to accurately detect cars in an image, but can also classify a car by make and model, given sufficient image resolution [10].

However, CNNs are notoriously hungry for labelled training data. Data labelling is normally performed by human annotators, either in-house or employed via crowd-sourcing

services, such as Amazon Mechanical Turk [11, 12, 13]. The annotation process is tedious for workers and expensive for customers. For example, pixel-wise labelling of a single image takes more than an hour [11, 12], and according to our rough estimates, millions of labelled images are needed to cover various corner cases. Therefore, the amounts of data sufficient to train a robust detector are very expensive.

We aim to close this gap by incorporating synthetic, computer rendered data into the training process. The purpose is to significantly reduce the required amount of labelled real data and reduce the cost of annotation per image, thus democratizing the practical usage of CNNs for the particular case of fixed viewpoints of surveillance cameras. We argue that low-level differences in appearance between real and synthetic images can be learned in an unsupervised way. At the same time, synthetic data captures the relevant high level information about the image of a car, such as how individual parts of a vehicle look like, how they combine together to image a car under a given viewpoint. This is the information we aim to learn from synthetic data and transfer to the real data. Moreover, synthetic images can be used where real annotations are expensive or difficult to collect, for example, to represent rare situations.

Several datasets have recently been proposed to aid research in traffic surveillance systems, e.g., UA-DETRAC [14], CompCars [10], and AI CITY [15]. They focus on the tasks of car detection, tracking, and re-identification of vehicles. The applications of these tasks range from automatic traffic density estimation to tracking a vehicle across a city [14].

At the same time, many problems require rich information about individual cars. For example, detecting if a car has crossed a lane or did not stop in front of a stop sign line requires the detailed knowledge of the position of the car on the road. Such applications require analyzing cars at the level of pixels.

In order to accomplish these goals, we build and release (1) a dataset of annotated images from traffic cameras in New York City [1], (2) the complementary dataset of synthetic

images of cars as if viewed from traffic cameras at different viewpoints and under different light and weather conditions, and (3) and a dataset of high-quality 3D car models that were used to generate these synthetic images.

Finally, collecting images for a dataset and managing an annotation team is a challenge without an effective tool to manage data and annotations. For a quick example, consider the task of combining the work of three annotators who labelled the same set of images (the practice of acquiring annotations with redundancy serves to ensure quality of the result). Labels that match should be merged, non-matching instances should be analyzed and either discarded or re-annotated. Additionally, it should be easy to collect statistics for any part of the annotated dataset. The complexity lies in the fact that many one-time scripts have to be written for individual sub-tasks. Such scripts are error-prone and are difficult to maintain. We have developed and are releasing a lightweight dataset management tool Shuffler, , which supports annotations from the moment they are created to the moment they are loaded into a Machine Learning framework, such as PyTorch [16] or Keras [17]. The tasks in the example above can be accomplished in Shuffler by running only a few commands.

To sum up, in this thesis we present a system that consists of 1) a dataset of pixel-wise labelled low-resolution car images and a dataset of synthetic car images tailored for surveillance cameras, 2) a dataset of high-quality 3D models of cars, and 3) a dataset management tool that facilitates working with annotations. We demonstrate through comprehensive experiments the usefulness of synthetic images in learning for surveillance camera scenarios. In particular, we compare training neural networks for the detection and the semantic segmentation tasks on real data, on synthetic data, and on their combination, and evaluate the performance of the trained models. We also evaluate the effects of different weather conditions, various components of the rendering pipeline, and domain adaptation methods on the performance.

The thesis is organized as follows. Chapter 2 formalizes the problem of traffic analysis, discusses experiments on synthetic data, describes our approach to training datasets, and introduces the problem of dataset management. Chapter 3 continues with the dataset management problem and presents the toolbox Shuffler, that we developed to solve it. Chapter 4 discusses the developed system for generating synthetic data, presents the dataset of 3D CAD car models called CADillac, and describes the process of rendering a synthetic image dataset Obelix. In Chapter 5, we describe how we collected and annotated a real dataset of car images Asterix, and in Chapter 6, we formally present Asterix and its synthetic counterpart Obelix. Chapter 7 continues by presenting hybrid synthetic-real images and describes the experiments we conducted to evaluate these data in the object detection setting. In Chapter 8, we conduct experiments on foreground image segmentation with Asterix and Obelix datasets. Finally, Chapter 9 concludes the thesis.

Chapter 2

Problem Statement and Approach

In this section, we first formalize the problem of traffic analysis as 1) the detection and 2) the foreground segmentation tasks. Then, we describe the experiments for evaluating the performance of the synthetic data in the context of these two problems. Finally, we describe the datasets that we created for the experiments.

2.1 Traffic analysis

We formulate the problem of traffic analysis as analyzing each car in each frame, in particular, detecting its direction, the vehicle type, and the foreground mask. In this context, we aim to explore how synthetic images may be incorporated into the training purpose, and whether they can complement or replace the real training data. We split the problem into two separate parts: (1) detecting individual cars and (2) analysis of individual cars.

2.1.1 Detection

Detecting individual cars is approached as finding all bounding boxes with any type of vehicles in an image, independent of type. In other words, an algorithm is trained to draw a rectangle around each car.

Object detection with bounding boxes is an established problem in computer vision that has found its way to various domains [18, 19, 20, 21, 22]. In 2001, the work of Viola and Jones [18] achieved real-time speed for face detection. Later, in 2010, Deformable Part Models [23] allowed to use detectors for more general categories of objects. Later, with the growing popularity of deep neural networks after 2012 [9], detectors based on convolutional neural networks quickly gained popularity. In 2015, the two-stage Faster-RCNN [24] achieved the top performance at PASCAL VOC [25] detection challenge and became a widely-used benchmark. Though new architectures [26] achieved comparable or exceeding performance in 2017, we use Faster-RCNN in our experiments as it is a well-established benchmark.

For detection, the algorithm receives the whole image as an input. Our interest is focused on traffic cameras, which have a stable, though changing in appearance, background image. It is important that a training image represents a typical image during the inference phase, therefore, the natural choice is to train separate models for each individual cameras. In particular, in Chapter 7, two cameras in NYC are analyzed.

In our experiments, a detection algorithm is trained to find one category of objects: "vehicle." Naturally, vehicles can be classified by type, however, we leave that classification for the next step of analyzing individual cars.

Small cars are not considered in training or inference because their appearance does not provide enough visual information for further analysis. In practice, if the average between the width and the height of the bounding box is less than 25 pixels, the car is discarded from training and being treated as "don't care" class.

2.1.2 Foreground segmentation for individual cars

Some applications require analyzing cars at the level of pixels, and the bounding box detection is no longer sufficient. As noted in Chapter 1, examples include finding empty

parking spots in a street and traffic rules enforcement. For example, the detailed knowledge of the car position is needed to determine if a car did not stop in front of a stop sign line.

Thus we consider the next problem – analysis of individual cars. In particular, we address the problem of predicting foreground masks of individual cars. This problem is known in computer vision as semantic segmentation [27]. After Convolutional Neural Networks proved successful in the classification task in 2012 [9], CNN architectures have been redesigned for the semantic segmentation problem [28, 29, 30, 31]. Fully-connected layers of classification networks can be replaced by convolutional layers. In effect, that allows to reuse architectures built for image recognition as the backbone of a network for semantic segmentation. In our experiments in Chapter 8, we use the Dilated Residual Network [32] for the architecture of the backbone.

2.2 Experiments with synthetic data

As described in Chapter 1, CNNs are notoriously hungry for labelled training data. At the same time, image annotation has remained a tedious process for human workers. Creating a segmentation mask was reported to take on average 1.3 minutes for one object instance for COCO [13], 60 minutes for the whole image for CamVid [11], and 90 minutes for the whole image for Cityscapes [12]. Large expenses in terms of time translate into high cost for business. For example, MioVision [33] has reported spending $7.7 million on collecting annotations for images from traffic cameras by 2015 [34]. Synthetic images could solve this problem since annotations for synthetic data are computer-generated and readily available.

At the same time, photo-realism draws synthetic images closer to real ones only to a certain extent, and there is always low-level and high-level appearance differences between real and synthetic data [35]. As a result, the performance of a model trained with synthetic data alone always lies behind the performance of the model when trained with sufficient

annotated real data. Our goal is to explore how synthetic training data effects the performance of the model and how it can be combined with real data in order to improve performance. Ultimately, we want to answer the questions: how we can reduce the amount of real training data while achieving the same performance, what are the characteristics of synthetic data that increase generalization abilities, and can synthetic data outperform real data in some cases?

Learning with domain adaptation. The problem of the performance drop when generalizing from one domain to another (i.e., from synthetic to real) is called domain gap and has received much attention lately boosted by the raising popularity of synthetic datasets simulating image taken from a car-mounted camera. The gap between domains can be narrowed by making synthetic images look more realistic by applying domain transfer algorithms [36, 37, 38, 39] based on Generative Adversarial Networks [40] to the generated synthetic images. However, the appearance nuances picked up by a domain transfer algorithm do not necessarily result in improved performance of a semantic task such as classification or segmentation because image features learned by a semantic classifier may not be related to ones learned by the domain transfer algorithm [41]. Instead, the gap between the domains can be narrowed using a set of semantic domain adaptation methods [42, 43, 44, 45, 46]. For the segmentation task, we use the Maximum Classifier Discrepancy for Unsupervised Domain Adaptation algorithm [46], which is described in Chapter 8.

Experiments with detection and segmentation. We analyze the effectiveness of the synthetic data separately for the car detection (Subsection 2.1.1) and for the car-by-car background segmentation (Subsection 2.1.2.) As described below, synthetic data is built differently for the detection and the segmentation tasks. However, in both cases, we train ML models on real data, on synthetic data, or on their combination. The models are

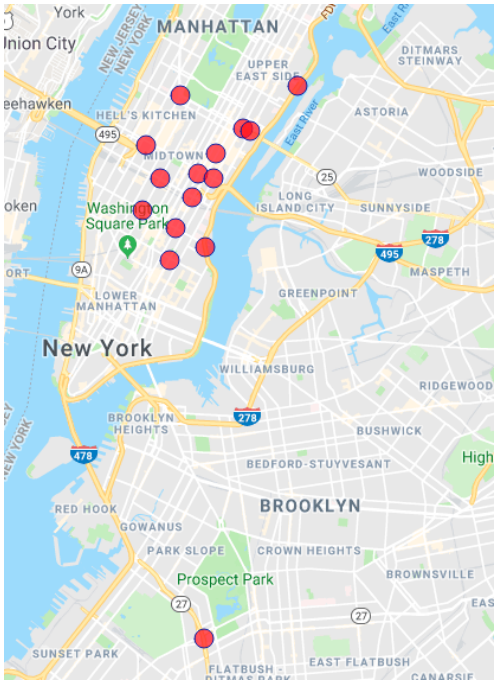


Figure 2.1: Geography of cameras that provide low-resolution low-framerate videos.

always evaluated on the test subset of the real data.

2.3 Image data

In this section, we first discuss the source of our real images and our approach to generate synthetic data. Then, we build on this discussion to show how we build the datasets for our detection and segmentation experiments.

2.3.1 Approach to real image data

Some existing datasets of traffic videos [10] were collected from high-resolution video cameras. The access to high-resolution video is generally restricted to the system owners, and algorithms trained on such datasets can be used by municipalities or their contractors (e.g., Pennsylvania [47]) or private companies (MioVision [33]) that own the cameras.

At the same time, many municipalities around the world provide public access to their

cameras via a web-interface, which can be used by anyone to inspect weather conditions or traffic congestion. The web-interface usually provides low framerate and low resolution video. Many examples of cameras that have open public access can be found on specialized aggregator web-sites [48, 49]. The diversity and the zero cost of these data are an attractive opportunity for research, while the poor image quality presents interesting challenges. We focus our research on low-resolution images.

We chose to use video from New York City cameras [1]. These cameras are used by the municipality to monitor road conditions, while the publicly available web-interface allows free access to the video. Currently, there are over $1,000$ cameras located in NYC. The interface provides a very low framerate, ranging from 0.3 to 1 frames per second. The frame resolution varies from 320×240 to 720×480 pixels. The cameras record a color image from dawn to dusk, and switch to the monochromatic mode at night. We only collected color data because of the poor quality of night images that does not allow to distinguish individual cars.

Videos from 15 cameras located in the Manhattan and Brooklyn boroughs of NYC (Figure 2.1) were collected and annotated by another member of our group, and constitute the CityCam dataset [50]. A total of $73,540$ images have been labelled with bounding boxes of cars, as well as their type and discretized orientation. The images were collected under different weather conditions and at different times of the day. This thesis uses only the labels of vehicles in the form of bounding boxes. We discuss how these labels are used to build a dataset for detection and segmentation tasks in subsections 2.3.3 and 2.3.4 below.

2.3.2 Approach to synthetic image data

Analysis of road video is popular in the context of autonomous driving. Here, data comes from multiple sensors, including car-mounted optical cameras. Highly diverse conditions

make it necessary to train and test self-driving systems on very large amounts of data. The problem can be addressed by employing simulated environments where one can generate plenty of data at no-cost. Methods of creating synthetic data include generating the whole virtual world in game development engines such as Unity [51, 52, 53] and reverse-engineering frame generation in computer games [54, 55]. In the fixed-camera setup, the road and scenario conditions are not as diverse as in autonomous driving, but still require prohibitively large amounts of labeled data.

These existing datasets are designed to model all the variety of conditions one can experience with car-mounted cameras. At the same time, surveillance cameras are more constrained, since, for example, the background does not change for a particular camera (though the background changes in appearance with weather and time of the day). Therefore, methods tuned to these specific conditions to generate a synthetic dataset are more applicable in this scenario, as described below.

Chapter 4 details the process of generating the synthetic dataset. First, we collect CADillac – a dataset of CAD models of cars from 3DWarehouse[1] – an online public repository for 3D CAD models. Next, the 3D models are rendered as images. Rendering for the detection and for the segmentation tasks differ and is further described in the next two subsections 2.3.3 and 2.3.4. In Chapter 4, we emphasize using public open source software, increasing photo-realism using generated background, and increasing the dataset diversity using weather patterns in rendering. Finally, we describe the released software to manage and render models from CADillac.

2.3.3 Building a dataset for detection

A dataset for object detection must include annotations in the form of bounding boxes for every object of interest in every image. In our case, the detection dataset includes the real

[1] `https://3dwarehouse.sketchup.com`

Figure 2.2: Examples of frames from the two cameras used in the detection task. Left: real frames, right: hybrid frames.

and the synthetic parts.

NYC dataset. For the real part of the dataset, we expanded CityCam to include annotated videos from two cameras in the Manhattan and in the Brooklyn boroughs of NYC. These videos are now a part of the CityCam dataset. The videos span the total of about 14 hours and include $4,000$ annotated frames. Examples of the frames are displayed in the top row of Figure 2.2. Because the framerate is so low (from 0.3 to 1 frames per second), the $4,000$ annotated frames constitute about 1.4 hours or 10% of the total number of frames. That is, the density of information per second is very small. Compare it with a video with a regular framerate of 24 frames per second. In that case, the same $4,000$ labelled frames would fit in only 2.5 minutes of video.

Building hybrid data. The synthetic counterpart of the videos from the two cameras was built using car models from the 3D CADillac dataset as described in Subsection 2.3.2. Because the camera view does not change, it is possible to take advantage of the real background in the video and to combine it with the synthetic renders of the car models. Such hybrid images are demonstrated in the bottom row of Figure 2.2, where computer-generated renders of vehicles are overlaid on the background image of frames. In Chapter 7, we will detail the process of building these hybrid images.

2.3.4 Building a dataset for segmentation

As discussed in Subsection 2.1.2, we perform car by car background segmentation in order to refine the knowledge about individual cars assuming the bounding boxes with cars have already been detected. The patches inside bounding boxes are resized to a 64×64 shape. Each patch displays a single vehicle. For training, the corresponding pixel-level background masks of the same size 64×64 are also required.

In this thesis, we present two twin datasets, Asterix and Obelix 2.3. Asterix is a dataset with 8K annotated *real* car images. Obelix is much bigger and contains 80K *synthetic* images. The two datasets are described in Chapter 6.

Building Asterix. Asterix is a real-image dataset built from images in CityCam images (Figure 2.2.) CityCam features images taken by 16 NYC cameras at different times of the day and weather conditions. Hence, images in Asterix differ significantly in appearance, and cars exhibit a variety of yaw and pitch angles. As opposed to UA-DETRAC [14], CompCars [10], or AI CITY [15], annotations in Asterix include pixel-level foreground masks, as well as pitch and yaw angles and car type. A total of $8,336$ annotated images constitute the dataset.

In order to build Asterix, we filtered vehicle bounding boxes in CityCam by size, ex-

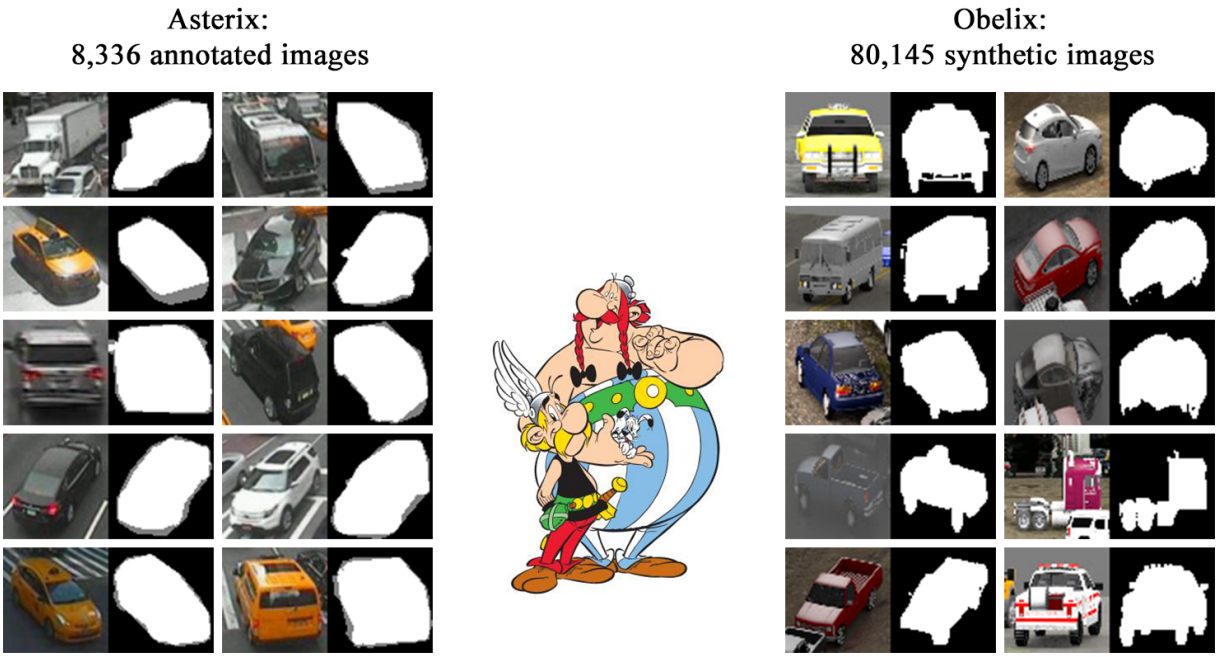


Figure 2.3: Examples of images from Asterix & Obelix datasets. Images are annotated with background mask, yaw, pitch, color, and type of cars.

panded them to include a portion of the background, cropped the image patches from the bounding boxes, and resized them to 64×64. Then additional annotations were added. First, camera orientation and position was reconstructed for all 16 cameras. That allowed us to label the pitch automatically for every car based on its position in the image. After that, we worked with a team of annotators to label the background mask, the yaw (car orientation in the ground plane), the color, and the type of each car. The process of building the dataset is detailed in Chapter 5.

Building Obelix The synthetic Obelix dataset was designed to represent real images in Asterix. That is, images are of the same size 64×64 and one car occupies the center of the image. As opposed to the case of the self-driving scenario ([51, 52, 53, 54, 55]) or to the task of detection from a traffic-camera (Subsection 2.3.3), there is not much background to be seen around the car in the center. Therefore, instead of using background with high-

level structure, we render the 3D models on top of two planes that are placed 90 degrees to one another. One plane is horizontal, it represents a road and has road texture from a library. The other plane is vertical, provides the view of the buildings in the background and is textured with a snapshot from Google Street View or images of houses. Chapter 4 discusses both the 3D CAD collection CADillac and the process of building Obelix dataset which is rendered from the CADillac collection.

2.4 Toolbox for dataset management

Finally, manipulating images and objects from the datasets is a nontrivial engineering task. We read and modify image annotations when we filter bounding boxes by size, border, or object properties such as "type" (as when building Asterix from CityCam [50]), when we transform polygons into image masks (as when making masks from the LabelMe [56] annotations), when we compare and merge labels produced by different annotators (as when annotating Asterix), when we plot and print dataset statistics (as when producing the plots for Chapter 6 of this dissertation), when we merge datasets (as when combining synthetic and real data for training), or when we evaluate trained models.

Each of these tasks is a small engineering problem that can be solved with a script, but chaining tasks, modifying scripts, and keeping them up to date requires a suitable format to store annotations. Commonly used file formats for storing annotations in computer-vision include `xml` (PASCAL2012 [25], UA-DETRAC [14]), `txt` (KITTI [57]), and `json` (Cityscapes [12], BDD [58]). As we discuss in Chapter 3, these formats are (1) not human-friendly on thousands of images, (2) take long to parse, and (3) each task requires programming business logic in the script.

In Chapter 3, we propose to store annotations in a relational database [59] and develop a database schema suitable for the tasks of classification, detection, segmentation, and

matching/tracking. We further present a command-line cross-platform tool named Shuffler that implements the schema as a SQLite3 [60] database and allows to perform all the operations listed in the beginning of this subsection (among others) with a few commands. Shuffler solves the three problems above. It (1) makes a dataset easy to examine by a human using any public SQLite3 reader or editor; (2) is fast to read and write data; and (3) eliminates coding most of the business logic in a script because it can usually be encoded in an SQL query.

Chapter 3

Shuffler: Data Manipulation Toolbox

3.1 Introduction

In the academic community, day-to-day work in the area of computer vision has emphasis primarily on algorithms rather than data. From this point of view, annotated image datasets should ideally be built once and remain fixed from then on. This approach allows the community to use datasets as benchmarks. Researchers choose to store these datasets in formats that are most common and fast to load for machine learning packages. However, from the point of view of a data scientist in industry, the same algorithm is applied to various partitions, modifications, and extensions of a dataset in hand in order to improve the performance in production (Figure 3.1.) In this case, a dataset is not considered static, but instead is constantly altered to fit the task at hand. That makes it necessary for multiple modifications of the same dataset to co-exist on disk or in a distributed system. Ideally a practitioner would want 1) a simple way to manipulate annotations and 2) a file format that allows to store multiple copies of the annotation set in an organized and efficient way and to inspect them manually.

Dataset manipulation tools are sometimes packaged with a dataset, but they typically allow to perform only a limited number of operations only on that particular dataset, and

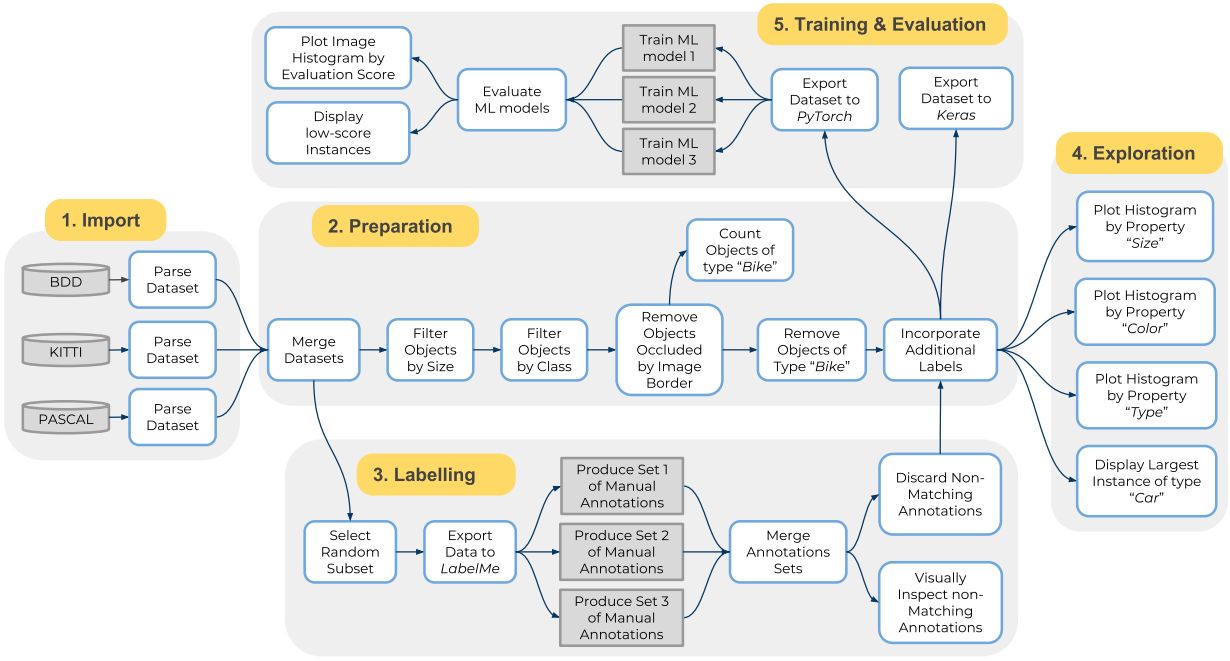


Figure 3.1: Simplified typical ML data workflow for object detection in computer vision. The rectangles represent operations and the arrows show the direction in which the data flows. Each operation produces new data and/or modifies the input data. The sample pipeline starts with the import of the BDD, KITTI, and PASCAL datasets. Data preparation activities and additional data labelling occur prior to data exploration and model training and evaluation. At any point in time, different parts of the pipeline could be executed sequentially or in parallel. The data management and software maintenance complexity is significant, and a specialized open-source data management system that simplifies operations (white rectangles) is necessary for practitioners.

Table 3.1: Raw annotation formats of popular object detection datasets are not designed for data manipulation

Dataset	Annotation file format
PASCAL2012 [25]	`xml` file per image
KITTI [57]	`txt` file per image
Cityscapes [12]	`json` file per image
BDD [58]	a single `json` file

often for a single programming language. An example is the PASCAL VOC dataset [25] that had 10 releases in different years and each release came with a Matlab toolbox for that specific year. Alternatively, researchers write small one-time scripts in-house to quickly alter a dataset, for example, change the size of object bounding boxes. The set of scripts created in this way contain duplicate code and gradually become difficult to maintain.

Datasets typically come in a custom format, which usually includes an image and a annotation file in one of the formats: `xml`, `txt`, or `json`. Table 3.1 presents an overview of several popular object detection datasets in the area of computer vision and the formats of the associated annotation files. On the one hand, these formats are human-readable, but on the other hand, slow to load. Additionally, changing annotations and saving them as a copy means duplicating the whole directory with the annotation files, which is inconvenient and slow. Many development kits cache annotations by serializing them with formats such as `pickle`[1] or `protobuf`[2]. Such formats are easy to load by a machine learning framework and convenient to store, but they are not interpretable by humans and can not be inspected or modified outside of a specialized programming environment.

To sum up, we consider (Figure 3.2) a typical data preparation workflow of a computer

[1] `https://docs.python.org/3/library/pickle.html`

[2] `https://developers.google.com/protocol-buffers/?hl=en`

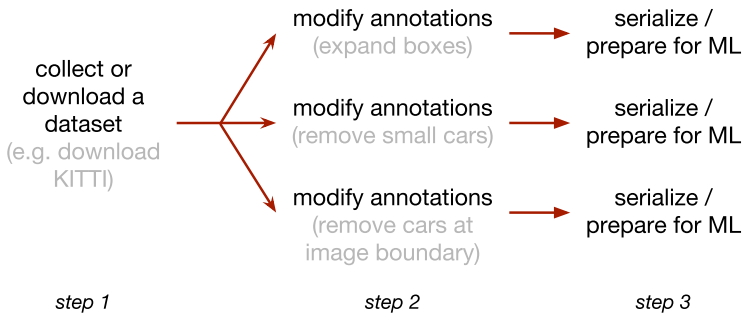


Figure 3.2: Typical data preparation pipeline lacks convenient tools on step 2

vision practitioner to consist of three steps: 1) download or collect a dataset, 2) modify annotations, and 3) serialize the dataset. We further consider a common situation when multiple modifications of annotations are used. Modifications could be a chain of trivial tasks, for example, removing objects at image boundaries and then increasing the size of bounding boxes. We note 1) the lack of software for manipulating annotations and 2) a convenient format to store such annotations.

In this work, we close this gap by proposing a format to store annotations and a software toolbox designed to manipulate them. The key idea is to employ the widely known relational databases and the associated SQL query language for storing and manipulating annotations. The proposed toolbox is essentially a wrapper around SQL and allows to chain multiple operations in a single command. Annotations are stored in an SQLite database with schema designed to cover the bulk of the common tasks in computer vision. The proposed solution satisfies the following properties:

- it has basic manipulation tools and allows to easily add new functions;
- annotations are fast to load and to modify and convenient to store;
- annotations are stored in a human readable format that can be manually edited;
- it is agnostic to the format of how images are stored on disk; and
- it supports image-level classification, object detection, semantic segmentation, and object matching tasks in computer vision.

The toolbox, the installation instructions, the manual, and use cases are available at `https://github.com/kukuruza/shuffler`.

3.2 Related work

Questions of data management have been studied in great detail since the early days of automated information processing. It proved convenient to store data using the Relational Model, where information is stored in a set of interconnected tables. Shortly after the Relational Model was proposed in 1970 [59], research was started on developing an appropriate language to handle it [61], until finally the International Organization for Standardization (ISO) accepted the Structured Query Language (SQL) as a standard in 1987. Since then the Relational Model has remained dominant in industry.

Typically, a computer vision dataset can be well described by the Relational Model since a machine learning algorithm is trained on samples of training data with the same structure. In particular, in the computer vision field, an image classification algorithm can be trained on pairs $\{\text{image}, \text{label}\}$. Pairs $\{\text{image}, \text{list of bounding box coordinates}\}$ are used to train for the object detection task. Semantic pixel-wise image segmentation is trained on pairs $\{\text{image}, \text{label map}\}$, where label map is of the same size as the image.

However, the Relational Model is not popular in the modern computer vision field for several reasons. First, the Relational Model does not fit well in the academic workflow. For distributing a dataset, researchers choose universally known, human readable formats (Table 3.1). For training, the dataset can be serialized as files in Google `protobuf` format or similar in order to minimize the time to read the data from disk. In this work, we argue that a Relational Model is needed not for dataset distribution or for data serialization, but rather for the intermediate step of modifying and filtering annotations, extracting a subset of the dataset, visualizing information, and so on.

Second, different datasets define different information they need to store. For example, angles are important for objects in images from traffic surveillance cameras but are irrelevant for generic images such as in the Pascal VOC dataset [25]. That makes any particular schema hard to generalize across datasets. Instead of providing a universal schema, we focus on setting up a base that can be customized for particular needs of a particular company or group. At the same time, our design of the schema is generic enough to fit the tasks of image-level classification, object detection, semantic segmentation, and object matching, which covers a big part of the computer vision landscape.

Moreover, some data entries in a dataset may contain additional information that is not included with other data entries. Such datasets are better described by the flexible schema and the Non-Relational Model ("NoSQL" databases). However, using it for the backend of the toolbox would greatly increase the code complexity and may make the database non human-readable. Therefore, we chose to use the Relational Model and we support non-standard properties of data entries (such as the aforementioned "angle") by introducing the `properties` table into the schema (Figure 3.3.)

Finally, the Relational Model needs to be implemented with one of the many database management systems. Examples include SQLite [60], MySQL [62], or MariaDB [63] engines that are available under a GPL license, besides many proprietary systems. Database management systems for industry-level scales of data include a service that runs in the system background as a daemon. That would complicate the design of our toolbox. On the other hand, the light-weight SQLite engine does not need a daemon and allows for sufficiently fast processing: most time-consuming operations took seconds to complete on 1 million entries. It is important to note that the same operations would take dozens of minutes when implemented as operations on files of `txt`, `xml`, and `json` formats and run on the same machine.

The rest of the chapter is organized as follows. Section 3.3 describes the schema of the

SQL database, the covered use cases, and the limitations of the schema. Section 3.4 and 3.5 present the Shuffler toolbox and the operations it supports. This is further explored in Section 3.6, that illustrates an important feature – chaining multiple operations in a single command. We conclude the chapter in Section 3.9.

3.3 Database schema

The core of this work is the proposed Relational Model for common datasets that were collected to train for image classification [13, 64], object detection [13, 25, 56, 57], semantic segmentation [13, 58, 65, 66], and object matching tasks [67]. The proposed schema is presented in Figure 3.3 in the form of an Entity Relation (ER) diagram. The diagram presents five tables. Each table consists of several fields. Two fields in any table are mandatory to be filled in: the unique Primary Key (PK in bold font) and the Foreign Key (FK in italic font).

In this section, we describe a typical dataset in the computer vision field and show how it corresponds to this schema. An example is provided in Figure 3.4. It illustrates a traffic camera dataset and is focused on the tasks of object detection and matching.

Typically, a computer vision dataset consists of a number of images. Each image corresponds to an entry in `images` table and is uniquely defined by the `imagefile` field. Normally, this field contains the path to the image, but may also have other descriptors, such as the frame in a video. In the case when the dataset is focused on the image classification task, each image is associated with a label, such as "cat" or "dog." The label is recorded in the `name` field of the `images` table. In the case when the dataset focuses on the semantic segmentation task, each image comes with a segmentation map. In this schema, the segmentation map descriptor is recorded in the `maskfile` field of the table `images`.

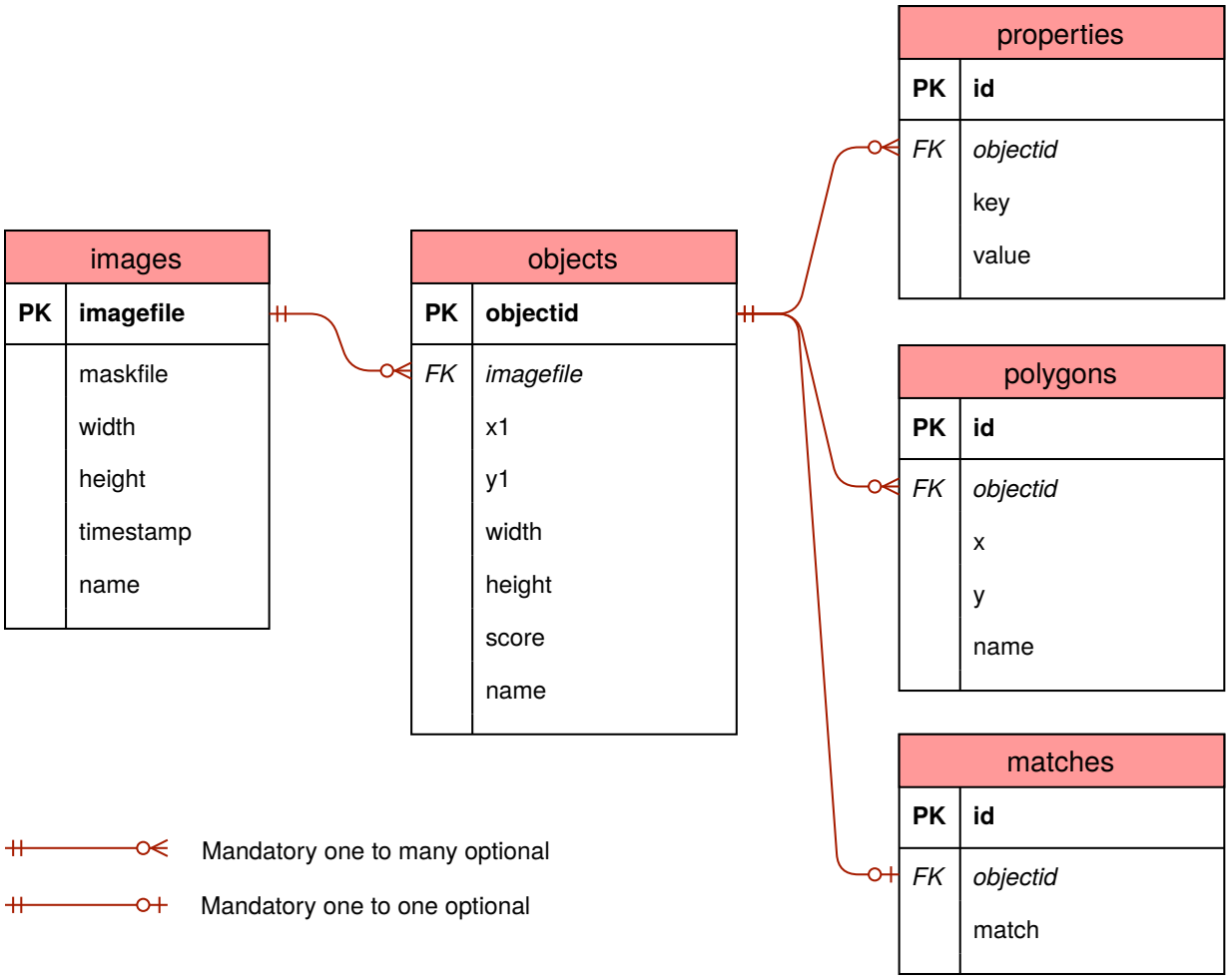


Figure 3.3: Proposed database schema

1.jpg

2.jpg

properties			
id	objectid	key	value
1	3	color	black
2	1	yaw	60
3	2	yaw	60
4	3	yaw	60

polygons			
id	objectid	x	y
1	1	69	68
2	1	351	63
3	1	351	142
4	1	64	183
5	2	1	97
6	2	114	86
7	2	128	173
8	2	1	195

images
imagefile
1.jpg
2.jpg

objects						
objectid	imagefile	x1	y1	width	height	name
1	1.jpg					bus
2	2.jpg					bus
3	2.jpg	133	129	188	96	car

matches		
id	objectid	match
1	1	1
2	2	1

Figure 3.4: A database populated according to the schema in Figure 3.3. Only non-empty columns are shown. The same bus is recorded under `objectid=1` and `objectid=2` in the two images and is matched via `match=1`. The two bus objects are assigned a bounding polygon, while the car is assigned a bounding box. The polygons and the box are overlaid on the image.

A dataset may contain multiple objects in each image. In this schema, each object corresponds to one entry in table `objects` and has the unique `objectid`. Each image may contain zero, one, or multiple objects, while any object must belong to some image. That is encoded as "mandatory one to many optional" relations between `images` and `objects` tables. In the object detection setup, each object is characterized by its bounding box, which is encoded as $\{\texttt{x}, \texttt{y}, \texttt{width}, \texttt{height}\}$ in table `objects`. Besides, each object typically belongs to a single class out of the pre-defined set of classes, for example, "car" and "truck" in Figure 3.4. The class name, if present, is to be encoded in the `name` field of table `objects`.

Furthermore, an object may have auxiliary properties. For example, some cars in Figure 3.4 have the assigned color and the two angles, yaw and pitch, as seen by the camera. This extra information is recorded in the table `properties`, which is linked to the table `objects`. Any object may have any number of different properties, thus this schema is applicable to datasets with unstructured information on their objects.

Some datasets, such as LabelMe [56], provide fine-grained boundaries of their objects using polygons instead of rectangular bounding boxes. Our proposed schema supports this via table `polygons`. The rows in `polygons` table with the same value of `objectid` field describe the points in a closed polygon. Thus the order of their `id`-s matters: the points should be recorded either clockwise or counter-clockwise. It may happen that two polygons correspond to the same object. Then they should be differentiated from each other with the `name` field.

Finally, the task of object matching is supported with table `matches`. The matched objects are recorded as separate entries in the `matches` table, and, as such, they have different `id` and `objectid`. However, they share the same value of the `match` field. That value uniquely identifies these particular matched objects among other matches. This idea is illustrated in Figure 3.4.

The proposed schema comes with certain limitations. For example, it does not support the increasingly popular format of video clips [58, 66]. It also does not inherently support 3D bounding boxes of objects [57, 66], though they can be encoded with our schema via the `properties` table. In both cases the schema can be trivially extended, but such extension is beyond the scope of this work.

3.4 Toolbox

The SQL schema alone would be useless without the tools that take advantage of it. We developed a toolbox that allows to 1) import annotations from other formats, 2) save annotations as an SQLite database, 3) modify them, and 4) export them into other formats.

A user interacts with the toolbox by calling the program `shuffler.py` from the command line. In a minimal working example below, Shuffler creates a new database and prints information about it to the standard output:

```
$ ./shuffler.py printInfo

[shuffler.py:62 INFO]: will create a temporary database in memory.
=== Running printInfo ===
{'num objects': 0, 'num images': 0}
```

In this example, Shuffler calls the sub-command `printInfo`. In general, all the work with `Shuffler` is performed via sub-commands. The example below illustrates how the sub-command `importKITTI` and its command-line arguments `--images_dir` and `--detection_dir` is used to import annotations from the KITTI dataset [57]. It is assumed that KITTI has been downloaded and is located in the directory `KITTI`.

```
$ ./shuffler.py importKitti
    --images_dir='KITTI/data_object_image_2/training/image_2' \
    --detection_dir='KITTI/data_object_image_2/training/label_2'
```

```
[shuffler.py:62 INFO]: will create a temporary database in memory.
=== Running importKitti ===
100% (7481 of 7481) |#################| Elapsed Time: 0:00:16 Time:  0:00:16
```

In this example, the database is created in-memory and is never recorded to the hard-drive. Loading and saving databases is controlled by the two command-line arguments: `-i` and `-o`. For example, KITTI annotations for the object detection task can be imported and then recorded as `kitti.db`:

```
$ ./shuffler.py  -o='kitti.db'  importKitti
    --images_dir='KITTI/data_object_image_2/training/image_2' \
    --detection_dir='KITTI/data_object_image_2/training/label_2'

[shuffler.py:36 INFO]: will create database at kitti.db
=== Running importKitti ===
100% (7481 of 7481) |#################| Elapsed Time: 0:00:17 Time:  0:00:17
[shuffler.py:122 INFO]: Committed.
```

The next example shows how to load the recorded `kitti.db` and print basic information about it:

```
$ ./shuffler.py  -i='kitti.db'  printInfo

[shuffler.py:50 INFO]: will load database from kitti.db, will not commit.
=== Running printInfo ===
{'image height': '4 different values',
 'image width': '4 different values',
 'matches': 0,
 'num images': 7481,
 'num masks': 0,
 'num objects': 51865,
 'properties': ['alpha', 'dim_height', 'dim_length', 'dim_width', 'loc_x',
    'loc_y', 'loc_z', 'occluded', 'rotation_y', 'truncated']}
```

Table 3.2: Different combinations of the arguments `-i` and `-o` and their meanings

–	–	Create a new database in-memory. Discard it at the end.
`-i in.db`	–	Open `in.db` in read-only mode.
–	`-o out.db`	Create a new `out.db` and commit transactions there.
`-i in.db`	`-o out.db`	Open `in.db` but commit transactions to `out.db`. Backup `out.db` if it already exists.

Finally, when both `-i` and `-o` are specified, a database is loaded, modified, and saved under a different name:

```
$ ./shuffler.py  -i='kitti.db'  -o='clean.db'  filterObjectsAtBorder

[shuffler.py:44 INFO]: will copy database from kitti.db to clean.db.
=== Running filterObjectsAtBorder ===
100% (7481 of 7481) |################| Elapsed Time: 0:00:03 Time:  0:00:03
[dbFilter.py:146 INFO]: Deleted 6966 out of 51865 objects.
[shuffler.py:122 INFO]: Committed.
```

The effects of all the combinations of `-i` and `-o` command-line arguments are summarized in Table 3.2. Finally, for completeness, we present Shuffler interface in Listing 1. In the next section, we focus on individual sub-commands.

3.5 Sub-commands

Sub-commands is the workhorse of Shuffler. Their complete list is presented in the Appendex but can be printed out with:

```
$ ./shuffler.py -h
```

Besides the global command-line arguments, each sub-command defines its own arguments. One can get help on an individual sub-command and its arguments like in the

Code Snippet 1 Shuffler interface. Input/output is controlled by `-i/-o`, `--relpath` determines the root path that all `imagefile`-s are relative to, `logging` controls the verbosity of the output, `-h` prints out more information about arguments.

```
Shuffler.py [-i IN_DB_FILE] [-o OUT_DB_FILE]
            [--relpath RELPATH]
            [--logging {10,20,30,40}]
            [-h]
            sub-command-1 [sub-arguments-1]
            [sub-command-2 [sub-arguments-2] ...]
```

example below:

```
$ ./shuffler.py printInfo -h
usage: shuffler.py printInfo [-h] [--imagedirs] [--imagerange]
Sum up and print out information in the database.
optional arguments:
  -h, --help          show this help message and exit
  --images_by_dir     print image statistics by directory
  --objects_by_image  print object statistics by directory
```

The sub-commands can be divided into several major groups:

1. "Import" group allows to add annotations from datasets of different formats to a new or existing database. At the moment, `importLabelme`, `importKitti`, and `importPascalVoc2012` are implemented.
2. "Filtering" group serves to remove images, objects, or matches from the database according to some criteria. Two self-explanatory examples are `filterEmptyImages` and `filterObjectsAtBorder`. A call to `filterObjectsByIntersection` removes objects that intersect with others beyond a threshold. `filterObjectsSQL` is a more flexible tool that filters out images or objects based on an SQL query. For

example:

```
$ ./shuffler.py  -i='my.db' \
    filterObjectsSQL --where_object='width < 64 AND name = "car"'
```

Under the hood, this sub-command opens `my.db` and run the `DELETE` SQL query on its tables. Its simplified version for the `objects` table may look like this:

```
DELETE FROM objects WHERE width < 64 AND name = "car"
```

3. "Modifying" group changes entries in a database. For example, `expandBoxes` expands bounding boxes from each side, `addDatabase` merges another database with the open one, `splitDatabase` on the contrary splits the database into several parts (for example, into the "train", "test", and "validation" sets), `polygonsToBoxes` computes a bounding box for each polygon. Sub-commands in this group have unique meaning and serve various purposes. It is worth noting that all operations are performed on the database while images on disk are not modified or filtered in any way.

4. "Info" group prints aggregated information or dumps a part of the database, and creates different types of plots using `matplotlib` package [68]. For example, a histogram of angles from the "properties" table can be plotted as shown below:

```
$ ./shuffler.py -i='my.db'  plotObjectsHistogram \
    --sql_query 'SELECT value FROM properties WHERE key="angle"'
```

5. "GUI" group provides a graphical interface for browsing a dataset or modifying it. A user may loop through images with bounding boxes or polygons overlaid on them using `display`, assign or change object names using `examineObjects`, and view or change matches with `examineMatches`. For example, the command below iterates over images in a random order and displays them and all the objects. We use OpenCV [69] as the backend for the graphical interface.

```
$ ./shuffler.py  -i='my.db'  displayImages --shuffle --show_objects
```

6. "Evaluate" group assumes that the open database contains predictions made by a machine learning algorithm. The sub-commands evaluate these predictions with respect to another database, which contains the ground truth. Currently, evaluations of the object detection the semantic segmentation tasks are supported. Below predictions are evaluated for the machine learning tasks.

```
$ ./shuffler.py  -i='predictions.db' \
    evaluateDetection --gt_db_file='ground_truth.db'
```

7. "Export" group saves annotations in one of the formats that allows to use the dataset for training.

   Section 3.7 describes an alternative way to load data in Keras [17] and PyTorch [16] using the provided data generator classes.

3.6 Chaining operations

One main motivation for the toolbox design was the ability to conveniently chain operations. An example is shown in listing 2. Commands are chained via the vertical bar symbol "`|`", that must be escaped in a Unix shell as `\|`, `'|'`, or `"|"`. The program exports bounding boxes of cars into a new dataset that will be further sent to LabelMe annotators. First, expand bounding boxes by 20% from every side. Then filter those cases that intersect with other cars by more than by 30%, those at the image border, those with bounding boxes of size less than 64 pixels in either dimension, and those with names other than "van," "taxi," or "sedan." Then the bounding boxes are cropped, scaled to 64×64, and written to a new dataset – a database with a video. Note the five chained calls: `expandBoxes`, `filterObjectsByIntersection`, `filterObjectsAtBorder`, `filterObjectsSql`, and finally `cropObjects`. Without chaining, one would have to store intermediate results

– one after each operation.

Code Snippet 2 Examples of chaining operations. The program crops out objects into a directory `crops` that will be further sent to LabelMe annotators. Note the chained calls to `expandBoxes`, `filterObjectsByIntersection`, `filterObjectsAtBorder`, `filterObjectsSql`, and `cropObjects`.

```
#!/bin/bash
./shuffler.py -i='in.db' \
    expandBoxes --expand_perc=0.2 \| \
    filterObjectsByIntersection --intersection_thresh_perc=0.3 \| \
    filterObjectsAtBorder \| \
    filterObjectsSQL --where_object='width < 64 AND name="car"' \| \
    cropObjects --edges=distort --target_width=64 --target_height=64 \
      --image_pictures_dir='crops'
```

3.7 Interface to Keras and PyTorch

Apart from the functionality of `shuffler.py`, the toolbox also provides support for loading data in PyTorch [16] and Keras [17] directly from a database. Keras allows to use a custom `DataGenerator` class that loads data by batch. At the same time, a custom `ImagesDataset` class in PyTorch can be used to load individual items, which are further collected into batches by PyTorch's native `DataLoader`.

File `lib/interfaceKeras.py` provides a custom `DataGenerator` class, which can load data for the tasks of image classification, semantic segmentation, and object detection. The advantage of this class is the ability it gives a user to choose data entries from the database with `where_images` and `where_objects` arguments. Similarly, file `lib/interfacePyTorch.py` contains a class inherited from the PyTorch

`ImagesDataset`. Arguments `where_images` and `where_objects` can be used in the same way to use only a subset of the dataset. An example of using this class for semantic segmentation is shown in Listing 3.

Code Snippet 3 We provide the class `ImagesDataset` used to load data in PyTorch

```
import torch.utils.data
import shuffler.src.interfacePytorch
# Create an object of Dataset associated with 'my.db'.
dataset = shuffler.src.interfacePytorch.ImagesDataset (db_file='my.db')
# Get one item from the dataset.
image, mask = next(iter(dataset))
# The standard way to load data in PyTorch.
loader = torch.utils.data.DataLoader (dataset, batch_size=10, shuffle=True)
```

3.8 Implementation details

The program Shuffler is implemented as a Python script, which can call one of many functions. For example, imagine a user typed:

```
./shuffler.py -i my.db filterObjectsAtBorder --border_thresh_perc 0.01
```

Shuffler will parse the first command line arguments "`-i my.db`", open the database `my.db`, and get the "cursor" that allows to send queries to the database. It then will find a subparser for `filterObjectsAtBorder`. The subparser will parse the remaining argument "`--border_thresh_perc 0.01`." This functionality is implemented via the `argparse` package. Then Shuffler will call the function `filterObjectsAtBorder` passing it the cursor and the parsed arguments.

Shuffler analyzes command line arguments sequentially and executes the sub-commands when it runs across them in the command line. That allows to chain sub-commands inside

a single call to `shuffler.py`. The database is opened or created by Shuffler in the beginning according to rules 3.2. The database cursor is passed to each function that is called by Shuffler. Therefore, each function has the possibility to perform transactions and introduce changes to the database. These changes may or may not be committed by Shuffler in the end, depending on whether the arguments `-o out.db` were passed to Shuffler. By convention, no function has the right to commit changes itself. That allows to run Shuffler in "read-only" mode and to interrupt the program without making changes.

All functions share the same interface:

```
def myFunction(cursor, args)
```

where `cursor` is an SQLite3 cursor for the open database, and `args` is the named namespace with parsed arguments for `myFunction`. That makes adding sub-commands straightforward. To add a new function and its associated sub-command, one first needs to pick a file where the function would fit the best based on its functionality. For example, all operations of filtering reside in `dbFilter.py`. Then one needs to 1) write its body, which implements the interface, 2) write the parser, and 3) register the parser in the `add_parsers` function. Listing 4 provides the skeleton for function `myFilter`.

3.9 Conclusion

Imagine a computer vision practitioner training a vehicle detector, for example, on the KITTI [57] dataset for an autonomous vehicle. As such, he/she wants to remove all labels except for "Car," "Van," "Truck," and "Tram," then to experiment if the bounding boxes around objects need to be expanded, if the objects on the image boundary should be removed, and whether it is better to also remove small objects. Furthermore, he/she would like to quickly see the distribution of objects by class and by size in the dataset. All

in all, the researcher is using the workflow depicted in Figure 3.2.

Annotations in the KITTI dataset come as text files, one per each image (Table 3.1.) Making each of the dataset modifications described above requires the researcher to use KITTI's toolbox to load the data, write the custom code to filter/modify annotations, and write the annotations as the new set of text files, while somehow bookkeeping the paths to each annotation set.

The tool Shuffler that we developed allows to perform all these operations out of the box. Each of the original and modified annotations will be stored as a SQLite database file. The databases can be exported back into the KITTI format. Alternatively, a researcher can directly use the provided `DataGenerator` class in Keras or `ImagesDataset` class in PyTorch during training and testing. The toolbox is made public at `https://github.com/kukuruza/shuffler`. The detailed usage instructions are a part of the repository.

In more general terms, we consider the workflow of the ML expert in the computer vision domain. Multiple modifications of the dataset are an important part of this workflow. No tool or data representation is specifically designed to address this problem. In this thesis, we specifically address this gap.

The design of our toolbox was motivated by the fact that data used in the Computer Vision field generally fits well the Relational Model.

The data model or the toolbox that we presented in this chapter do not target the questions of convenient distributing the dataset for public use or the efficient data storage for fast loading by machine learning packages. Instead, we have focused on the issues raised by of working with annotations between the stages of downloading/collecting the dataset and training a ML model.

Code Snippet 4 Adding a new sub-command `myFilter` to Shuffler. File `dbFilter.py`

```
# Function add_parsers already exists. Register your new parser in there.
def add_parsers(subparsers):
  < ... >
  myFilterParser(subparsers)


# Write the parser for command-line arguments.
def myFilterParser(subparsers):
  parser = subparsers.add_parser('myFilter', description='My new operation')
  parser.set_defaults(func=myFilter)
  parser.add_argument('--mandatory_arg', required=True, type=string)
  parser.add_argument('--extra_arg', required=False, type=int, default=1)
  < ... >


# Write the implementation of your function.
def myFilter (c, args):
  ''' The implementation of sub-command myFilter.
  Args:
    c:     SQLite3 cursor
    args:  Command-line arguments parsed according to myFilterParser
  Returns:
    None
  '''
  < ... >
```

Chapter 4

Synthetic Data: Building Obelix

Figure 4.1: Examples of taxi model renders. Left: cloudy weather; center: sunny weather; right: wet asphalt.

4.1 Introduction

The domain of road traffic analysis from traffic surveillance cameras is related to visual perception in the self-driving car research area. I will now discuss general tasks and data-related challenges of the self-driving field and connect them to the domain of road traffic analysis.

In recent years, much attention has been focused on developing effective perception algorithms for self-driving cars. The perception component of the self-driving technology usually includes processing data from a LIDAR and visual camera(s). Perception from

visual cameras includes detection of objects in the picture, such as construction cones or traffic signs, recognition of particular traffic signs, segmentation of the driveable area in the image, depth estimation, etc. Previous generations of algorithms used geometry information and machine learning with manually crafted features to solve some of these tasks, and have found their way to market as a part of successful products such as Mobileye[1].

In the past few years, rapid development of deep learning algorithms and new powerful and energy-efficient GPU cards allowed processing video on cars using neural networks, achieving significant improvement in performance on the aforementioned tasks. Deep learning methodology often includes two steps. During the first "training" step, the weights of models are learned using collected and curated image data. During the second "inference" step, the model is used to make predictions in real time on the incoming image data. Many top-performing algorithms are trained in the supervised way. That means that performing the training step requires ground truth annotations for each image in the collected training dataset. In particular, if a deep learner plans to train on image classification, he or she must collect annotations in the form of a label for every image. If a researcher plans to train a network for the task of object detection, he or she must collect annotations in the form of bounding boxes and the corresponding label for every relevant object in every image. Finally, if a researcher is planning to train for the task of semantic segmentation, he or she must collect a pixel-wise annotated mask for every image. The time for annotating one image increases when moving from image classification to object detection and to semantic segmentation. At the same time, the number of images required to train for a semantic segmentation task can reach thousands to millions of images.

There exist a number of publicly available large datasets, specifically collected for training and evaluating perception algorithms for car-mounted cameras (figure 4.2). CamVid [11] dataset contains 701 annotated image. Cityscrapes [12] currently has 25,000 images with

[1] `https://www.mobileye.com`

Figure 4.2: Examples of images and annotations from popular car-centric datasets.

pixel-level and polygon-level annotations. KITTI [57] contains 15,000 images, among them 400 images with pixel-level annotations. Recent ApolloScape [66] contains 140,000 images with pixel-wise annotations and BDD100K [58] has 100,000 images annotated with bounding boxes and 10,000 images annotated with pixel-wise masks. Mapillary [65] has 25,000 images with pixel-wise masks and object-level annotations.

These dataset provided a large enough base for training a model for research tasks and to compare performance of different algorithms. However, using these datasets to train a self-driving car to drive on real roads is limited by several factors. First of all, some datasets present limited geographical variety. For example, Cityscrapes includes images from roads of 50 cities in Germany and nearby countries, ApolloScape was collected exclusively in China, and BDD100K dataset – in USA. Even though roads and cars look similar in different countries, some details may not generalize from one dataset to another. An example is traffic signs in different countries, school buses, and even road lane markings. Moreover, driving conditions even in the same geographical locations present a vast amount of corner cases because of a variety of driving and weather conditions, as well as the vehicles and objects that can be seen on a street, etc. At the same time, the performance of a self-driving car in all corner cases is crucial. Even the largest public datasets contain only hundreds of thousands of images and can not cover all the corner cases reliably. Therefore, large industry players in the area of self-driving had to collect their own datasets consisting of extremely large number of annotated images tailored to their own cameras, camera mounts, and their cars, as well as the geography of the operation. These collected datasets remain private and are regarded as commercial assets of the high value.

In the meantime, in order to compensate for the lack of labelled data, there exists several datasets consisting only of synthetic images that simulate images from car-mounted cameras (figure 4.3). Notable examples include SYNTHIA [52], GTA5 [54], Virtual KITTI [53]. These datasets simulate diverse driving environments and weather conditions, and aim to

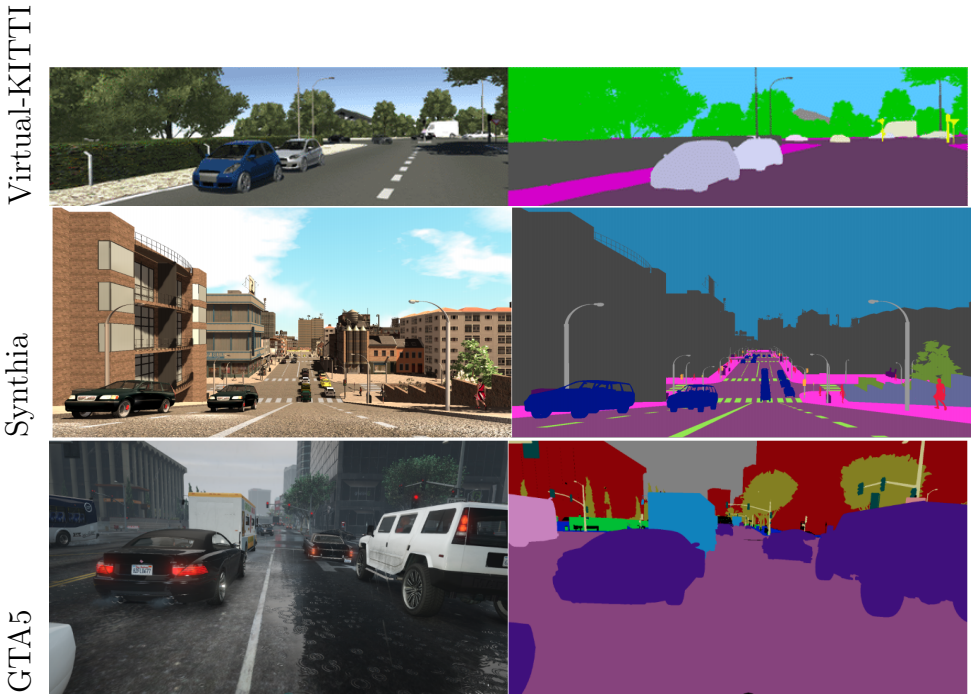


Figure 4.3: Examples from popular car-centric *synthetic* datasets.

augment or replace real datasets. The main advantage of synthetic datasets is the possibility of free virtually unlimited data. Additionally, it becomes possible to simulate different varieties of the same rare corner case. Unfortunately, models trained on synthetic data do not exhibit the same performance as models trained on real data due to domain gap between a synthetic and a real dataset, no matter how photo-realistic the synthetic dataset is [35]. The problem of domain gap is addressed with a number of techniques under the common name Domain Adaptation and is discussed in detail in chapter 8.

Deep Learning can also be used for traffic analysis with images from traffic surveillance cameras. In this setup, annotations come in the same formats as in the self-driving domain – image-level labels, object-level bounding boxes, and pixel-level semantic segmentation masks. Just the same, the annotated data is also hard to collect. Therefore, it makes sense to employ synthetic datasets for our traffic surveillance setup.

SYNTHIA, GTA5, and Virtual KITTI datasets [52, 53, 54] render the environment as

seen by a car-mounted camera, while in our application of traffic analysis we use images from surveillance cameras. Cars in these two setups look very different, hence, these datasets can not be used without changes. GTA5 [54] is generated by reverse engineering frames from a video game Grand Theft Auto V and, therefore, cannot be possibly turned into a bird-eye viewpoint dataset. Synthia [52] is generated by building a virtual city using the game engine Unity. Changing Synthia generation scripts would allow to move the camera and create a synthetic dataset from a traffic camera perspective. However, 3D assets of Synthia are not available for public use. CARLA [70] open source virtual world simulator build with the Unreal Engine offers a limited set of 3D models and no option for rendering on real background. Our collection of 3D models augment CARLA rather than competing with it.

We created a synthetic dataset by rendering accurate high-resolution models of cars of different makes and models. The emphasis is to render a variety of existing vehicles and to provide diverse environment conditions. The dataset is generated in two steps:

- Collecting 3D CAD models from publicly available resources into a novel 3D dataset, which we named **CADillac**.
- Rendering 3D models superimposed to random backgrounds *or* real images as viewed by a known traffic camera. We call the produced dataset **Obelix**.

The rest of this chapter describes the process of generating CADillac and Obelix in detail. The Obelix dataset is further described in Chapter 6. Additionally, Chapter 7 introduces a dataset of hybrid images, made of synthetic CAD models rendered on top of real background.

4.2 CADillac: Dataset of 3D CAD models

The core of the dataset is the 3D CAD models of vehicles that comprise the dataset that we named CADillac. We collected CAD vehicle models at 3DWarehouse[2] sharing platform specialized at 3D CAD models. 3DWarehouse generously granted us the permission to use the data for the project. Models are contributed by individual artists and are accompanied by the name and the description. Fortunately, there are many detailed and high-quality models of vehicles, often organized as collections. We download individual collections automatically using the Selenium[3] web-browser automation tool.

After the models were downloaded, we manually classified them by type (passenger, bus, truck, van, and bike), by domain (general, emergency, military, and fiction), and by color (white, black, gray, red, yellow, blue, green, and mixed). Additionally, we manually extracted the information about car make, model, and year from the original model name and description, when such information is available. Two examples of typical names are presented below:

$$\begin{aligned} \text{name} = \texttt{"F.D.N.Y. Ford Crown Vic"} \quad &\longrightarrow \quad \text{make} = \texttt{Ford} \\ &\qquad\quad \text{model} = \texttt{Crown Victoria} \\ &\qquad\quad \text{year} = \texttt{N/A} \\ \text{name} = \texttt{"HAZMAT SUPPORT UNIT F-350 2005"} \quad &\longrightarrow \quad \text{make} = \texttt{Ford} \\ &\qquad\quad \text{model} = \texttt{F-350} \\ &\qquad\quad \text{year} = \texttt{2005} \end{aligned}$$

Using this information, we further searched for physical dimensions of the models using the CarQueryApi database[4]. In particular, we retrieved vehicle length, width, height, and wheel base. We found the dimensions of the roughly 70% of the models in the dataset,

[2] https://3dwarehouse.sketchup.com

[3] https://www.selenium.org

[4] http://www.carqueryapi.com

Figure 4.4: Examples of models returned by SQL queries from listing 4.1. Top: models 9 to 10 meters long, center: white Fords, bottom: Toyota trucks. The grid size is 1 meter.

Listing 4.1: Example SQL queries to CAD database

```
SELECT model_id FROM cad WHERE dims_L >= 9 AND dims_L <= 10
SELECT model_id FROM cad WHERE make = "Ford" AND color = "white"
SELECT model_id FROM cad WHERE make = "Toyota" AND type = "truck"
```

while the missing 30% either do not have a proper description, or are missing from the CarQueryApi database.

After that, CAD models were post-processed. Each model was oriented along the X-axis, centered, and scaled to match the information from CarQueryApi. We found that the wheel base is the most robust property to match, since length, width, and height of a CAD model may be affected by bumper bars, side mirrors, taxi checkers or emergency light bar, and so on.

All the information about a mode is stored in a relational database. This choice makes it convenient for the end user to query the dataset using the standard SQL syntax. For example, the first of the queries in listing 4.1 retrieves all 9 and 10 meters long cars, the second one – all white Fords, and the last one – all Toyota trucks. The model that the queries returned are presented in figure 4.4.

CAD models originally come in format `skp`, which is the native format of Sketchup[5] 3D modelling software. They are converted to Blender [71] native format `blend`. Due to partial incompatibility of the two formats, some models get triangular artifacts on some surfaces, while others get matte gray artificial looking glass surfaces (figure 4.5). The latter is admittedly not very noticeable. Instead of disposing of them, we run ablation studies of employed visual domain adaptation algorithm on them to determine the kind of appearance differences that affect domain adaptation. These appearance problems are recorded in the database and can be retrieved with an SQL request (see listing 4.2).

[5] `https://www.sketchup.com`

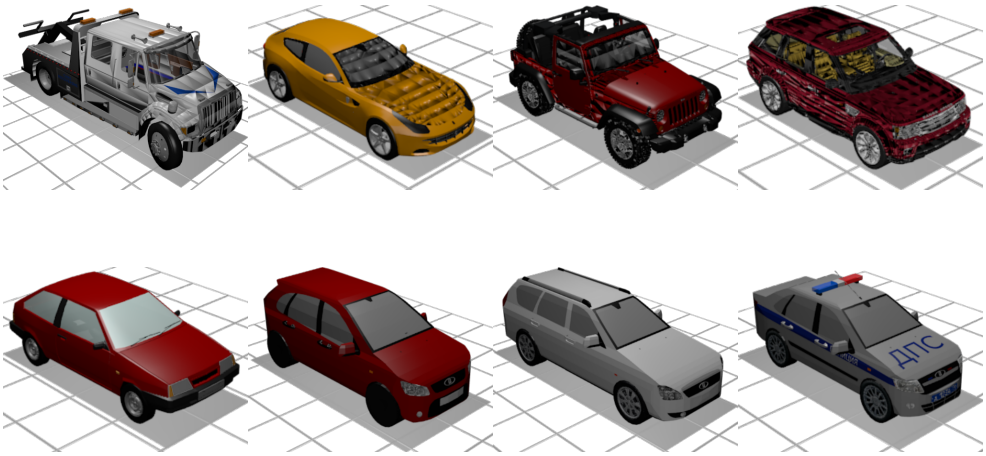


Figure 4.5: Examples of broken models for ablation study. Top: "triangulation" issue, bottom: matte glass material issue. Models are returned by the first two SQL queries from listing 4.2. The grid size is 1 meter.

Listing 4.2: Some models have visual problems and can be used for ablation studies

```
SELECT model_id FROM cad WHERE error = "triangles"
SELECT model_id FROM cad WHERE error = "matte glass"
SELECT model_id FROM cad WHERE error IS NULL
```

4.3 Obelix dataset: Rendering photo-realistic images.

The next step is to render 3D models as images. We use the Blender [71] rendering engine for photo-realistic rendering. The advantages of Blender are it being open-source and its support for scripting in Python. Some other synthetic datasets [72, 73] also use Blender, while SYNTHIA and Virtual KITTI uses Unity, and UnrealCV [74] uses Unreal Engine. GTA5 and Driving In the Matrix [75] datasets are produced by extracting frames from the Grand Theft Auto V video game which does not involve rendering.

Figure 4.6 explains the process step by step. First, we randomly pick the main vehicle from the 3D dataset and place it in the center of a 3D scene. Then we pick and place a number other vehicles around to simulate occlusions. All vehicles are heading in the same direction, but they are slightly tilted one with respect to another. The vehicles are placed close to each other, but they do not intersect. The background is simulated by two textured planes. The first plane imitates the road and is placed horizontally at the ground level. The second plane imitates buildings and vegetation in the background and is placed vertically behind the scene as a wall in the back. The road plane is assigned a texture from a photo bank of road texture images, while the back wall is assigned a texture from a set of photographs of urban environment. To further augment the images, we randomly shift the textures across the road and the back wall planes. Next, we simulate weather conditions. We use presets for three types of environment conditions – sunny, cloudy, and rainy weather. These conditions differ in the sources of lighting in the 3D scene and in the presence of fog (Figure 4.6, 2nd row).

We move the camera to view the scene from different viewpoints. We vary the yaw from 0° to 360°, but the pitch varies only from 5° to 35°, following the range of pitch angles in the real dataset Asterix presented in Chapter 5.

To sum up, we vary a) the yaw of the camera from 0° to 360°, b) the pitch of the camera from 5° to 35°, c) the number and placement of occluded and occluding cars, d)

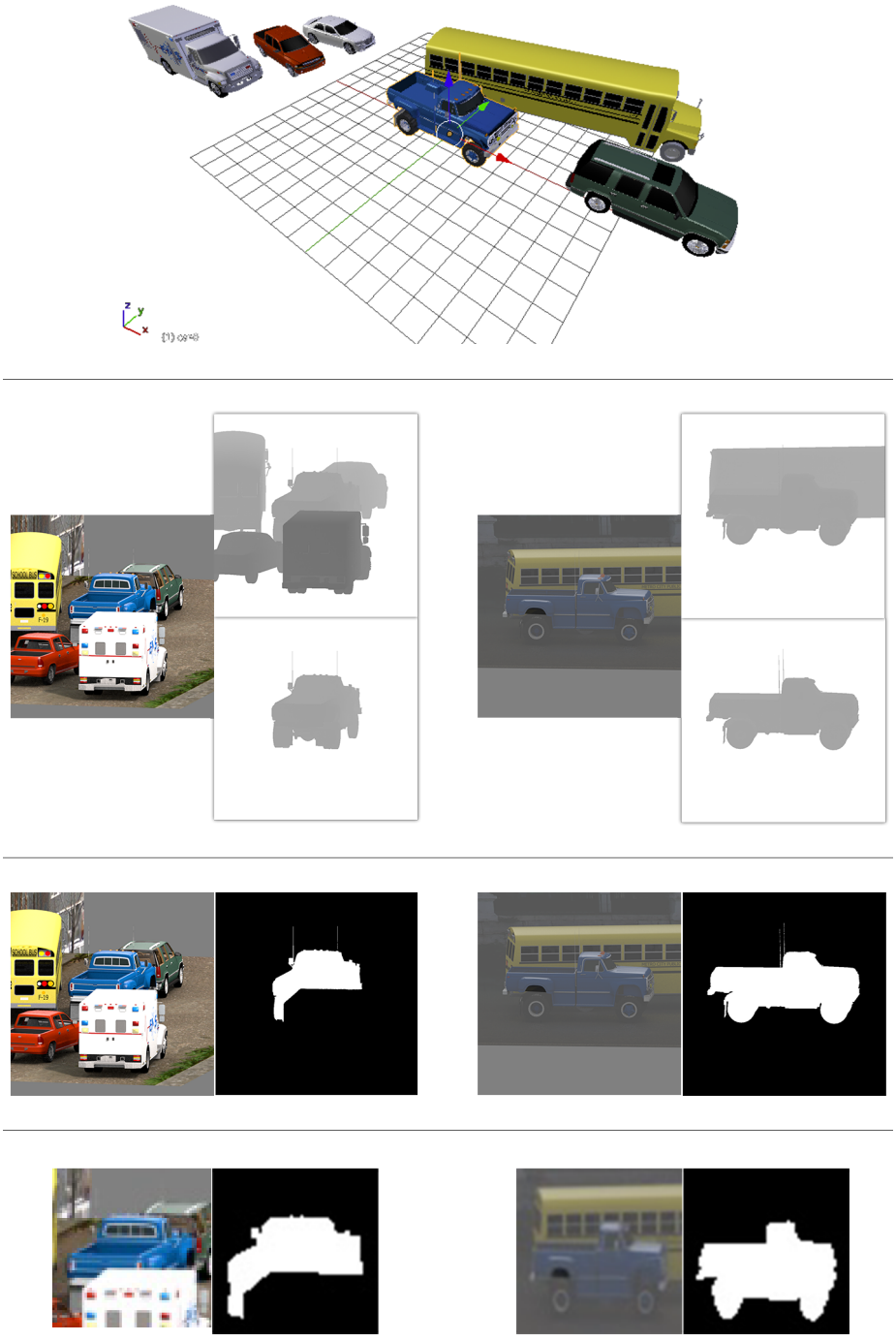


Figure 4.6: Rendering 2D synthetic dataset. 1st row: randomly pick the main car and it place in the center, picked and randomly place around the occluding cars, 2nd row: render the same scene under different viewpoints with different backgrounds, 3rd row: compute the visibility mask from the rendered depth maps, 4th row: crop the central car.

the background texture and position, and e) the weather type and sun angle.

Finally, we render the scene as an RGB image and as the depth maps of individual vehicles. We compare these depth maps pixel-wise to find the non-occluded area of the main vehicle. If a pixel in the depth map of the central vehicle is further away than that pixel in the depth map of another vehicle, then that main vehicle is occluded in that pixel. That argument allows us to generate the visibility mask of the main vehicle (Figure 4.6, 3rd row.)

At this point, the rendered image includes multiple vehicles, while we are primarily interested in the one in the center. We expand the bounding box by 20% from each side, so that the central car takes 60% width and 60% height of the image, and crop it. Then we scale the cropped patch to 64×64 pixels, distorting the car in the process (Figure 4.6, 4th row.)

Finally, we record the CAD model id, yaw, pitch, and color of a rendered vehicle into a database. The CAD model id is later used to infer more information about the vehicle from the 3D CAD dataset. In particular, we labelled cars by type ("passenger", "van", "truck", "bus") and domain ("fiction", "military", "emergency", "taxi"). New types can be added, as shown further in Listing 4.3.

We use one simple optimization trick to speed up the rendering process. A substantial time is spent on loading 3D models into blender. For speed, we re-use loaded 3D models and render each scene 10 times as seen from different viewpoints. Different scenes can be rendered in parallel. Overall, rendering 1000 scenes each under 10 viewpoints completes in 6 hours on 20 cores of Intel Xeon CPU running at 2.30 GHz. GPU acceleration did not significantly improve this time.

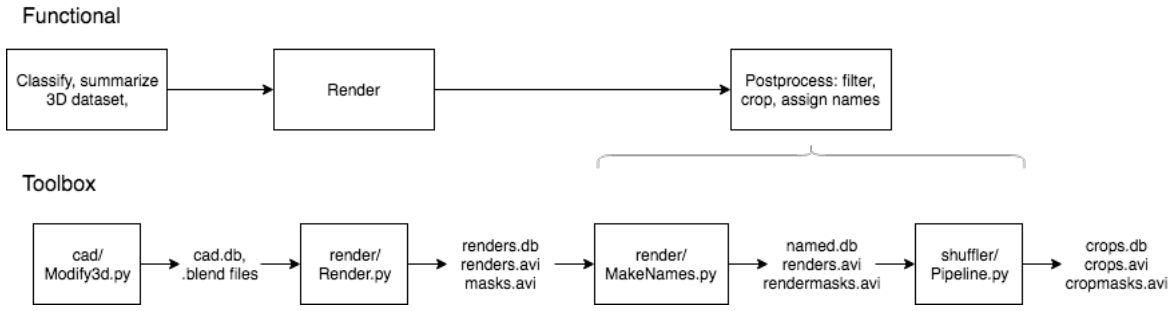


Figure 4.7: The pipeline of generating synthetic data from the functional viewpoint and from the perspective of using the toolbox.

4.4 Software toolbox

We also open source a toolbox to manipulate the 3D dataset. That includes tools to label CAD models with a user-defined labels as well as tools to generate 2D renderings. The toolbox is written in Python, it uses Blender for rendering, relies heavily on OpenCV [69] image processing library for interaction with the user, and on Matplotlib [68] and Pandas [76] toolboxes to visualize information about the dataset.

From the point of view of a user, the pipeline consists of the following steps as shown in Figure 4.7:

- 3D dataset manipulation (with `Modify3D.py`),
- Rendering (with `RenderCars.py`), and
- Post-processing (with `NameCars.py` and `Shuffler.py`).

3D dataset manipulation includes manually classifying models into user-defined types and summarizing the information about a dataset. The tool "`Modify3D.py`" was created for this purpose. Rendering includes creating random 3D scenes of the main and the occluding cars and rendering them into 2D images.

The tool `Modify3D.py` provides a set of tools to edit the 3D CAD database, as well as to summarize the data via histogram plots, and to export visualizations of CAD models.

Listing 4.3: Label 3D models by domain with `Modify3D.py`

```
src/cad/Modify3D.py \
  --in_db_file data/cad/collections.db \
  --out_db_file data/cad/collections.db \
  classify \
  --class_name='domain' \
  --key_dict_json='{"m": "military", "f": "fiction", "e": "emergency"}'
```

The tools are implemented as sub-commands, for example, command "`Modify3D.py classify`" (listing 4.3) starts a classification tool. To view the full list of available tools one can type: "`Modify3D.py --help`."

A usage example of the classification tool is shown in listing 4.3. A user can assign one of the three labels to a model: "military," "fiction," and "emergency," all under the common class "domain". A detailed documentation on a sub-command can be accessed by passing "`--help`" as the last argument after the sub-command (listing 4.4.)

In many cases a user is interested in a subset of the 3D collection, for example, only short cars. This issue is addressed by the `clause` argument to `Modify3D.py`, which essentially has the SQL syntax and is used in an SQL query internally. Its usage is illustrated in listing 4.5 for the tool "`makeGrid`." This tool queries the database for cars 9-to-11-meters-long models and arranges their renders into a grid, as in figure 4.4.

Finally, listing 4.6 presents the third capability of "`Modify3D.py`" – summarizing data. In particular, "`plotHistogram`" tool is employed to plot a histogram of CAD labels under the class "domain." The labels can be generated using listing 4.5, which was discussed above. Note that unlabelled models are plotted under a separate label "unlabelled." Also note the name of the table: "`properties`" – this is the table where any classification results are stored.

The tool `RenderCars.py` performs the second part of the pipeline, namely, creates

Listing 4.4: Label 3D models by domain with `Modify3D.py`

```
src/cad/Modify3D.py \
  --in_db_file data/cad/collections.db \
  --out_db_file data/cad/collections.db \
  classify --help
```

Listing 4.5: Display renders of cars between 9 and 11 meters with `Modify3D.py`

```
src/cad/Modify3D.py \
  --in_db_file data/cad/collections.db \
  --clause 'WHERE dims_L >= 9 AND dims_L <= 11' \
  makeGrid \
  --out_path data/cad/visualization_length_9_to_11.png
```

Listing 4.6: Plot a histogram of models by label within class "domain" `Modify3D.py`

```
src/cad/Modify3D.py \
  --in_db_file data/cad/collections.db \
  plotHistogram \
  --query 'SELECT label FROM properties WHERE class="domain"' \
  --xlabel 'length, m' \
  --show_unlabelled \
  --out_path data/cad/visualization_hist_length.eps
```

and renders 3D scenes. Each 3D scenes consists of one main vehicle surrounded by occluding vehicles and two planes – one with road texture and another with background texture.

Listing 4.7 illustrates a usage example of `RenderCars.py`. Here, 1000 scenes are created, each scene consists of a single main vehicle surrounded by 5 occluding vehicles. Both the main and the occluded vehicles are chosen from the 3D collection database `data/CAD/collections.db`. The main vehicle is picked with the SQL query defined by argument `clause_main`:

```
SELECT * FROM cad WHERE error IS NULL AND dims_L < 20
```

This query selects all the models that do not have appearance problems described in Section 4.2 and that are shorter than 20 meters. Occluding vehicles are picked with a query defined by argument `clause_occl`, which only requires them to not have appearance problems. Each of the 1000 scenes is then rendered 10 times from different angles as defined by argument `num_per_scene`. Higher value of `num_per_scene` would allow to speed up rendering but would result in more of similar looking images. Finally, argument `mode` indicates that multiple scenes should be rendered in parallel on a multicore processor.

For convenience, we store the $10,000$ renders as frames of a single video. Thus the output of the program consists of three files: a video with $10,000$ frames where each frame is a rendered image, a video with $10,000$ black-and-white visibility masks of the main vehicle in the scene, and a SQLite database with information about each frame, such as 3D model id, the bounding box of the main car, its yaw and pitch with respect to the road, the percent of the occlusion, and the color. This format was developed for easy manipulation of a 2D dataset for this dissertation. It is described in detail in Chapter 3.

At this point, 2D renders are named after the main 3D CAD model, which is in the center of the rendered image. In turn, a user may classify the 3D dataset with the

Listing 4.7: Create and render 3D scenes of a main car surrounded by occluding cars

```
src/render/RenderCars.py \
  --num_scenes 1000 \
  --num_per_scene 10 \
  --num_occluding 5 \
  --mode PARALLEL \
  --clause_main 'WHERE error IS NULL AND dims_L < 20' \
  --clause_occl 'WHERE error IS NULL' \
  --cad_db_path data/CAD/collections.db \
  -o data/render/rendered.db
```

Listing 4.8: Label an image with "type" and "domain" labels of 3D models

```
src/render/NameCars.py \
  --in_db_path  data/render/rendered.db \
  --out_db_path data/render/rendered-named.db \
  --cad_db_path data/CAD/collections.db \
  --classes 'type' 'domain'
```

"`Modify3D.py classify`" tool, as described in this chapter above. For example, in out published 3D dataset some of the models are labelled with car type: "passenger", "van", "bus", "truck" and with domain: "fiction", "military", "emergency", "taxi". The next step of the pipeline is to copy those 3D labels to the rendered cars database file with the tool `NameCars.py` as shown in Listing 4.8. In particular, an image rendered from a 3D model with two labels: "type:truck" and "domain:emergency" will be assigned a name "type:truck, domain:emergency", while an image rendered from a 3D model with only one label: "type:van" will be assigned the same name "type:van." This admittedly simple and straightforward operation was not combined with the rendering step to avoid re-rendering of the whole dataset when some 3D models are re-labeled.

Listing 4.9: Post-processing of the rendered dataset: remove cars outside image borders, expand bounding boxes around the central car by 20% on each side, crop out the bounding box, and distort it to size 64×64.

```
src/shuffler/Shuffler.py \
  -i data/render/rendered-named.db \
  filterByBorder \
  expandBoxes --expand_perc 0.2 \
  exportCarsToDataset \
    --edges=distort --target_width 64 --target_height 64 \
    --patch_db_file data/render/patches.db
```

We render the central car with occluding cars from different angles, but the same distance from the center of the scene to the camera. As a consequence, the central car of interest may be quite small in the rendered image. We are not interested in keeping much synthetic background in the rendered dataset, and, therefore, crop out only the bounding box with the central car as shown in the last row of Figure 4.6. Listing 4.9 shows how the Shuffler tool from the 2D dataset toolbox Shuffler (Chapter 3) is used to accomplish cropping in the a few steps.

1. remove all renders where the central car does not fit into the image,
2. expand the bounding box around the car by 20% from each side,
3. crop out the expanded bounding box, scale it to 64×64 pixels, with distorting the image, and
4. write the crops as a new dataset, which includes the database `patches.db`, the video with cropped renders `patches.avi`, and the video with the corresponding cropped masks `patchesmasks.avi`.

Note that the new dataset of cropped renders has the distorted proportion of the height

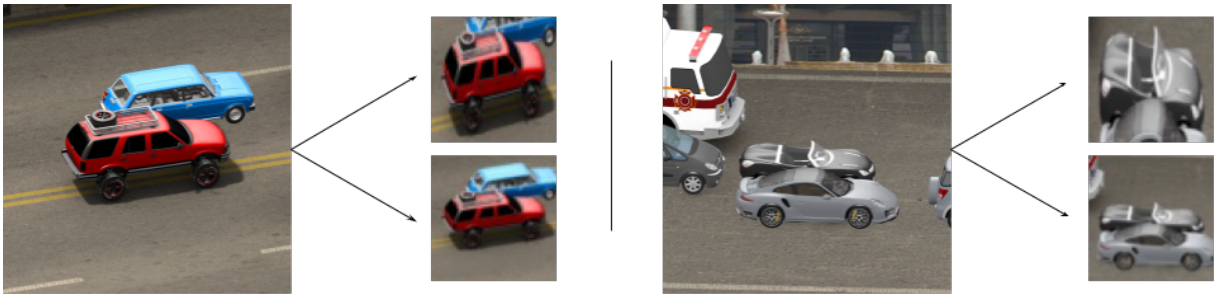


Figure 4.8: A central car can be cropped either with distortions (top row) or without distortions but including more background (bottom row).

Listing 4.10: An alternative post-processing of the rendered dataset: bounding boxes are expanded to match the target 1-to-1 height-to-width ratio, then cropped, and finally scaled to 64×64. Note the argument `--edges=background`.

```
src/shuffler/Shuffler.py \
  -i data/render/rendered-named.db \
  filterByBorder \
  expandBoxes --expand_perc 0.2 \
  exportCarsToDataset \
    --edges=background --target_width 64 --target_height 64 \
    --patch_db_file data/render/patches.db
```

and the width. It may be desirable in our case, however, and a researcher may find it more suitable to expand the bounding box to match the height-to-width image ratio, crop the image, and finally scale the crop to the size 64×64 (Figure 4.8). A small change in the previous listing allows to perform cropping while keeping the ratio, as presented in listing 4.10.

Chapter 5

Real Data: Building Asterix

The position of a car on a road can be described by its pixel-wise mask and its orientation. A car can also be characterized by a number of parameters intrinsic to the vehicle itself and not depending on the car location, such as the car type and color. For the case of high frame-rate traffic videos, one would also consider the car speed, however, that information is not available for low frame-rate publicly available camera interfaces.

We build a dataset for car segmentation and classification from the low-resolution videos of the New York City cameras [50]. We call the dataset Asterix and describe its statistical properties in Chapter 6, where it is contrasted to its synthetic counterpart Obelix.

At the same time, in this Chapter, we describe **how we collect and annotate** Asterix. In particular, we describe how we labeled the foreground mask, color, type, pitch, and yaw of each car, either automatically or with the help of an annotation team. We also illustrate the benefits of Shuffler (Chapter 3) for managing annotations efficiently.

A well-annotated dataset is a crucial step for learning a robust machine learning model, as a few mistakes in the ground truth may undermine the model performance. Therefore, it is important to double check annotations produced by humans. For example, the authors of COCO note that hired Mechanical Turk workers sometimes accidentally make mistakes, fail to follow the instructions [64], or simply provide consistently bad annotations [13].

Therefore, it is necessary to either duplicate the annotations by assigning the same task to several workers or to split annotation and verification across workers [13, 64]. We found that asking one worker to annotate and another one to verify the label adds a bias to the annotation. In a borderline case the second worker may label the image differently than the first. At the same time, if simply asked to verify the label, he/she may agree with the label produced by the first worker. Examples of borderline cases are explored in the context of labelling color and type of cars (Figure 5.9 and 5.11.) Therefore, in all annotation tasks, we ask several workers to provide labels, which are then compared, and, when not matching, dismissed.

It is worth noting that one common practice to get annotations at scale is to have Mechanical Turk workers to label images through a web interface such as LabelMe [56] (as in ImageNet [64], COCO [13], PASCAL VOC [77]), or to hire a third party annotation company (as in CityCam [50]). Though it is evident that Mechanical Turk is an easy way to organize the annotation work at scale, we noticed that personal communication with annotators creates trust and improves the quality of the result.

5.1 Image data

In this section, we talk about image data that was used to build Asterix, while the next section, we focus on annotations.

5.1.1 CityCam

We build on top of the dataset CityCam [50] which contains images from surveillance cameras in New York City. The total of $73,540$ images with $776,513$ bounding boxes of cars have been labelled.

CityCam was collected from 13 cameras located in the Manhattan borough of NYC

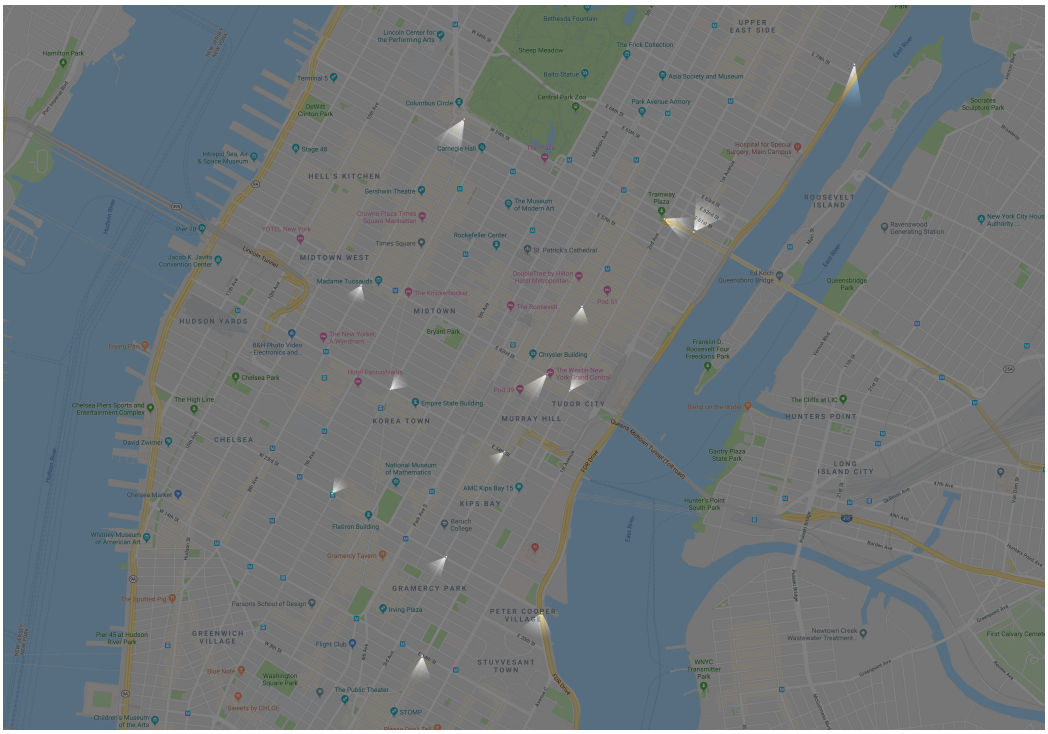


Figure 5.1: CityCam dataset [50] contains images from cameras in NYC. The field of view of the 15 cameras in Manhattan is displayed as cones. Narrow cones signify narrow field of view, short cones signify cameras looking down into the ground.

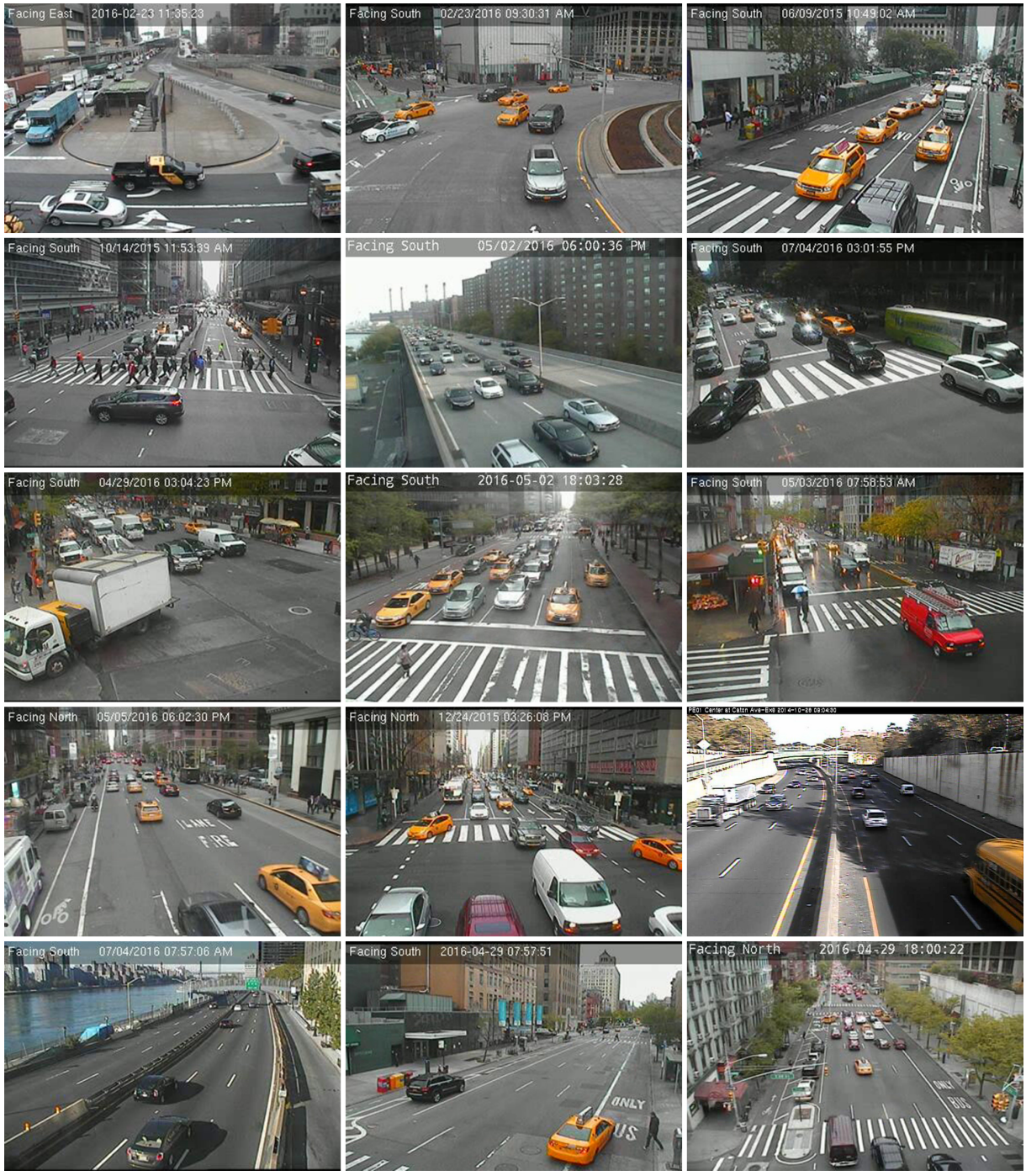


Figure 5.2: Examples of one frame from each of 15 cameras in the CityCam [50] dataset.

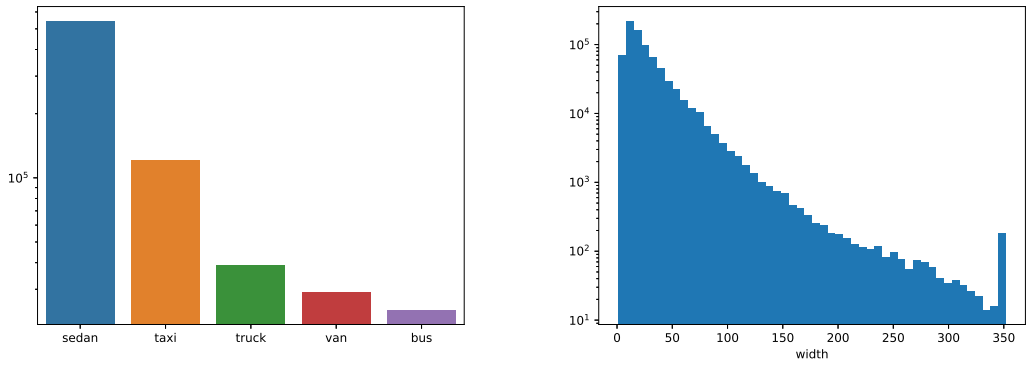


Figure 5.3: The distribution of objects (a) by type and (b) by width of ROI in CityCam [50] dataset, which we used to build Asterix.

(Figure 2.1). Later, we collaborated with the authors of CityCam to add another 2 cameras in Manhattan and Brooklyn and made it a total of 15. Examples of frames are presented in Figure 5.2. Hence, images differ significantly in appearance, and cars exhibit a variety of yaw and pitch angles. Figure 5.1 illustrates the orientation and the field of view of 14 cameras in Manhattan in more detail. Each camera is represented by a cone. The orientation of a cone coincides with the camera orientation. Furthermore, short cones signify cameras looking down into the ground, and narrow cones signify narrow field of view, which in turn corresponds to higher zoom. Cameras with narrow field of view are usually installed on high buildings or bridges to overlook a part of a highway.

Each image in CityCam contains annotations of vehicles in the form of a bounding box. Additionally, each vehicle is classified as belonging to one of the following classes: passenger, truck, bus, taxi. The distribution of vehicles by class are available in Figure 5.3,a. The authors were not interested in vehicles far away from the cameras, hence, annotators were given instructions to label cars only inside a predefined area of interest for each particular camera. Even so, the dataset contains many very small vehicles. The distribution of cars by the width of their bounding box is depicted in Figure 5.3,b.

5.1.2 Asterix

In the Asterix dataset, each image corresponds to one car. Therefore, we build Asterix by cropping out image patches in bounding boxes.

At the first step, the bounding boxes are expanded in all four direction to include some information about the background. However, a bounding box that is too large shifts the focus of a CNN model from a car to the background. Preliminary experiments indicated that expanding the bounding box by 10% in every direction results in the optimal performance.

The second step is to filter out cars which are too small. Figure 5.3,b suggests that the majority of the bounding boxes are very small, with little visual information is captured inside the bounding box. We filtered all the boxes which are less than 64 pixels tall and wide.

Additionally, we further constrain the problem by filtering out the cars at image borders and the cars that are severely occluded. That left us with $8,336$ cars.

Finally, the image patches in the bounding boxes are cropped out, and resized to the shape of 64×64 pixels.

Finally, additional annotations were added. First, camera orientation and position was reconstructed for all 15 cameras. That allowed us to label the pitch automatically for every car based on its position in the image. After that, we worked with a team of annotators to label the background mask, the yaw (car orientation on the ground plane), the color, and the type of each car. Note that all annotations except for mask were made on the CityCam dataset and then copied to Asterix. The rest of the chapter describes the annotation process in details.

Filtering and extracting bounding boxes for Asterix from the CityCam dataset was performed with the help of the Shuffler toolbox (Chapter 3) as a single command in the command-line:

```
shuffler.py -i "CityCam.db" -o "Asterix.db" \
```

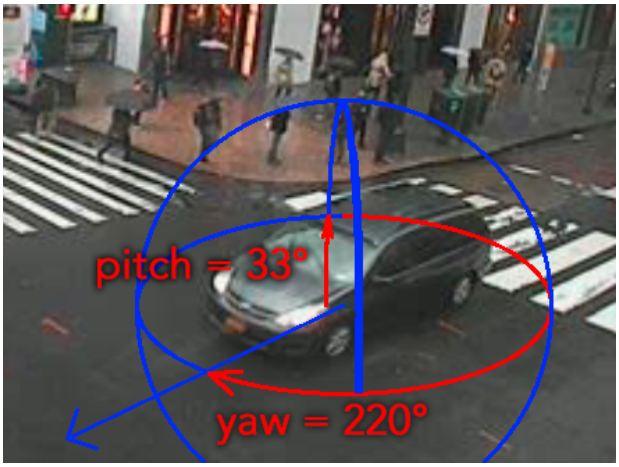


Figure 5.4: **Pitch** is the angle of the ground plane as seen by the camera. Pitch is 0° for cars at the horizon and 90° for cars that are seen top-down. **Yaw** signifies the orientation of a car on the ground plane. 0° is the orientation away from the camera, 90° is heading to the right.

```
expandBoxes --expand_perc 0.2 \| \
filterObjectsSQL "SELECT␣objectid␣FROM␣objects␣WHERE␣width␣<␣64␣AND␣height
    ␣<␣64␣AND␣name␣IN␣('van',␣'taxi',␣'sedan',␣'truck',␣'bus')" \| \
filterObjectsAtBorder \| \
deleteEmptyImages \| \
cropObjects --edges distort --target_width 64 --target_height 64 --
    image_path "patches"
```

5.2 Collecting annotations

This section details the process of collecting each of the annotations type for Asterix.

5.2.1 Pitch

Pitch is the angle of the ground plane as seen by the camera (Figure 5.4). The pitch of a car in the top-down view is 90°, while the pitch of a car at the horizon is always 0°.

We augmented the dataset with pitch automatically, without the help of human annotators. In order to do that, we reconstructed the 3D geometry of the scene for each of the 15 cameras. Below the process is described in detail:

For every camera, we first estimate its extrinsic and intrinsic parameters using point correspondences between a frame from the camera and a top-down view of the scene. We use a snapshot from Google Satellite for the top-down view image. Assuming flat ground, a point in the image frame (x, y) and a point in the top down view (x', y') are connected through a 3×3 homography matrix H [78].

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5.1}$$

The homography matrix H is defined up to scale, and usually $H_{3,3}$ is set to 1. Therefore, the matrix-form Equation 5.1 can be rewritten as one equation for x and another one for y. Thus, one point correspondence generates two equations and four correspondences generate eight equations. Then H can be computed by solving the linear system of the eight equations relative to the entries of H [78]. In practice, we gather more than four correspondences and keep the best fitting ones with the help of the RANSAC algorithm [79]. We labelled point correspondences for the 15 cameras using a custom written GUI tool with the OpenCV [69] package (Figure 5.5). OpenCV was also used for computing H.

Next, we estimate the pitch for any point on the ground plane. In order to do that, we estimate the flattening of a circle in the top-down view when projected as an ellipse onto the image plane. In particular, we consider a point $\mathbf{p'_0}$ in the top-down view and its deviations along the x and y axes: $\mathbf{p'_1} = \mathbf{p'_0} + \mathbf{\Delta'_x}$ and $\mathbf{p'_2} = \mathbf{p'_0} + \mathbf{\Delta'_y}$. By applying Equation 5.1 to points $\mathbf{p'_0}$, $\mathbf{p'_1}$, and $\mathbf{p'_2}$ we compute their projections in the image plane $\mathbf{p_0}$, $\mathbf{p_1}$ and $\mathbf{p_2}$. Just like the vectors $\mathbf{p'_1}$, and $\mathbf{p'_2}$ lie on a circle with the center in $\mathbf{p'_0}$ in the image plane, their projections $\mathbf{p_1}$ and $\mathbf{p_2}$ lie on an ellipse with the center in $\mathbf{p_0}$. Let

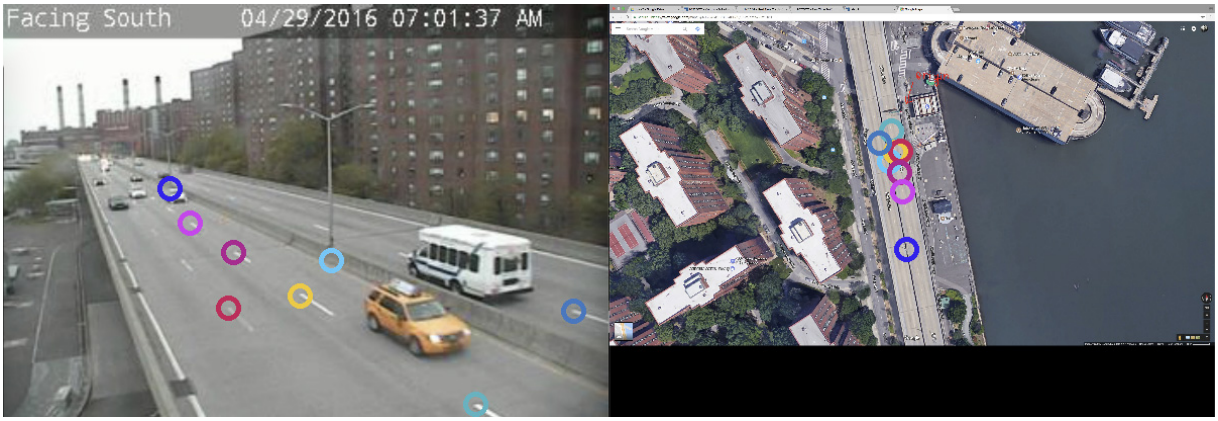


Figure 5.5: For each of the 15 cameras in CityCam, we match points between a frame from a video (left) and a Google Satellite snapshot (right) in order to compute a homography between them.

$\mathbf{p_1} - \mathbf{p_0} = (x_1, y_1)$ and $\mathbf{p_2} - \mathbf{p_0} = (x_2, y_2)$. We calculate the flattening of that ellipse α as follows:

$$\begin{cases} \frac{y_1^2}{\alpha^2} + x_1^2 = r^2 \\ \frac{y_2^2}{\alpha^2} + x_2^2 = r^2 \end{cases}$$

where r is the ellipse unknown radius. That results in estimating the flattening as:

$$\alpha = \sqrt{\frac{y_1^2 - y_2^2}{x_2^2 - x_1^2}} \tag{5.2}$$

Finally, once the pitch of points on the ground plane are known, we estimate the pitch of individual cars. We are interested in the pitch of the car center, or more precisely, its projection onto the ground plane. That point on the ground plane does not coincide with the center of the bounding box. Instead, empirically, we found that it can be estimated as:

$$\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0.5\, x_{\min} + 0.5\, x_{\max} \\ 0.25\, y_{\min} + 0.75\, y_{\max} \end{bmatrix} \tag{5.3}$$

That is, it lies in the middle between $x_{\min}$ and $x_{\max}$, and closer to $y_{\max}$ than to $y_{\min}$.

The result of automatically computing the pitch for the 15 cameras is demonstrated in Figure 5.6. Note that, since the road is seen differently by the cameras, the distribution of the pitch also varies across cameras.

5.2.2 Yaw

The yaw angle represents the orientation of a car on the ground plane (Figure 5.4.) 0° corresponds to a car heading away from the camera, 90° – to a car heading to the right.

Knowing the combination of the yaw and pitch angle is equivalent to knowing the 3D pose of a car as in KITTI [57], or StanfordCars 3D and CompCars 3D [80]. These datasets represent car-mounted cameras where camera pose relative to the ground is not necessarily known and the ground can not be always assumed to be flat. Therefore, in these cases it is not possible to label pitch of other cars automatically like we did in the previous subsection, and instead authors hire human annotators to label cars with 3D boxes, which would automatically provide both the pitch and the yaw.

In our case, however, the pitch is known and there is no need to employ labelling with 3D boxes. Instead, for every car, we draw an ellipse on the ground next to it (Figure 5.7). The ellipse represents the projection of a circle in a top-down view as seen from the camera. Annotators are asked to draw an arrow which starts at the center of the ellipse and extends in the direction of the car. As can be seen from Figure 5.7, the direction of the arrow coincides with multiple contours in the car, such as the line between the wheels. That facilitates annotating for a human worker. The annotation process is recorded as a video and is available at `https://youtu.be/1gPkq_t_Utw`.

We assigned two annotators to label the yaw of every car independently. If the labels for any particular car differ by more than 10°, that car is re-labelled. Otherwise, the yaws from different annotators are averaged.

The tool is written in Python using OpenCV library [69]. It was integrated into the

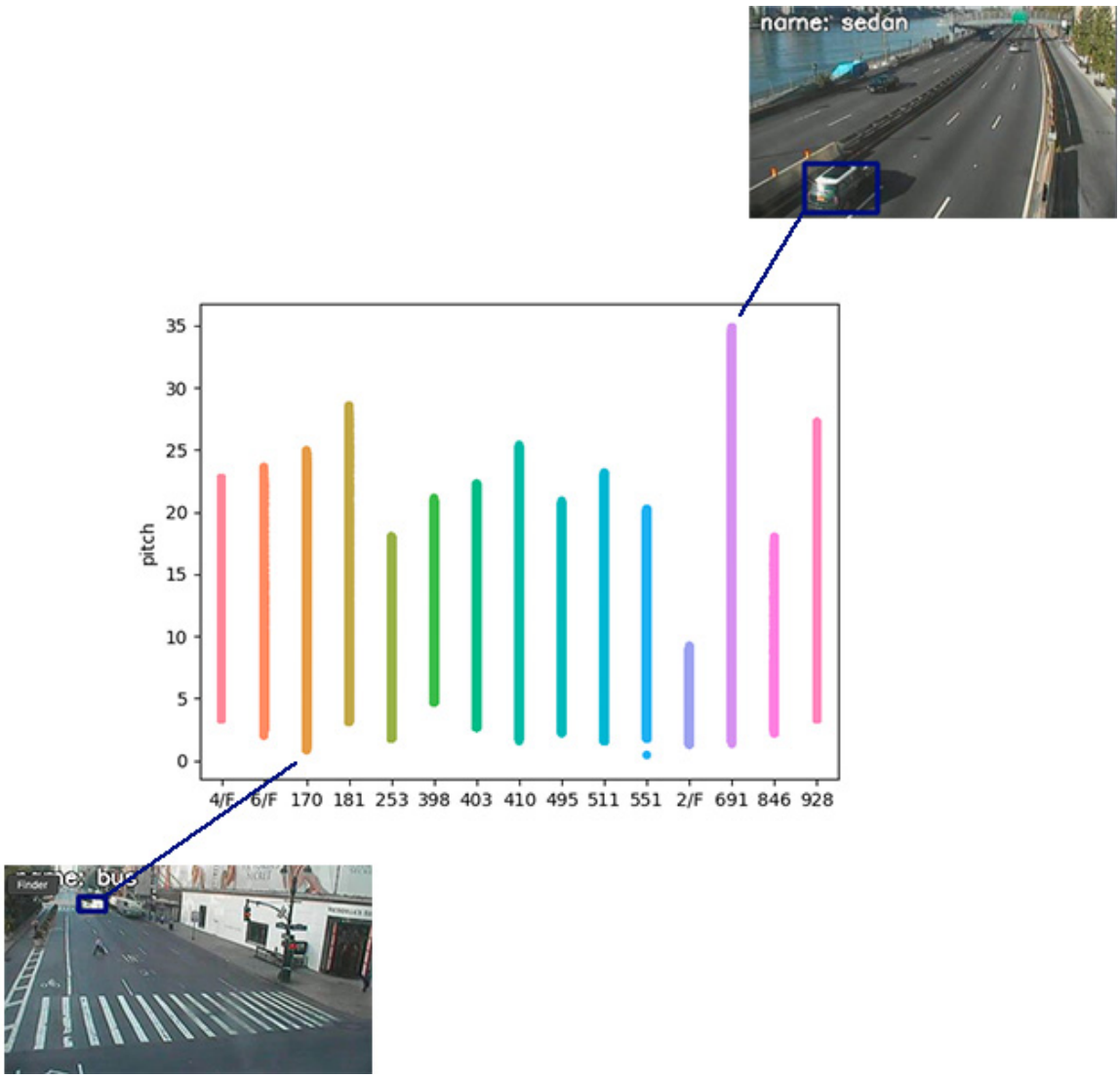


Figure 5.6: The distribution of the pitch angle of vehicles varies across cameras. It varies from the maximum of 35° for cars near the lower boundary of the image in camera 691 to the minimum of 1° for cars close to horizon in camera 170. Cars even further away from the camera are very small and were not labelled.

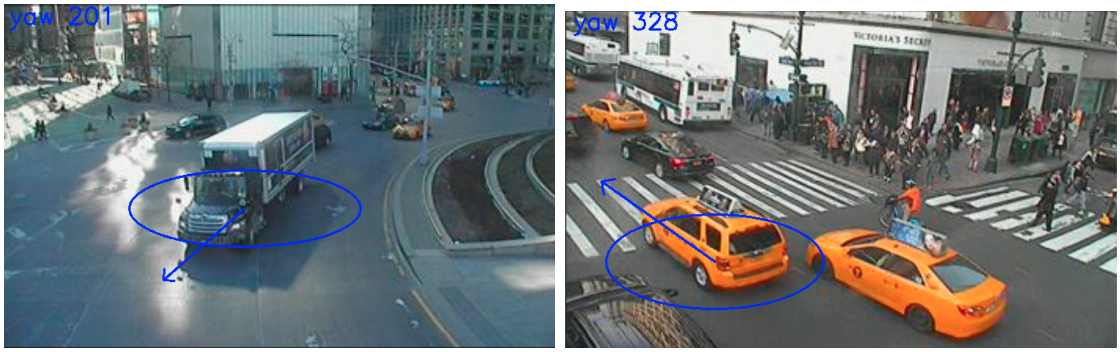


Figure 5.7: Yaw annotation tool at work. Scene homography allows a user to label the yaw with respect to the car pitch. The arrow follows the direction from the center of the ellipse to the mouse.

Shuffler toolbox (Chapter 3). Annotators were provided with the Shuffler command below. The command would iterate through those cars in CityCam that (1) were chosen for the Asterix dataset and (2) do not yet have the "yaw" class property. The latter point is important since it allows to run the command repeatedly while hiding all the cars that have been labelled before.

```
shuffler.py -i CityCam.db -o CityCam.db \
  sql "ATTACH␣'Asterix.db'␣AS␣'Asterix'" \| \
  labelYaw --where_object \
    "objectid␣IN␣(SELECT␣objectid␣FROM␣Asterix.objects␣obj)␣AND␣objectid␣NOT
       ␣IN␣(SELECT␣objectid␣FROM␣properties␣WHERE␣key␣==␣'yaw')"
```

5.2.3 Color

Color is a distinctive feature of the car appearance, despite many borderline cases and the change of car appearance in different lighting conditions.

We used human workers to annotate color. The annotators could pick from red, green, blue, orange, yellow, black, white, and gray colors. Additionally, they were given the instruction to assign the "undefined" label if a color could not be assigned. Below we cite

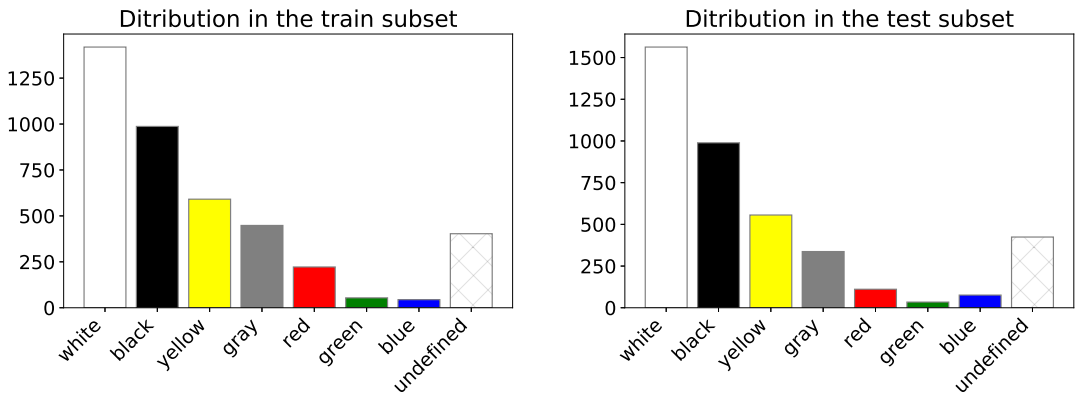


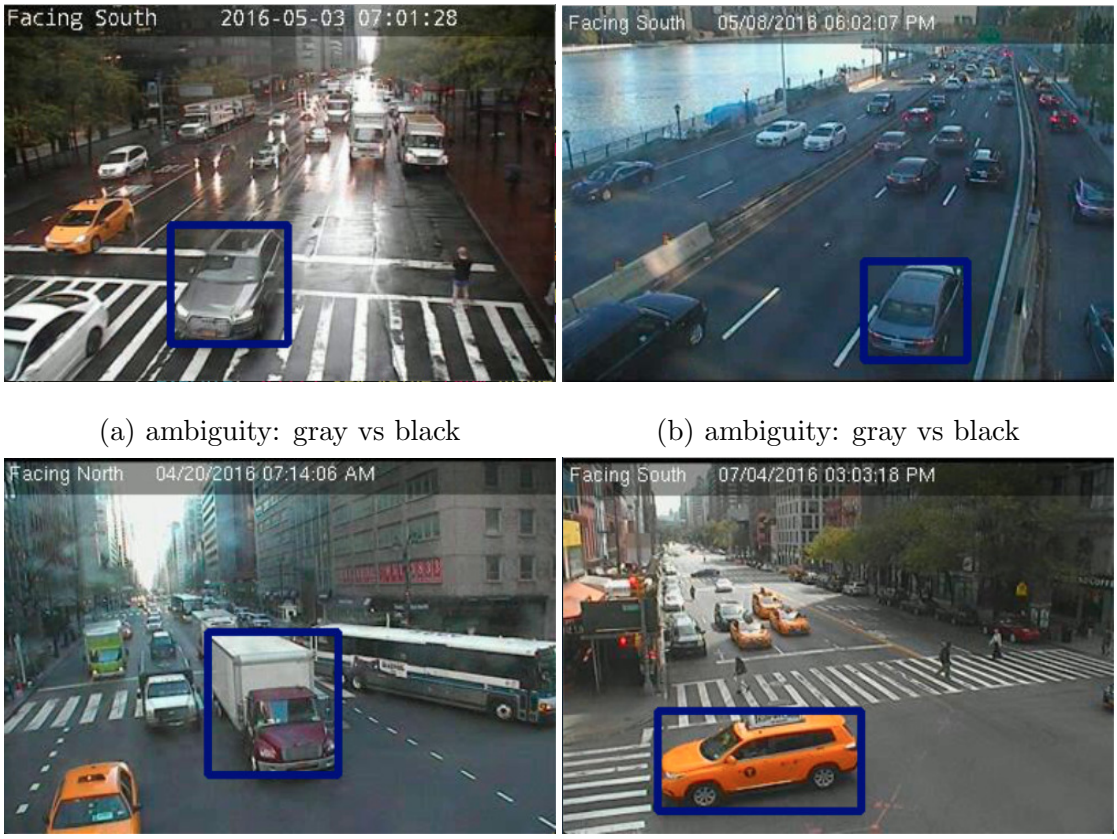
Figure 5.8: Distribution of colors in NYC dataset.

our instructions on when to leave the color undefined:

- if a car does not have a dominant color (e.g., it is striped). If it has a dominant color though, assign that color (e.g., a yellow taxi with a black taxi sign should be labeled yellow)
- if the color is not from the list (e.g., purple)
- if the color is in between two other colors, and you can't say which one is closer (e.g., in between blue and green)
- if you can not see what the color is (e.g., in the dusk)

We hired two workers to annotate the color of each car independently. That is in contrast to the protocol of annotating an object in [13], where in order to reduce cost, the first worker was given the annotation task, and the second one – the verification task. We argue that seeing the label of the first worker would introduce a bias to the judgement of the second worker. The examples of ambiguities in color are presented in Figure 5.9.

If the annotators agreed about the color of a car, their labels were merged. No label was assigned to an object in case one annotator decided that a color can not be assigned or if the labels given by the annotators did not match.



(a) ambiguity: gray vs black

(b) ambiguity: gray vs black

(c) ambiguity: white vs red

(d) ambiguity: orange vs yellow

Figure 5.9: Annotators did not agree on the color of 246 images out of 4,090 in the test subset. The captions read the two different colors of each car. The reasons for confusion include (a) the poor visibility of a car because of reflections from the sky or (b) because of being in a dark area in the image; (c) a car that is painted in multiple colors; and (d) a color that is on the borderline between two base colors.

We present the distributions of colors for the train and the test subsets in Asterix in Figure 5.8. The relatively large number of yellow cars corresponds to the New York City taxis. The "orange" color was deleted from the database, because only 3 cars total were labelled orange.

Both the train and the test subsets have roughly 10% of cars with undefined color. As described above, some of them are labelled "undefined" by annotators, while the others are labelled so because annotators did not agree on the color. It is instructive to explore the reasons of the mismatch between annotators. We identified four distinct reasons, as illustrated in Figure 5.9: (a) the poor visibility of a car because of reflections from the sky or (b) because of being in a dark area in the image, (c) a car that is painted in multiple colors, and (d) a color that is on the borderline between two base colors.

Annotators used the Shuffler toolbox (Chapter 3), which allowed us to incorporate and compare the annotations easily. The command below calls a GUI for labelling those objects that do not have a property "color" yet, that is, that have not been yet labelled.

```
shuffler.py -i CityCam.db -o CityCam.db \
  labelObjects --property "color" \
    --key_dict "{'-':'previous',␣'=':'next',␣27:'exit',␣127:'delete_label',␣
       'g':'green',␣'b':'blue',␣'o':'orange',␣'y':'yellow',␣'k':'black',␣'r
       ':'red',␣'w':'white',␣'a':'gray',␣'␣':'undefined'}" \
    --where_object "objectid␣NOT␣IN␣(SELECT␣objectid␣FROM␣properties␣WHERE␣
       key␣==␣'color')"
```

More advanced capabilities of Shuffler are demonstrated by the next command. It iterates through objects that have been labelled by both annotators (not left "undefined") but whose labels mismatch. Each of these objects is displayed on screen together with its two mismatching labels. Note that most of the business logic is implemented via the standard SQL syntax.

```
shuffler.py -i annotator1.db \
```

```
sql "ATTACH␣annotator2.db␣AS␣attached" \| \
examineObjects --where_objects \
  "objectid␣IN␣(SELECT␣p1.objectid␣FROM␣properties␣p1␣INNER␣JOIN␣attached.
     properties␣p2␣ON␣p1.objectid␣==␣p2.objectid␣WHERE␣p1.key␣==␣'color'␣
     AND␣p2.key␣==␣'color'␣AND␣p1.value␣!=␣p2.value␣AND␣p1.value␣!=␣'
     undefined'␣AND␣p2.value␣!=␣'undefined')"
```

Finally, Chapter 3 talks about compatibility of Shuffler with SQL readers. In particular, a standard Unix command line program `sqlite3` allows us to print the pairs of colors (including "undefined") that mismatch across the two annotators.

```
sqlite3 annotator1.db "
␣␣ATTACH␣annotator2.db␣AS␣attached;
␣␣SELECT␣p1.value,␣p2.value␣FROM␣properties␣p1
␣␣INNER␣JOIN␣attached.properties␣p2␣ON␣p1.objectid␣==␣p2.objectid
␣␣WHERE␣p1.key␣==␣'color'␣AND␣p2.key␣==␣'color'␣AND␣p1.value␣!=␣p2.value"
```

5.2.4 Vehicle type

Just like the color, the vehicle type was annotated by human workers. Because of the difficult and ambiguous hierarchy of vehicle types, we converged on the following four classes: passenger, truck, bus, van. Additionally, annotators could use the "undefined" label if the type could not be assigned. Below we cite the instructions that define the types:

- "p": "passenger". Sedans, hatchbacks, sport cars, SUVs, taxis, etc.
- "t": "truck". Commercial trucks, UHauls, pickups, etc.
- "v": "van". Commercial vans, normally with no windows.
- "b": "bus". In-city or inter-city buses, school buses.

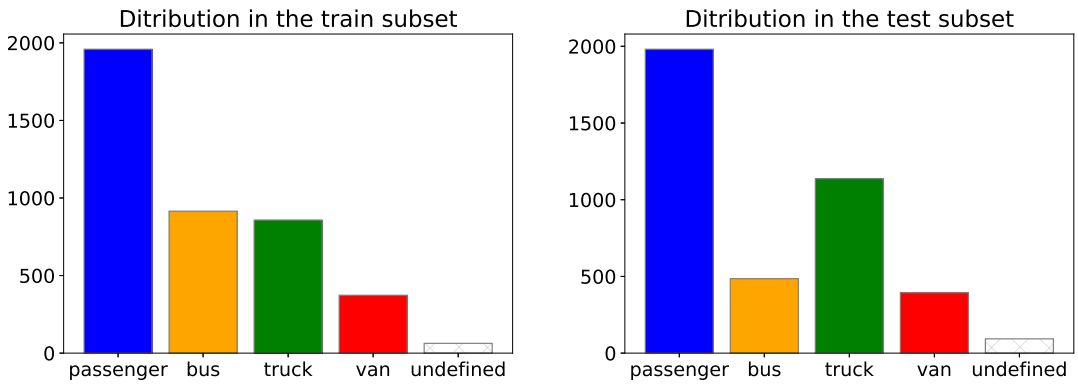


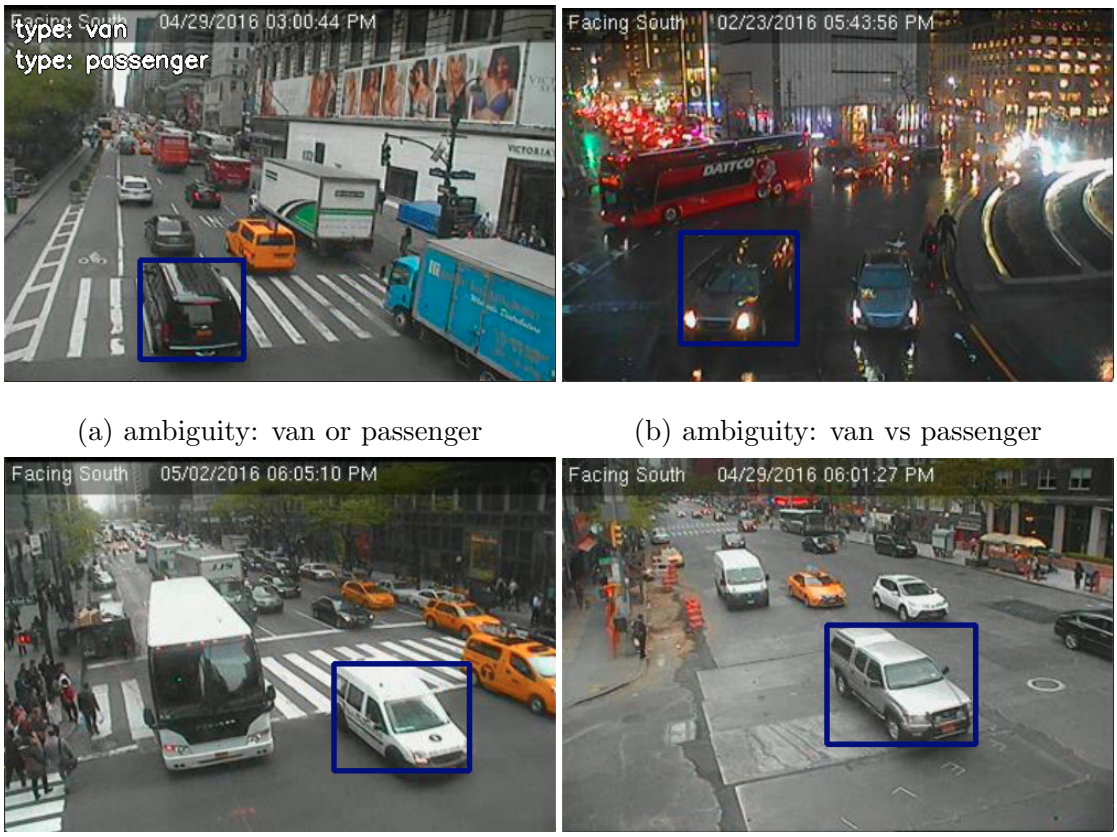
Figure 5.10: Distribution of car types in NYC dataset.

If the annotators agreed about the type of a car, their labels were merged. No label was assigned to an object in case one annotator decided that a type can not be assigned or if the labels given by the annotators did not match. We present the distributions of types for the train and the test subsets in Asterix in Figure 5.10.

As described above, some of the cars with undefined type are labelled so by annotators, while the others are labelled so because annotators did not agree on the type. It is instructive to explore the reasons of the mismatch between annotators. We identified four distinct reasons, as illustrated in Figure 5.11: (a) inconvenient angle; (b) poor visibility ; general ambiguity about the definition of (c) "van" and (d) of "truck."

Annotators used the Shuffler toolbox (Chapter 3), which allowed us to incorporate and compare the annotations easily. The command below was given to the annotators. It runs a GUI for labelling those cars that have not been previously labelled.

```
shuffler.py -i CityCam.db -o CityCam.db \
  labelObjects --property "type" \
    --key_dict "{'-':'previous',␣'=':'next',␣27:'exit',␣127:'delete_label',␣
       'p':'passenger',␣'b':'bus',␣'t':'truck',␣'v':'van',␣'␣':'undefined'}"
        \
    --where_object "objectid␣NOT␣IN␣(SELECT␣objectid␣FROM␣properties␣WHERE␣
       key␣==␣'type')"
```



(a) ambiguity: van or passenger

(b) ambiguity: van vs passenger

(c) ambiguity: van or passenger

(d) ambiguity: truck vs passenger

Figure 5.11: 63 images out of 4168 in the train subset remained unlabelled. The reasons include (a) inconvenient angle; (b) poor visibility; general ambiguity about the definition of (c) "van" and (d) of "truck."

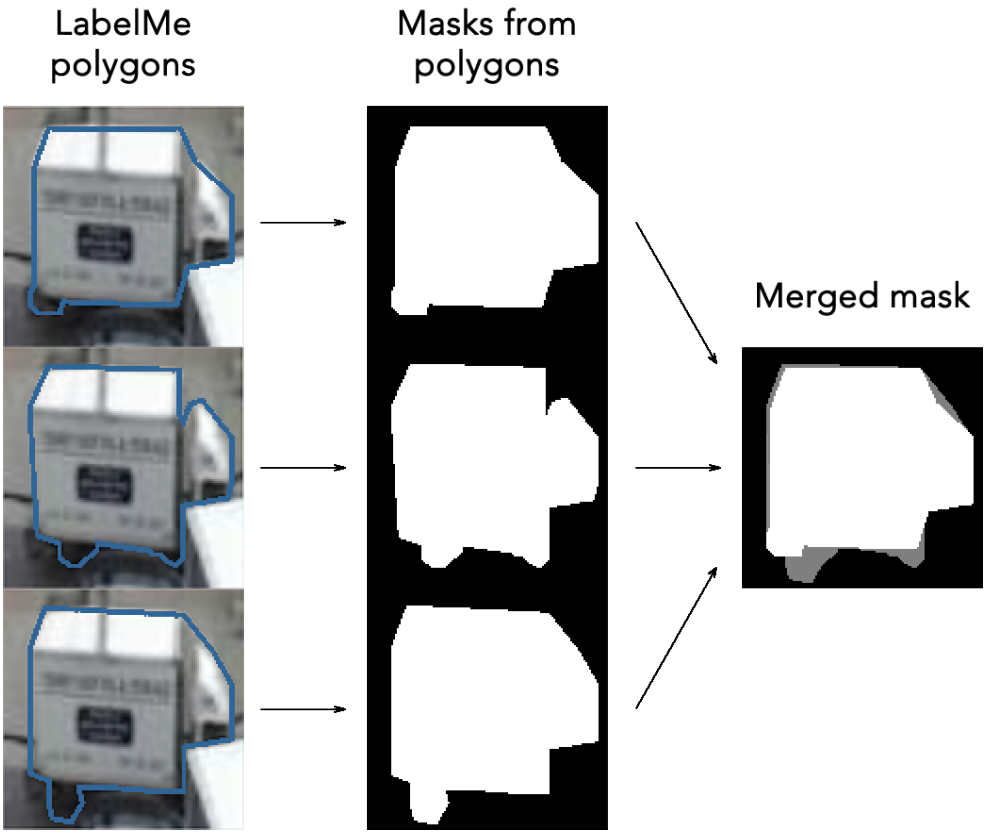


Figure 5.12: Annotating with mask is done in 3 steps. First, annotators draw a polygon around an car, then polygons are converted to masks. Finally, masks are merged and their mismatching pixels are turned labelled as "Don't care".

5.2.5 Mask

Finally, images were annotated with foreground mask. We annotated the cropped images from the Asterix dataset as opposed to the labelling source CityCam images as in pitch, yaw, color, and type. We used LabelMe annotation tool [56] for the task. Three annotators independently were asked to draw a polygon around a car (Figure 5.12.) The occluded parts were not annotated in accordance to the standard protocol [27, 56, 64]. After that, the areas inside polygons were used as the foreground mask.

Despite the instructions, the masks from different annotators always have differences. For example, the dark area under a car can sometimes be annotated as belonging to the car

Figure 5.13: Left: image; center: "Don't care" mask lies outside the car and does not include the blurred car boundary; right: "Don't care" mask after applying the erosion filter to the "Car" mask includes the car boundary.

or to the background. Instead of trying to eliminate the differences between annotations, we actually make use of them. In particular, we use them to initialize the third class of pixels: "don't care." This class is added to the segmentation datasets [27, 56] to compensate for the intrinsic ambiguity of pixels on the border of objects. Predictions for pixels of this class are not considered when evaluating segmentation algorithms. In our case, the mismatch in small details between human annotators also means that the mismatching pixels should be assigned to the "don't care" class.

Finally, the manual inspection of masks showed that annotators consistently assign the blurred boundary of the car to the "Car" class. Thus the "Don't care" mask lies outside the car (Figure 5.13 center). To compensate for this bias, we applied the erosion filter to the "Car" class. The eroded pixels were assigned to the "Don't care" class (Figure 5.13 right).

We conclude this section by noting that we used the Shuffler tool (Chapter 3) to import the LabelMe annotations, merge the annotations from different workers, to convert the polygons to a mask, and to assign "Don't care" class to mismatching pixels:

```
shuffler.py -i Asterix.db -o Asterix.db \
  importLabelmeObjects --annotations_dir "annotator1" \
```

```
  --keep_original_object_name --polygon_name "annotator1" \| \
importLabelmeObjects --annotations_dir "annotator2" \
  --keep_original_object_name --polygon_name "annotator2" \| \
importLabelmeObjects --annotations_dir "annotator3" \
  --keep_original_object_name --polygon_name "annotator3" \| \
mergeObjectDuplicates \| \
polygonsToMask --mask_path "masks"
```

5.3 Conclusion

In this chapter, we discussed the process of building Asterix – a dataset of real images which is formally presented in the next chapter. In particular, we described the image data from New York City cameras and the CityCam dataset that Asterix is based on. We discussed how Asterix is automatically annotated with the pitch angle, and how we worked with an annotation team to label the yaw, color, type, and the foreground mask of cars. Finally, we mentioned the developed toolbox Shuffler (Chapter 3), and showed how it was used to process annotations in an efficient way.

Chapter 6

Asterix and Obelix Datasets

In this Chapter, we present the twin datasets – Asterix and Obelix (Figure 6.1.) They represent images with a single car as viewed from traffic surveillance cameras. Asterix is the real image dataset with $8,336$ images, while Obelix contains $80,145$ synthetic computer-generated images.

We refer the reader to Chapters 5 and 4 for the details on building the Asterix and Obelix datasets respectively, while in this Chapter, we assume no prior knowledge on the process. Instead the goal of this chapter is to describe the properties of the datasets and compare them to other datasets.

Asterix contains annotated images collected from multiple NYC traffic surveillance cameras. It contains cars viewed under various viewpoints, in different light and weather conditions. Each image has one car, possibly occluded by other cars or structures. The images are cropped from the CityCam [50] dataset that contains frames from low-resolution low-framerate videos collected from 15 New York City traffic surveillance cameras. A reader is referred to Figure 5.2 in Chapter 5 for examples of these frames. Asterix was annotated by human workers. The number of images in Asterix is $8,336$.

Obelix is its synthetic counterpart and has an order of magnitude more data: it contains $80,145$ images. It has images of computer-generated cars as if seen by traffic cameras. Each

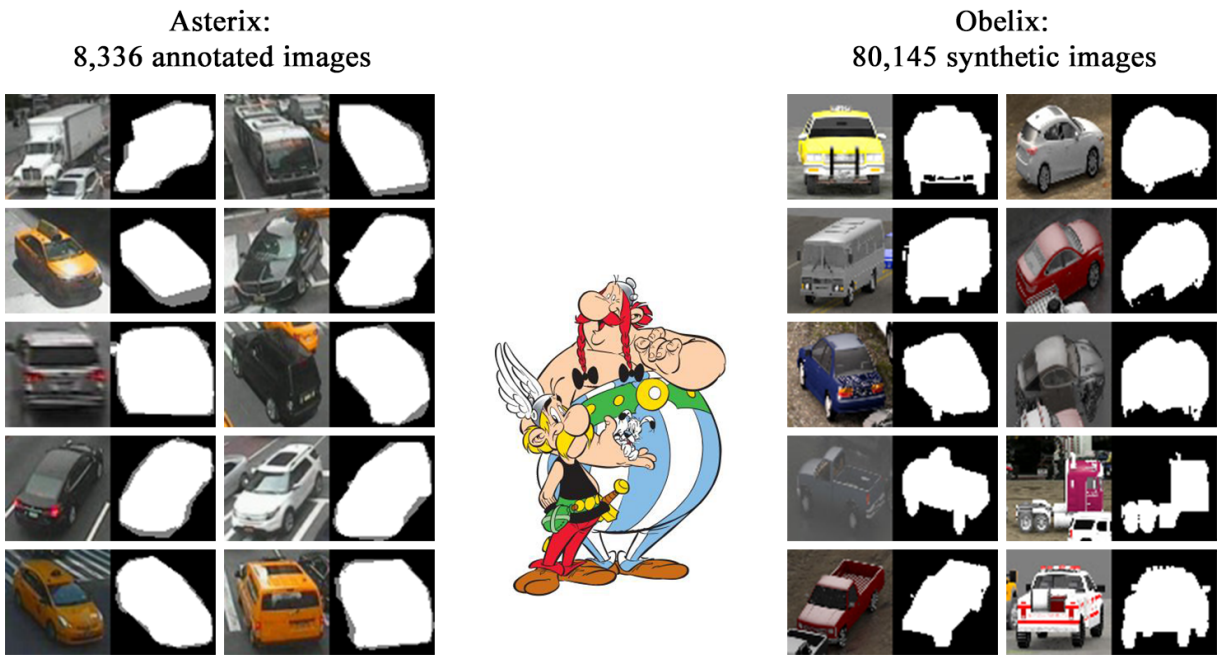


Figure 6.1: Asterix and Obelix datasets. This figure reiterates Figure 2.3 from Chapter 2.

image is generated by rendering a single 3D car model from a pool of about $1,000$ models in the CADillac dataset described in Chapter 4. In particular, a 3D car model is placed in a virtual road scene, with textured planes serving as background, and surrounded by other 3D models. Additionally, weather effects are simulated: in the sunny weather, a car drops a shadow on the ground, while in the rainy weather, the visibility is low and a car is reflected in the wet road.

Asterix and Obelix are designed for the experiments on how synthetic data can replace real data in the analysis of individual cars. Therefore, both datasets are annotated with the same labels: inherent car features such as **color**, **type**, and **domain** and external features that depend on the original video frame: **yaw**, **pitch**, and **foreground mask**.

6.1 Related work

Several datasets have recently been proposed to aid research in traffic surveillance systems, e.g. TRANCOS [81], DETRAC [14], CompCars [10], and AI CITY [15] (Table 6.1). These datasets focus on the tasks of car detection, tracking, and re-identification of vehicles. The applications of these tasks range from automatic traffic density estimation to tracking a vehicle across a city [14]. At the same time, Asterix & Obelix focus on problems that require rich information about individual cars. For example, detecting if a car has crossed a lane or did not stop in front of a stop sign line requires the knowledge of a detailed position of a car on the road. Asterix provides the annotations in the form of pixel-level foreground mask that addresses this problem.

Additionally, some datasets [10] are focused on the scenario where the researcher has access to a camera and consequently to the high-resolution video. At the same time, the web-interface that may be made publicly available for a municipal camera usually provides low framerate and low resolution video. The diversity and the zero cost of these data is an attractive opportunity for research, while the poor image quality presents interesting challenges. We focus our research on low-resolution images.

Furthermore, in this thesis, we focus on comparing the real and the synthetic data, and, therefore, present the twin Asterix and Obelix datasets. Synthetic datasets that match the real data are well known in the field of autonomous driving [52, 53, 54, 75]. They match images recorded by car-mounted cameras from datasets such as [11, 12, 25, 57, 58, 65]. Moreover, Virtual-KITTI [53] is specifically designed to match images from KITTI [57] closely. The work of Lopez et al. [35] examines the aforementioned datasets to explore the requirements for synthetic images that would make them useful for training semantic models. No such work was performed for traffic data, and Asterix & Obelix aim to close the gap in the individual car analysis.

Dataset	TRANCOS	CompCars	DETRAC	AI CITY	CityCam	**Asterix**
Number of frames	823	50K	84K		32K	8K
Cropped?	no	no	no	no	no	yes
Resolution	640×480		960×540	1920×1080	352×240	64×64
Foreground mask	no	no	no	no	no	**yes**
Vehicle type	no	yes	yes	yes	yes	yes
Orientation	no	discr. yaw	yaw		discr. yaw	yaw, pitch
Year	2015	2015	2015	2017	2018	2019

Table 6.1: Comparison of existing datasets of real data from traffic surveillance cameras. "discr. yaw" in "orientation" row corresponds to a small number of views, e.g., "front," "back."

6.2 Annotations

In this section, we discuss the annotations in detail.

Color is a discrete label that can take one of the following values: red, green, blue, orange, yellow, black, white, and gray. A color can also be undefined, because (1) a real or synthetic car is multi-colored, or (2) the color of the car is in between two of the base colors, or (3) because the color of the real car is not recognizable in the image for the case of Asterix. The latter case is explained in detail in Chapter 5. The distribution of cars across type is presented in the top row of Figure 6.2.

We recognize the following car **types**: passenger, truck, bus, and van. "Truck" can be as big as a trailer or as small as a pickup. "Bus" includes both public and private buses, as well as school buses. "Van" refers to commercial vans, either for transportation of passengers, or for moving goods, but not to minivans. Minivans and others cars owned by individuals for the purpose of personal use classify as "passenger" cars. The two exceptions from this

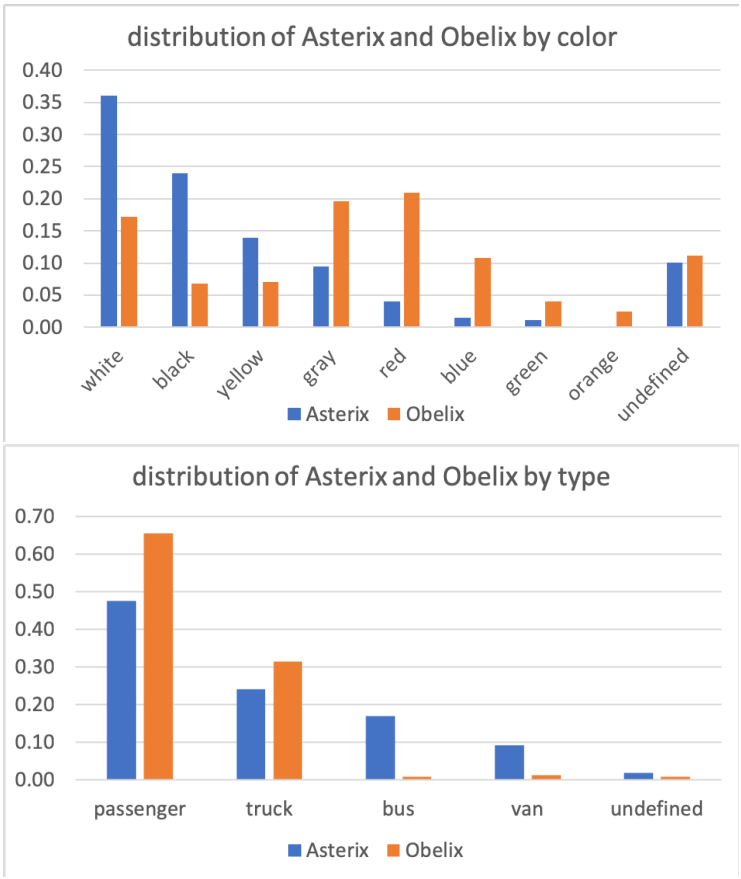


Figure 6.2: Relative distributions of cars In Asterix and Obelix by color and type

rule are pickups, which belong to the class "truck" though they may be for personal use, and taxis, which belong to class "passenger" despite the commercial use. The distribution of cars across type is presented in the bottom row of Figure 6.2. Balancing the "bus" and "van" classes is a subject of future work.

Taxis are an important subclass due to their high number in New York City. Both Asterix and Obelix have a label that we called **domain**, that corresponds to the domain of use. The only domain present in both Asterix and Obelix is "taxi." Obelix has cars from additional domains: "emergency" (which includes police, ambulance, and firetrucks) and "fiction" (which includes several cars models from pop-culture.)

Yaw and **pitch** are the two angles that represent the orientation of a car as seen by a

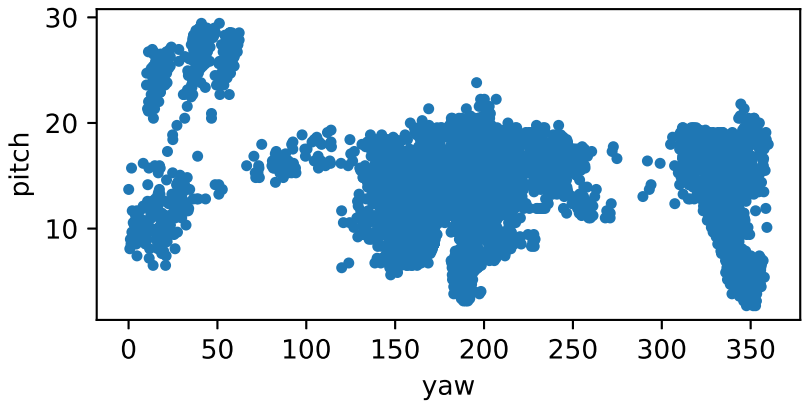


Figure 6.3: Distributions of yaw and pitch angles in the Asterix dataset.

camera. They are explained in Figure 6.4. Pitch is the angle of the ground plane as seen by the camera. The pitch of a car in the top-down view is 90°, while the pitch of a car at the horizon is always 0°. The distribution of the pitch angle in the synthetic Obelix also matches the one from Asterix. Yaw represents the orientation of a car on the ground plane. 0° corresponds to a car heading away from the camera, 90° – to a car heading to the right.

Figure 6.4 displays the distribution of yaw and pitch across the real Asterix dataset. The clusters correspond to the viewpoints of the 15 cameras that were used as the source for the image data. The values of the yaw group into two large clusters: at 0° and at 180° corresponding to cars seen as driving toward the camera and away from the camera.

Finally, images were annotated with **foreground mask**. The occluded parts of vehicles were not annotated in accordance to the standard protocol [27, 56, 64]. Masks for the synthetic Obelix images include two classes: "foreground" and 'background." The masks for the real Asterix images have an additional class: "don't care," which hides the areas of the image that can not be reliably referred to as either foreground or background [77]. For example, pixels in the blurred area around a car belong to "don't care" area.

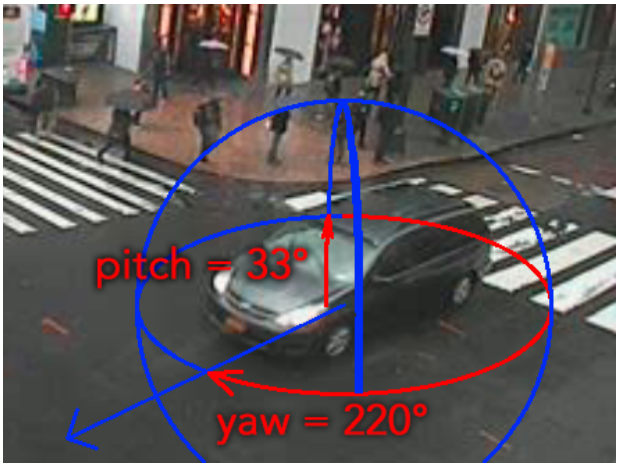
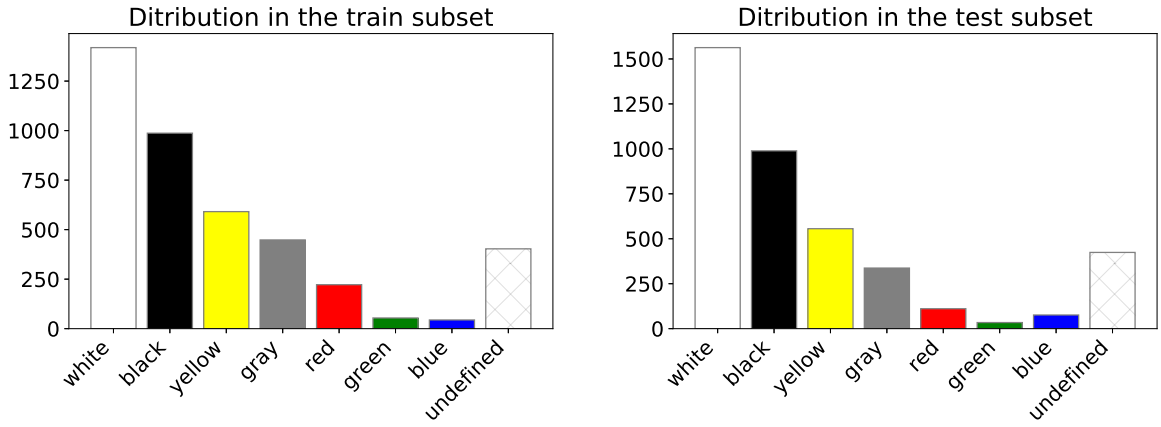


Figure 6.4: **Pitch** is the angle of the ground plane as seen by the camera. Pitch is 0° for cars at the horizon and 90° for cars that are seen top-down. **Yaw** signifies the orientation of a car on the ground plane. 0° is the orientation away from the camera, 90° is heading to the right. This figure reiterates Figure 5.4 from Chapter 5.

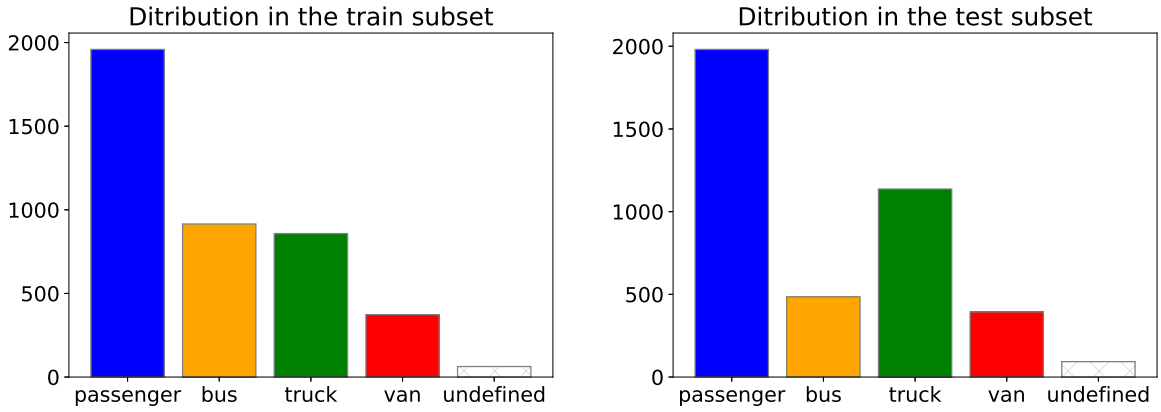
6.3 Splits of Asterix

We split the Asterix dataset into the `train` subset with 4168 cars and `test` subset with 4090 cars. In all experiments in Chapter 8, the `test` subset is used only for evaluation.

In some cases, such as during the stops at a traffic light, sequential frames in the source dataset CityCam have almost identical images of the same car. That means that Asterix has multiple almost identical images of the same car (though low-level details may differ). In order to avoid placing such images into both the `train` and `test` subsets, we made the decision to split the Asterix dataset into the subsets in a simple way: Asterix was cut in the middle forming `train` and `test` subsets, respectively. That in turn resulted in slightly different statistics across the subsets as can be seen from Figure 6.5.



(a) Distribution of colors in NYC dataset.



(b) Distribution of types in NYC dataset.

Figure 6.5: Statistics of the `train` and `test` subsets of Asterix.

6.4 Obelix and its 3D origin CADillac

The reader is reminded that Obelix was generated by rendering 3D car models from the CADillac dataset. Examples of the 3D models can be seen in Figure 4.4 in Chapter 4.

We picked 3D models for CADillac in such a way that the dataset includes various vehicles of types passenger, van, truck, and bus. However, we also included multiple models of emergency services (police, ambulance, and firetrucks). Their number is disproportionally high as compared to the number of emergency vehicles in Asterix. Our motivation is to allow researchers use the synthetic emergency vehicles to learn the corner cases of the real data. Additionally, CADillac includes a few models that are not found on public roads, in particular, military and fiction. They were not rendered into the Obelix dataset.

Obelix images are annotated with extra information that is not available for the real images. First, the labels for car make, year, and model were imported from the CADillac dataset, where available. Furthermore, we collected the information about dimensions of the 3D models in CADillac, namely, length, width, and height. This information is also made available in Obelix. Figure 6.6 presents the distribution of the length of CADillac models. The majority of the models are sedans, as can be seen by the peak at 5 meters. Though there are models as long as 30 meters, only those shorter than 15 meters made its way into Obelix. Longer models in many cases do not fit well into the synthetic 3D scene. Incorporating them into Obelix is left for future work.

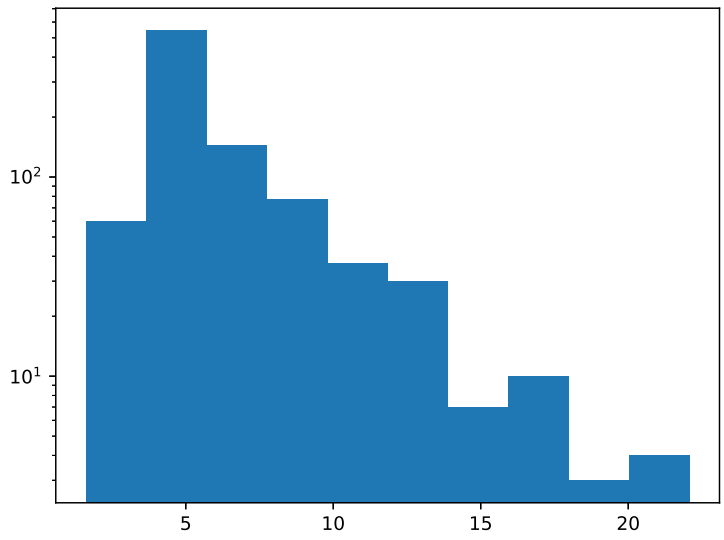


Figure 6.6: Distribution of vehicles in the 3D CADillac dataset by car length (meter).

Chapter 7

Car Detection with Bounding Boxes

This chapter addresses the first question posed in this thesis, namely, detection of vehicles in a low framerate video as bounding boxes.

In the general case, the problem of automatic traffic analysis can be split into detecting vehicles in video frames and tracking the detected vehicles across frames. However, in the case of low framerate video (less than 1 frame per second) tracking-based methods are not applicable and vehicles in different frames must be detected in isolation. Following the categorization of [8], the most relevant methods include 1) background based-detectors and 2) feature-based detectors. The background based-approaches perform well only for sparse traffic, but can be combined with learning-based methods. These include learning with hand-crafted features [6, 7], and more recently with CNNs [10]. In both cases, the learning is performed in a supervised way, and training data needs to be collected and annotated.

In the fixed-camera scenario, the video background can be easily extracted and used for training. In this chapter, we explore an improved method to generate synthetic data. We build hybrid synthetic-real datasets by rendering 3D CAD models as photo-realistic images and overlaying them on real video background (Figure 7.1). We find that the detector shows the best performance when the hybrid synthetic data is combined with real data. We also explore the effect of using the synthetic data in a scenario where training

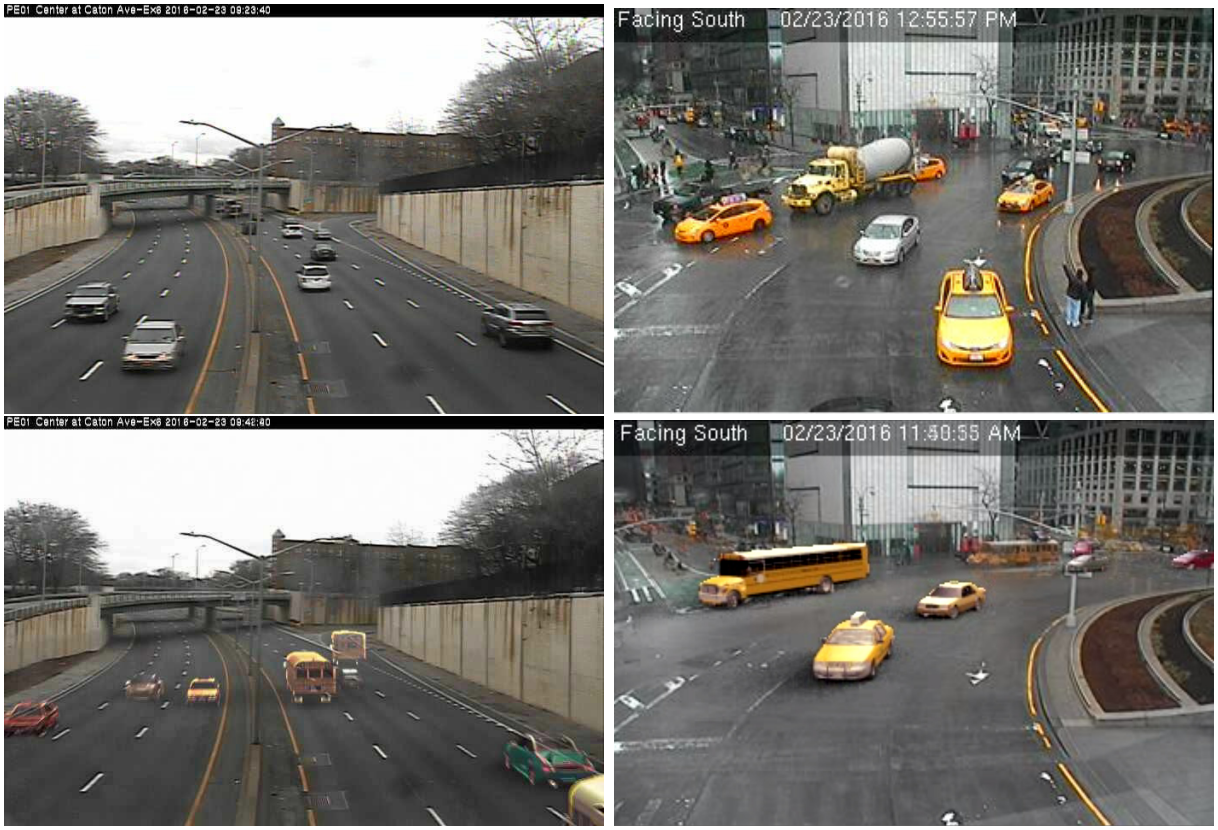


Figure 7.1: Example of real frames (top), synthetic frames (bottom).

has to be done with limited real data. We find that synthetic data allows to replace up to 75% of the real data without a drop in the model performance.

Two features of the fixed-cameras scenario in using photo-realistic renderings immediately simplify the problem:

- Fixed scene geometry constrains vehicle position and viewpoint in the frame. This allows us to render 3D car models only as they would be seen by the camera.
- We can extract the background from the camera video eliminating the need to simulate countless varieties of details, weather, and illumination in the background. Real background also accounts for low-level image features – artifacts introduced by the camera, hard-to-model shadows, and reflections on the road, among others.

With that in mind, we generate synthetic traffic data using the open source Blender rendering engine [71] and the CADillac dataset of the 3D car models, which is described in detail in Chapter 4. These datasets are quantitatively evaluated in the context of object detection. In particular, we use the well-established Faster-RCNN [24] neural network architecture as the detection pipeline. We perform experiments on videos collected from two publicly accessible traffic cameras in New York City [1]. We explore the effect of different proportions of synthetic and real data in the detector performance. Additionally, we explore adversarial learning [40] as a way to improve the quality of the synthetic data.

The four main contributions of this chapter are:

- We illustrate that hybrid real-synthetic data significantly improves detection performance in the fixed-camera scenario. We show that the main advantage of hybrid data that includes real background is to reduce (almost completely eliminate) false positives.
- We describe the proposed algorithm for hybrid synthetic-real data generation. We describe how to combine hybrid and photo-realistic data generation. We also explore further steps to improve the synthetic data by incorporating adversarial training.

- We explore how the proposed synthetic data achieves the main goal – to reduce the need for labeled data. In particular, we find that a model trained on synthetic data and a fraction (25%) of real data exhibits better performance than a model trained on real data alone.
- We illustrate that the model trained on real labeled data does not generalize well to different weather conditions. We conclude that the diversity of road, lighting, and traffic conditions makes data annotations insufficient. We emphasize that a cheap method of synthetic data generation allows to address this problem.

7.1 Related work

Object detection We evaluate our work in the object detection setup, where an algorithm is required to detect vehicles as bounding boxes. CNNs were pioneers in detection in 2013 with the R-CNN [82] system. R-CNN used Selective Search [83] to propose candidate object locations in every image ("proposals") and then used a CNN to classify them into either background or one of the classes. Further work built on it. Later, Fast-RCNN [84] introduces RoI Pooling Layer that combines feature vectors of all proposals from conv-layer and runs fully-connected layers on them directly. Similarly to [85], it allows only a single pass of conv-layers per image. Faster-RCNN [24] replaced proposal generation from 3rd party [83, 86] with a second CNN that uses the same convolutional layers to find class-agnostic proposals, and set a strong baseline as an object detector.

The aforementioned detectors use a subnetwork that generates proposals in order to control the foreground to background examples ratio during training. Conversely, the class of one-stage detectors [26, 87, 88] use a predefined set of "anchors" that serve as an initial guess for bounding boxes in various parts of an image. Each anchor is then adjusted in space to the most probable position of an object and evaluated in terms of the score.

Though one-stage detectors started as faster though less accurate alternatives of two-stage detectors, RetinaNet [26] in 2017 achieved the performance compatible with Faster-RCNN by introducing the focal loss, which suppresses the influence of easy to reject background ROIs into the total loss.

In this work, we use the baseline Faster-RCNN as it is a well-established benchmark for the detection task.

Hybrid synthetic-real data The closest approach to the presented work in the hybrid domain is from [89] where authors train a pedestrian detector in a fixed-camera setup. Similar to our work, renderings of pedestrian models are overlaid onto images of the background. While the scenario is similar, they do not include complex or wide backgrounds with high variation of detected objects appearance, which are common in traffic surveillance scenarios. More importantly, while the authors focus their work on building the best detector and tracker in a scene-specific setup, we are mostly concerned with comparing training on the synthetic data to training on real annotated data.

7.2 Method

In this section, we describe the algorithm to generate a hybrid synthetic-real image, given video from a fixed camera. An overview of the pipeline is depicted in Figure 7.2.

7.2.1 Extracting background

We subtract the foreground from the video, which results in video of an empty road.

First, the foreground in the video is detected. The foreground includes cars, their shadows, and some image artifacts. In simple cases, we employ Gaussian Mixture Models (GMM) [90, 91]. It models each pixel independently to belong to either background or foreground.

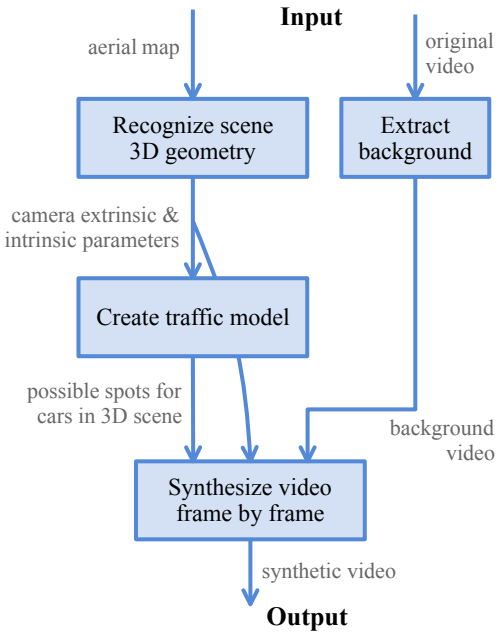


Figure 7.2: Rendering pipeline: 1) the 3D scene geometry (extrinsic and intrinsic camera parameters) is reconstructed; 2) background of original video is generated; 3) creating traffic model of how to distribute cars across the 3D scene; 4) for each frame of the original video a synthetic video is generated frame by frame, as shown in Figure 7.3.

Second, the background video is generated frame by frame with the Formula 7.1. Non-masked areas of each new frame update the background with update rate η:

$$\begin{aligned} \text{background}[t] = \text{background}[t-1] \\ - \eta\,(1 - \text{mask}[t]) \otimes \text{back}[t-1] \\ + \eta\,(1 - \text{mask}[t]) \otimes \text{frame}[t] \end{aligned} \tag{7.1}$$

7.2.2 Recognize scene 3D geometry

Since our goal is to render vehicles as viewed from a specific viewpoint, we first need to learn the scene 3D geometry for every camera in the experiment. Scene 3D geometry includes camera extrinsic parameters (location and rotation) and intrinsic parameters. Under perspective camera model they includes focal length, center offset, skew, and non-linear distortions. For many surveillance cameras we can assume zero center offset, no skew, or non-linear distortions. We work under the assumption of the flat ground.

7.2.3 Create traffic model.

After that, we annotate road lanes to create a traffic model. 3D models of cars are positioned in the virtual scene near centers of lanes. Cars do not change lanes. Distances between cars are chosen automatically and vary from high (sparse traffic) to low, corresponding to a traffic jam.

7.2.4 Synthesizing images.

For each frame of the original video a synthetic frame is generated. Synthesizing a hybrid frame includes four steps (Figure 7.3). We discuss them here in details.

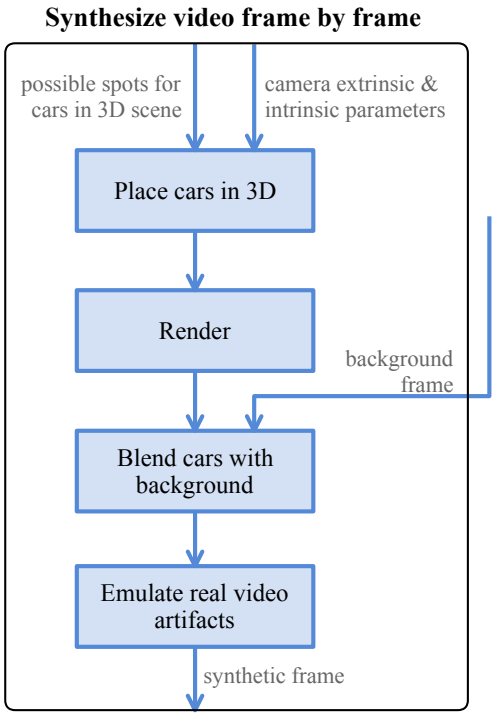


Figure 7.3: Pipeline for synthesizing a frame.

Placing cars in 3D. We make use of the CADillac dataset described in Chapter 4. A number of random 3D CAD vehicle models from the CADillac dataset are arranged in the scene according to the traffic model created before. Lighting conditions in the scene are set up according to the weather in the source video. We modeled weather to be one of following types: sunny, cloudy, wet, or rainy. These types reflect the amount of lighting the scene gets, shadows, and the appearance of the ground that one experienced in video. The ground is made to be reflective for wet and rainy weather. In sunny weather, the angle of the sun is set based on the real position of the sun at the time when the video frame was recorded. Additionally, shadows of surrounding shapes are cast on cars.

Rendering. Finally, cars and the shadows they cast on the road are rendered on the transparent background. For the purposes of data augmentation, we vary the saturation and brightness of the rendered image.

Number of images	Cam. #1	Cam. #2
Synthetic	21800	7440
Training real	875	750
Testing real	925	750

Table 7.1: Number of labeled real and synthetic images in the dataset.

Additionally, we render the scene depth map, which is further used deduce mutual occlusion between cars. If a car is more than 50% occluded, it is omitted from training.

Blending cars with background. At this step, we vary image saturation and brightness to increase data variability. Next, rendered images and overlaid on top of the frame background. Car shadows are blurred to conceal the noise introduced by rendering engine. The web-interface broadcasts video as a series of JPEG-compressed images with sharpened edges. Therefore, in order to emulate artifacts in images, we also sharpen the rendered cars before overlaying them on the background, and re-encode the final image with JPEG algorithm with high compression. Artifacts are further discussed in Section 7.3.6.

7.3 Experiments

7.3.1 Dataset

Real-time data from cameras is publicly available on the NYC Department of Transportation website [1]. However, instead of streamed video, individual video frames are uploaded to the website as JPEG-compressed images. As a result, "videos" have a variable and very low framerate of around 0.5 fps (2 sec. per frame), and individual frames have heavy JPEG compression artifacts. Further, strong edge-sharpening filter and ghost duplicate images distort the picture.

Figure 7.4: Reconstructing scene geometry: left: Google Satellite snapshot; center: rendered snapshot; right: image from camera.

The video data for NYC cameras comes without any meta information except for the street where the camera is located. However, a physical camera is usually easily recognizable in Google Street View images. We then find the camera's location in Google Earth Images. This way camera's 3D coordinates can usually be deduced with the precision of 20 centimeters.

Next, we use an image from a camera to model the scene geometry according to Section 7.2.2 (Figure 7.4). We take a snapshot of the area from the Google Satellite service, and semi-automatically re-create the scene in Blender 3D modelling environment. In particular, we place the Google Satellite snapshot onto the ground plane and place a camera in 3D space relative to it. We aim to find camera extrinsic and intrinsic parameters such that the rendering of the Google Satellite snapshot matches the image from the camera. We assume a perspective camera model, without nonlinear distortions. First, we find the vanishing points in the image and use them to refine camera rotation. Then the height and the focal length are optimized together. X- and Y-coordinates of the camera (longitude and latitude) are usually accurate enough from the start and do not need to be optimized.

We chose two cameras for our experiments. Camera #1 is installed on a highway (resolution 701×480) and presents a typical view from traffic surveillance cameras. The other is located in Manhattan downtown (resolution 352×240). Camera #2 was picked because it overviews a part of a road where vehicles are seen from a variety of angles. We

downloaded two videos which span the time interval from 9.30 AM to 4.30 PM in local time. Videos from both cameras exhibit cloudy and rainy weather.

We annotated 1800 and 1500 frames for the two cameras respectively (Table 7.1.) These labeled frames were split into training and test subsets of equal size. We would desire all weather and lighting conditions in both training and testing subsets, therefore, we first cut the original video into 8 clips Then we assign all clips with even numbers to training and clips with odd numbers to testing. The same car does not appear in any two adjacent clips, which ensures the fair result.

The vehicles present in the video are very unbalanced by types: more than 90% of cars are passenger cars. We leave differentiating between types of passenger cars for future work. The problem is posed as a single-class vehicle detection.

7.3.2 Training details

We trained with the baseline Faster-RCNN model built with VGG-16 [92] underlying architecture. Thus architecture has 13 convolutional and 3 fully-connected layers, and was pretrained with ImageNet. We used "end2end" training scheme, where the proposal and the classification branches of Faster-RCNN network are optimized simultaneously. We refer the reader to the original paper [24] for details of the algorithm.

All models were trained for $30,000$ iterations or until the convergence, whatever comes first. We used the stochastic gradient descent optimization with constant learning rate of 0.001. Training for $30,000$ iterations takes 8 hours with Nvidia Tesla K40 GPU.

7.3.3 Evaluation details

We follow the standard evaluation protocol for the object detection task. A bounding box of a detected car is considered a true positive when its intersection over union (IoU) with one of the ground truth bounding boxes is higher than a threshold. In all our experiments

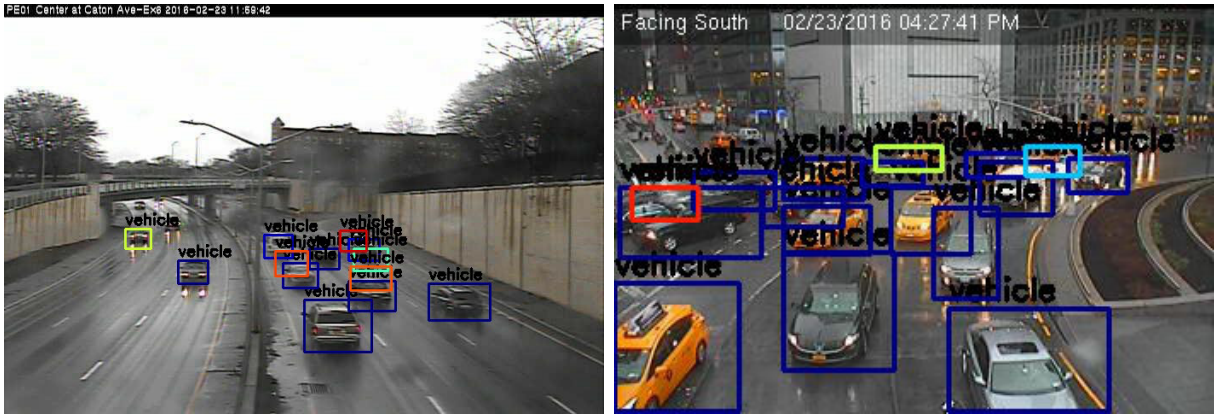


Figure 7.5: Car detection results using synthetic and real training data. Example of real frames (top), synthetic frames (middle) and detections results (bottom).

we set this threshold to 0.5. Precision and recall can be computed from the number of true positives, false positives, and false negatives. The average precision (AP) metrics is then calculated based on the precision-recall curve.

The question arises about how to evaluate detections of tiny cars, some of which are not even labeled by annotators. We could simply filter out ground truth bounding boxes that are smaller than a threshold Δ, but then detections of those filtered tiny cars would be incorrectly counted as false positives.

Instead we slightly modify the algorithm of counting. Correct detections smaller than a threshold Δ are ignored and undetected ground truth objects smaller than Δ do not contribute towards false positives. At the same time, false positive detections of any size are penalized. This scheme is explained in algorithm 1. We experimented with different values of Δ and found that the relative performance of models does not change with different choices of Δ. All results below are reported for $\Delta = 30$ pxl.

Finally, an example of applying the best performing model to one frame from each of the two cameras are presented in Figure 7.5.

Algorithm 1 Small-Object-Aware Evaluation

Input: detections det_i, ground truth, gt_k, IoU threshold μ, minimum size θ

for all det_i **do**

  if exists gt_k such that $IoU(det_i, gt_k) > thesh$ **then**

    mark gt_k as detected

    if size$(det_i) \geq \Delta$ **then**

      tp += 1

    end if

  else

    fp += 1

  end if

end for

for all undetected gt_k **do**

  if size$(gt_k) \geq \Delta$ **then**

    fn += 1

  end if

end for

7.3.4 Synthetic and real training data

In our first experiment, we trained the models with different combinations of real labeled and synthetic images, and evaluated them on real video.

Varying amount of real data. We trained the model on different numbers of real data, with and without synthetic data. The performance is depicted in Figure 7.6. Solid lines stand for training with real data alone, and dashed lines signify the added synthetic data. When all synthetic images are added to training data, the performance improves in three different ways:

1. Performance is higher in each point, when synthetic data is added.
2. The gap between performance of models trained with and without the synthetic data remains wide (2%) for camera #2 as the number of real images increases. That means that even with plenty of labelled data, the synthetic dataset is useful. We leave the question of when this gap is closed to future research.
3. Performance saturates much faster, when synthetic data is added. A synthetic dataset allows to reach the same performance with only a fraction of labelled images. In our case, that is 25%–70% of full real dataset.

Varying amount of synthetic data. Next we answer the question: if we keep a fraction of real labeled data, how much of the synthetic data should be generated. We trained with 10% of real data (about 80 frames) and varying amount of synthetic data. Performance of a model trained with 10% of real data alone is predictably low: 84.1% and 80.8% for the two cameras respectively (arrows on Figure 7.7). Even added only 800 synthetic images, which is close to the full size of the labeled dataset, already improves the performance by 8% and 2%. It can be noted that the performance does not saturate even with $20,000$ synthetic images. Even though it does not change dramatically at that point, fractions of percent can still be gained by simply increasing the synthetic dataset size.

An overview of the results can be found in Table 7.2.

7.3.5 Effect of the change in weather

Next, we investigate if a model trained with real labeled video generalizes well to different weather, therefore, avoiding the need for labeled data under various weather conditions.

We annotated the second video, this time in sunny weather but with strong shadows (Figure 7.9). For brevity, we call the original video "cloudy" and the new one "sunny." The video was cut into training and testing subsets. The experiment was conducted for

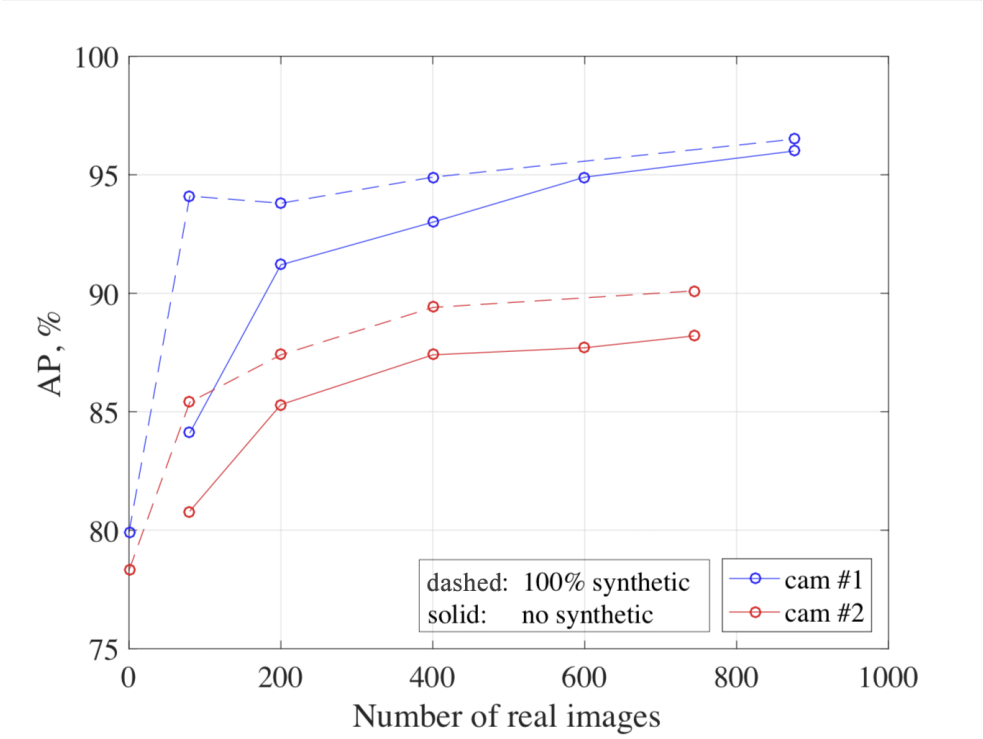


Figure 7.6: Average Precision (AP) for models trained with only real data (solid) and with a mix of real and 100%-synthetic data (dashed). For both cameras, the real-synthetic mix performs better than real data alone. Also, models trained with the real-synthetic mix saturate much faster than models trained with real data, allowing to use a fraction of real data to achieve desired performance.

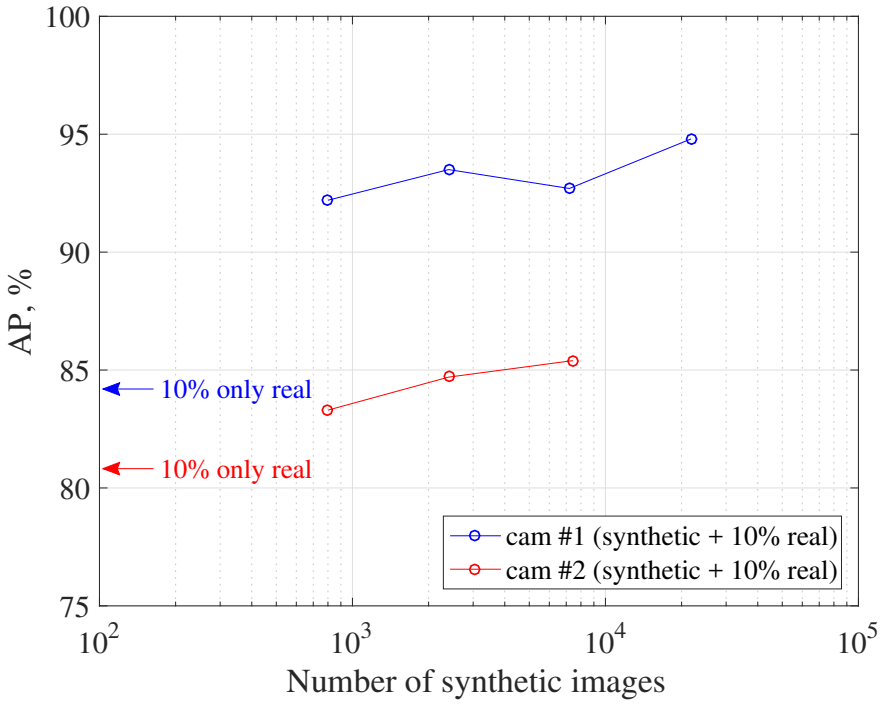


Figure 7.7: Average Precision (AP) for models trained with a mix of 10% of real and the varying number of synthetic images. Note the log-scale in x-axis. The first point corresponds to the number of synthetic frames equal to the 100% of labelled real frames. For both cameras, performance improves with higher number of synthetic images, although the performance slowly reaches a plateau.

Training Data	Cam. #1	Cam. #2
# synthetic frames	21,700	7,440
# real frames	800	800
Synthetic	79.9	78.3
Real	95.0	88.2
Synthetic + Real	**96.5**	**90.1**
10% Real	84.1	80.8
Synthetic + 10% Real	94.8	85.4

Table 7.2: Average precision (%) for various combinations of synthetic and real training data.

Figure 7.8: Example detections. Models are trained with synthetic + real data. Confidence of detections increase from red to blue.

Training Data	Cam. #1	Cam. #2
Synthetic trained on sunny weather	71.6%	72.4%
Real trained on cloudy weather	65.3%	79.2%

Table 7.3: Average precision evaluated in "sunny" weather.

both cameras.

We trained the first model using real data from the "cloudy" video. After that we synthesized a dataset with the "sunny" video, and trained the second model with it. Then we evaluated both models on the "sunny" video. The results presented in Table 7.3 show that the performance of the model trained on **real** data drops significantly. In particular, the performance dropped from 95.0% to 65.3% for camera #1 and from 88.2% to 79.2% for camera #2. The drop in the performance is wider for camera #1, suggesting that the difference between the two videos is more dramatic for camera #1 than for camera #2. As a result, the model trained on purely **synthetic** data outperformed the one trained on real data by 6.3% for camera #1 but was behind by 6.8% for camera #2. This experiment suggests that using synthetic data for new weather conditions in some cases may be more preferable than adapting a model trained on real data but in different conditions.

7.3.6 Effect of artifacts

Finally, we explore the influence of camera-introduced artifacts on performance. The baseline rendering algorithm simulates of edge-sharpening effect and JPEG compression artifacts that we observe in real data. We turned off both effects in turn (Figure 7.10) and trained models with purely synthetic data.

We discovered that when either JPEG compression effect or edge sharpening is turned off, the model loses $0-2\%$ of Average Precision, however, the effect varies between videos.

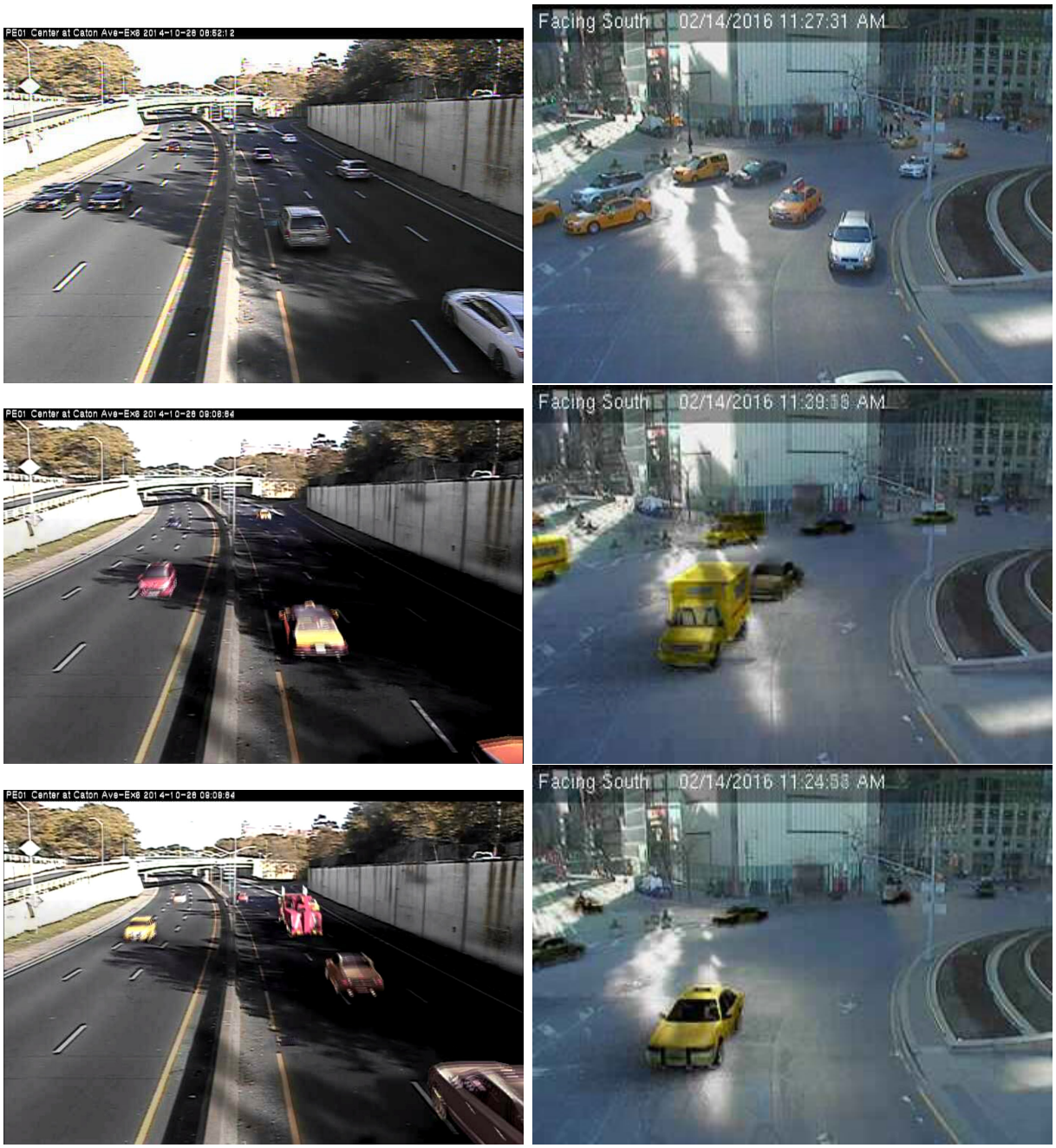


Figure 7.9: "Sunny" videos. Top: real frames; middle and bottom: synthetic frames.

Figure 7.10: Top: baseline rendered image; bottom: without edge sharpening

Figure 7.11: Synthetic patches (top) and their enhancements with GAN (bottom).

7.3.7 More realistic data

Finally, we explore if image realism can be enhanced automatically, in a domain adaptation setting. We employ the method described in [36], where authors bridge the gap between pixel distribution in simulated and real image with adversarial training [40] for the purposes of using synthetic data for training. Similarly to the usual GAN setup [36], the generator outputs a synthetic image that the discriminator tries to distinguish from real images. However, the generator receives a synthetic image on input instead of a vector with noise. The adversarial loss is complimented by the L_1 loss between an improved image and its synthetic original.

We used [36] without any changes in the network architecture. First, we extracted cropped synthetic cars from rendered videos and real cars from original videos, trained the network, and processed all the synthetic cars, and then paste them right back into the video. Examples of original synthetic cars and processed cars are presented in Figure 7.11. All cars are desaturated, which is reasonable for a video of a cloudy day (Figure 7.1). It can also be seen that the back lights of the car on the left were increased in size, while the back lights of the car on the right were repainted from red to gray. The latter can be explained by the fact that in most of the video the cars were driving in daytime with their lights off. Cars also appear blurred, which is a common drawback of using L_1 loss.

Average Precision	Cam. #1
All synthetic	79.9
Refined synthetic	82.9
All synthetic + real	96.5
Refined synthetic + real	95.4

Table 7.4: Performance after refining synthetic patches of cars.

As opposed to the object detection setup in our work, the task in [36] is image recognition. In particular, the refining network receives input images of dimensions 55×35 pixels. We cropped all car patches from our video, scaled them to 55×35, refined them with a pre-trained model, scaled them back, and pasted into their original locations. To avoid the loss of quality when scaling up, we only used car patches with $40 < width < 55$ pixels. The boundaries of replaced patches inside video frames are sometimes visible, the network may take them as clues for car detection. To overcome this problem, we only refined half of the cars, leaving the other half unchanged.

After that we trained two networks for camera #1: with synthetic data only, and with the combination of synthetic and real data. The results are presented in Table 7.4. As compared to the original synthetic video, training with the refined video showed an improved performance (82.9 vs 79.9). It is predictable since information about real data is now included in synthetic images. However, when training with both synthetic and real data, the performance drops by 1% (95.4 vs 96.5), which may mean that the necessary information about real images can be extracted from labeled real data alone. We leave this analysis to the future work.

Finally, we note three challenges of the presented approach: (1) semi-manually reconstructing 3D scene geometry for every camera, (2) generating high-quality background frames for dense traffic, and (3) semi-manual adjustment of the lighting for rendering.

Chapter 8

Car Background Segmentation

In this Chapter, we proceed to the analysis of individual cars one-by-one, in particular, we consider the task of foreground semantic segmentation. We train a segmentation model on image patches of size 64×64 with a single car in each image, and again we conduct experiments on synthetic and real data. In Chapter 6, we described the twin datasets Asterix and Obelisk to be used for this task.

8.1 Background segmentation

The background mask in a car is a valuable source of information about this car on the road. Among other tasks, it allows to automatically determine if a car has broken a traffic rule, such as whether it has crossed the double lane or has crosses the stop line before a stop sign. Existing solutions are yet not perfect [93], therefore, segmenting the image background efficiently remains an important problem to solve.

Video-based methods of background segmentation are not robust in our case due to the very low framerate of our video (0.3 to 1 frame per second.) Instead we segment out the background from individual images using convolutional neural networks.

We formulate the task as semantic image segmentation with two classes – "car " and

Trained on	Avg. IoU, %
Asterix (real)	98.4
Obelix (synthetic)	95.8

Table 8.1: Performance of the best models trained on the real data and on the synthetic data.

"background". We address it with a popular Dilated Residual Network [32] architecture, in particular, we experiment with its DRN-D flavor. Dilated Residual Network (DRN) uses dilated convolutional kernels instead of regular, dense convolutions. Dilated kernels sample the input feature map at a regular dilated grid, thus effectively increasing the receptive field of the convolutional layer. On the other hand, they have the same number of tuneable parameters, therefore, their usage does not increase the size of the network. At the same time, increasing of the receptive field of a dense convolutional layer via increasing the radius does result in a higher number of tuneable parameters.

For every experiment below, we used the Stochastic Gradient Descent (SGD) optimization scheme with the momentum 0.9 and weight decay $2e-5$. The training was performed with the learning rate set to 0.001. The best results was achieved by setting the batch size to 10. Setting a higher learning rate or decreasing the batch size resulted in greater training times, while further increasing the learning rate or training with a batch size equal to 100 caused poor convergence of the model, in accordance with [94].

Every model is evaluated on the `test` subset of Asterix, that has $4,090$ labelled images. Asterix-`test` is not used for training in any experiment. The segmentation result is evaluated in terms of the Intersection over Union (IoU) for the class "car". We present both the average IoU across the dataset, as well as its distribution across images. The distribution allows us to separate predictions with slight inaccuracies in the foreground mask from predictions where a model failed to find a car in the image. As much as we

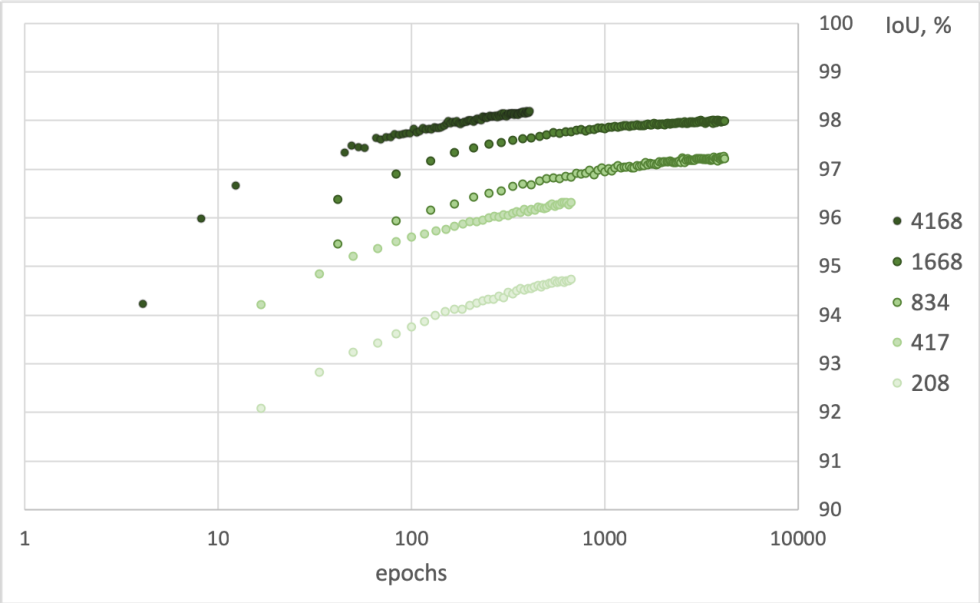


Figure 8.1: IoU on test dataset Asterix-`test` at different epochs indicates no overfitting. Note the log scale on x-axis.

would like the foreground mask to be accurate, it is the latter case that is of the most interest for us.

8.2 Training the model on real data

First, we set the baseline by training the segmentation model on the (real) Asterix-`train` dataset, that has 4,168 labelled images, and on its subsets of 1668, 834, 417, and 208 images. Figure 8.1 presents the average IoU evaluated on the *test* dataset Asterix-`test` at different points of training. Note the log scale on the x-axis. This figure demonstrates that the training converges in every case without signs of overfitting.

The average IoU of the strongest model reaches 98.4%. It is so close to the perfect 100% due to the good performance of the model but also because of the conservative "don't care" class in the ground truth. However, the saturation of IoU does not pose a problem for our task of comparing the real and synthetic data.

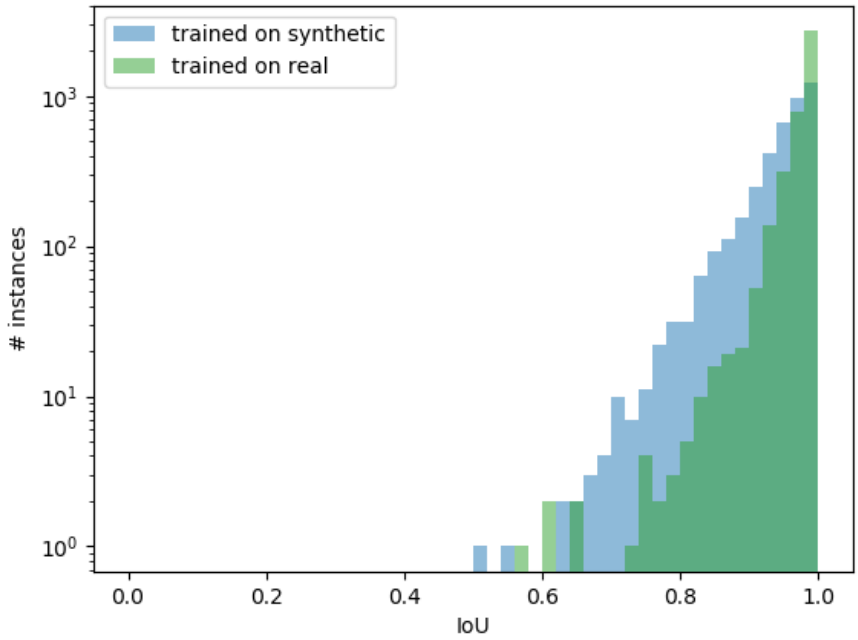


Figure 8.2: Histogram of IoU across images in Asterix-`test`. Note the log scale.

In order to examine the performance of the network on individual images, we present the distribution of IoU across images in Figure 8.2. It illustrates that in the majority of the cases, the network segments the foreground almost perfectly, and only in a small number of cases a big part of an image is mislabelled, which indicates that the orientation or the type of the car was not recognized.

For the qualitative analysis, we present randomly sampled segmentation results in Figure 8.3a. At the same time, Figure 8.3b demonstrates failure cases – images that yield the lowest scores of IoU. The two main sources of errors can be attributed to occlusions and cars being displaced into a corner of an image. The occlusions are not recognized well because of a small number very occluded cars in the training dataset Asterix-`train`. Section 8.4 further elaborates on this.

A concern of the practical importance is the dependency of the model performance on the size of the training dataset. Smaller training datasets have less diverse training

examples, which deteriorates the performance. Figure 8.4 compares the model performance when trained on different percentages of Asterix-`train`. The performance predictably drops with the decrease of the dataset size, however, it still stays at a reasonable level even with the small number of training examples.

8.3 Training the model on synthetic data

Now that the baseline has been set, we proceed to explore the performance of the same segmentation architecture, this time trained on the synthetic dataset Obelix. Obelix contains $80,145$ images, which is 20 times the size of Asterix-`train`. In the following experiments, we aim to explore the domain gap between Asterix and Obelix and find which properties of synthetic data may affect it.

The average IoU of our best model trained on Obelix reaches 95.8%. Even though it is 2.6% lower than IoU of the real data (Table 8.1), this value is still high. For qualitative analysis, random image samples and their predicted masks are displayed in Figure 8.5a.

One hard to simulate feature of a real image is low-level artifacts, including motion blur, ghost image, and JPEG compression artifacts (Figure 8.6.) It is natural to hypothesize that, because of these artifacts, models trained on synthetic data would fail to predict accurate car boundaries. This hypothesis can be verified with a histogram of IoU across images shown in Figure 8.2 (note the log scale.) In particular, one may notice less predictions with a close to perfect score than for the model trained on Asterix. A more important concern is the higher number of samples with a lower score. A low IoU corresponds to a case where the network failed to recognize a part of a car and, therefore, misinterpreted the car orientation or position. The most poorly predicted masks are presented in Figure 8.3b.

The influence of the dataset size on the model performance is evaluated in Figure 8.4. It can be seen that the performance increases with bigger synthetic datasets, though the

(a) random examples

(b) failure cases

Figure 8.3: Predictions of the segmentation network trained on real data.

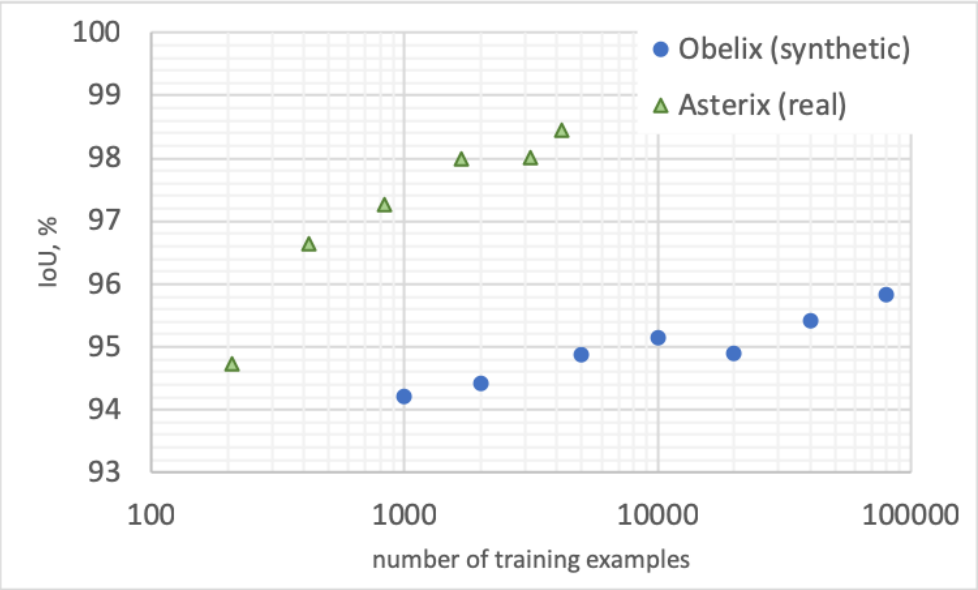


Figure 8.4: Performance of models trained on varying number of synthetic and real images.

gain becomes low at large sizes. At the same time, the synthetic data is zero-cost and, therefore, generating training data on the fly can be considered for practical applications.

We also qualitatively evaluate the performance of the model trained with synthetic data on another publicly available dataset of surveillance camera images, specifically UA-DETRAC [14]. Just like when building Asterix dataset, we expanded bounding boxes of vehicles in UA-DETRAC by 20% in all directions, cropped them out, and resized the cropped patches to 64×64 pixels. Then we applied the model trained on our synthetic Obelix dataset to segment the foreground in these patches. Random samples and their predicted foreground masks are displayed in Figure 8.7,a and selected worst cases – in Figure 8.7,b. UA-DETRAC is not annotated with background masks, therefore, the result can not be evaluated quantitatively. Qualitatively, most predictions appear reasonable, which implies that Obelix generalizes well to other surveillance cameras.

(a) random examples

(b) failure cases

Figure 8.5: Predictions of the segmentation network trained on synthetic data.

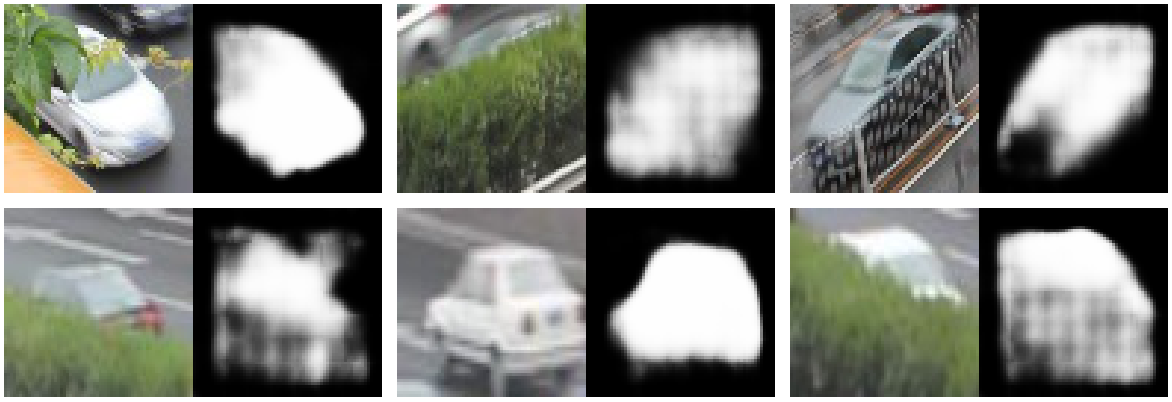
Figure 8.6: Artifacts in real data. (a): JPEG compression; (b): ghost effect in moving objects.

8.3.1 Ablation study

Finally, we evaluate the importance of different components of the rendering pipeline. In particular, the main car model, the occluding models, the weather effects, and the background are selected during the rendering process. The reader is referred to Chapter 4 for the details of our pipeline. We change the pipeline to exclude one component at a time and train the segmentation model on the resulting rendered dataset. By looking at the drop in the performance, we can evaluate the influence of that component. First, we render only the central car without any other cars in front or behind it to evaluate the effect of occlusions. Then, we remove the texture from the planes in our virtual scene, leaving the plain gray color as the background. Samples of rendered images for this experiment are presented in Figure 8.8a. Finally, we evaluate the importance of photo-realism by rendering a collection of "broken" 3D models that are described in more details in Chapter 4. Rendered examples of broken models can be seen in Figure 8.8b.

We generated 8K synthetic images for each of these experiments. The results of the evaluation on Asterix-`test` are presented in Table 8.2. We compare the results with the model trained on a subset of 8K images from Obelix.

(a) random examples



(b) selected failure cases

Figure 8.7: Segmentation of UA-DETRAC with the model trained on synthetic data.

(a) 3D models are rendered on the gray background

(b) broken 3D models have issues with the mesh, windows opacity, color, or detalization

Figure 8.8: Examples of images rendered for the ablation study.

Trained with	Avg. IoU, %	Change from baseline
Obelix, 8K (baseline)	95.2	0
No background	94.6	-0.6
No occluding cars	94.9	-0.3
Broken 3D models	95.8	+0.6

Table 8.2: The study of the importance of different rendering steps. In each experiment, the training dataset has 8K images.

First of all, the lack of textured background significantly reduces the performance. This can be expected, given that the network should learn to model both the foreground and the background. The reader is referred to Figure 8.9 that presents random predictions and all the failure cases where the metrics IoU < 0.7. One can notice that in most of the failure cases, the car has its head lights on, however, it is hard to draw conclusions from that.

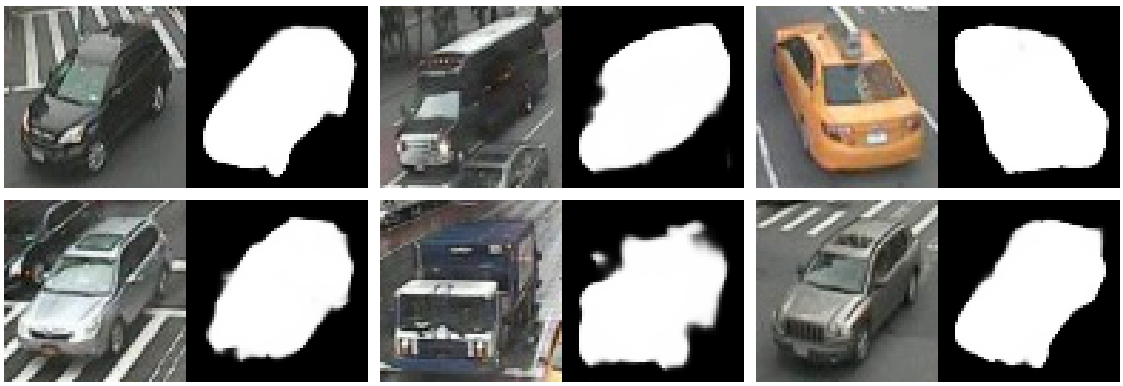
On the other hand, the absence of occluding vehicles does not change the performance. This can be explained by the fact that Asterix-`test` does not have many occluded cars. Those occluded cars that are in Asterix-`test` are in fact not recognized well. We again demonstrate the failure cases with IoU < 0.7 in Figure 8.10b. One can notice that 7 out of the 13 errors cases are due to the occlusions.

Finally, "broken" models unexpectedly showed the performance better than the baseline with "clean" photo-realistic models. This leads us to the conclusion that ML models trained on synthetic images overfit to the artifacts of rendering, and some data augmentation in the form of unrealistic but recognizable car appearances is necessary. Again, we demonstrate random samples and the worst predicted cases in Figure 8.11.

8.4 Use of synthetic data for rare cases

The world we live in is visually very diverse, and one may come across some situations only several times in the lifetime. For example, it is rare that drivers violate traffic rules. In other words, the distribution of the visual data has a very long tail. Models trained on real data show better performance than those trained on synthetic data only when sufficient amounts of annotated training data are available. At the same time, it is difficult to collect and annotate data for cases that happen rarely, and synthetic data presents an opportunity to solve these cases at lower cost and with acceptable quality.

In this section, we illustrate this claim on a toy example: occluded cars. Asterix

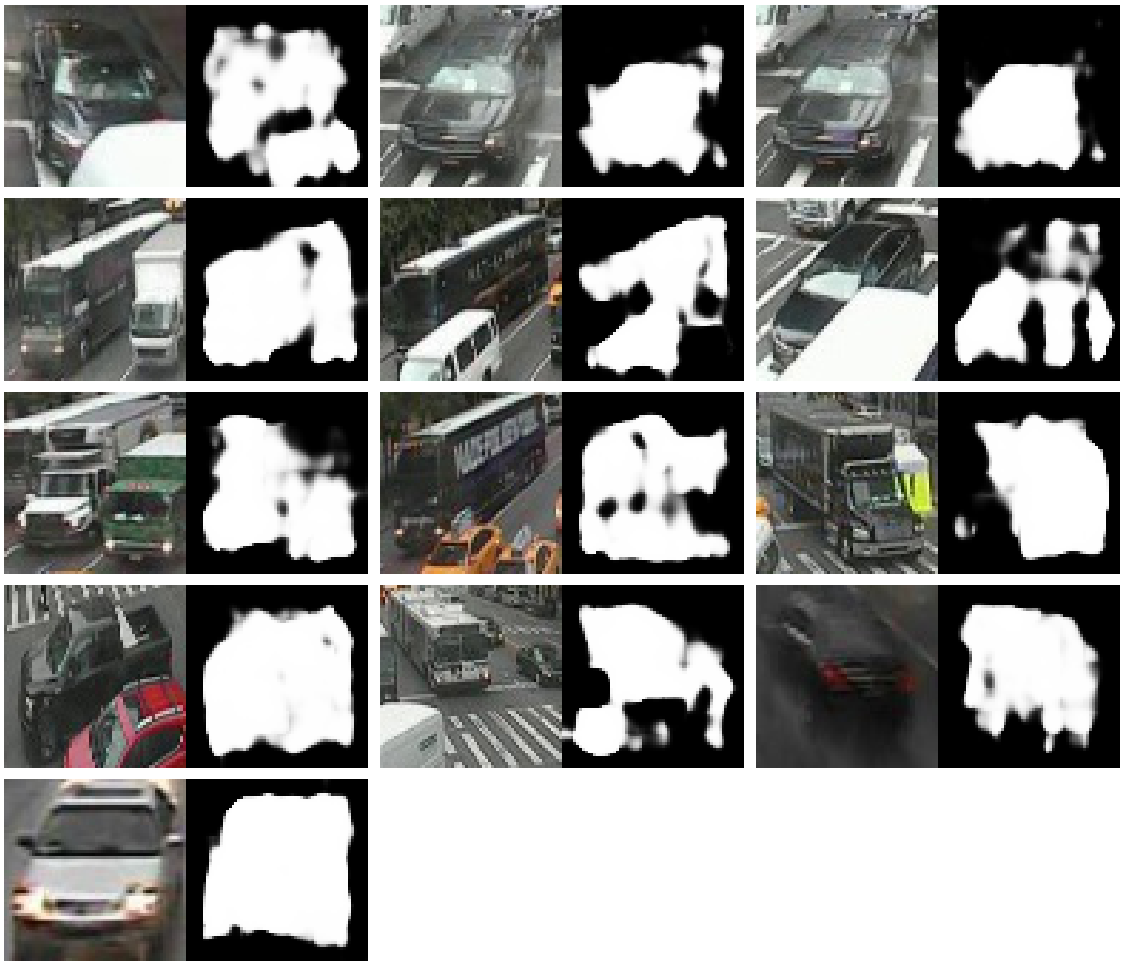


(a) random samples

(b) all failure cases, where IoU < 0.7

Figure 8.9: Segmentation of real-`test` images when trained on synthetic data rendered with **gray background**.

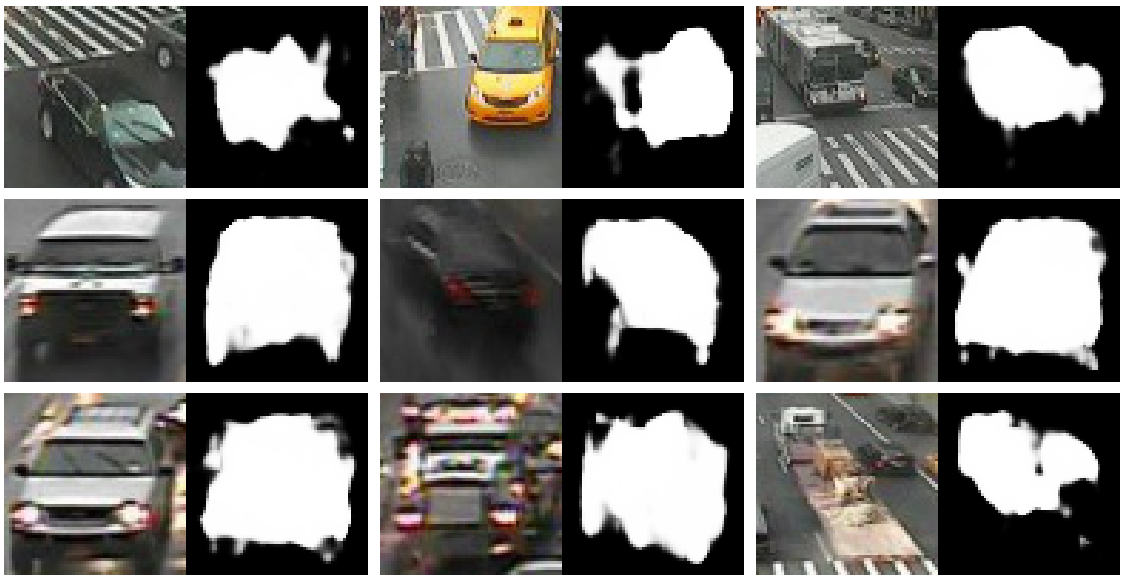
(a) random samples



(b) all failure cases, where IoU < 0.7

Figure 8.10: Segmentation of real-`test` when trained on synthetic data rendered **with no occluding vehicles**.

(a) random samples



(b) all failure cases, where IoU < 0.7

Figure 8.11: Segmentation of real-`test` when trained on synthetic data rendered with **broken 3D models**.

dataset does not contain cars that are very occluded. Therefore, we consider high degree of occlusion as a rare case. We collected an additional dataset with occluded examples and use it to qualitatively evaluate the model trained on Asterix and on Obelix. The segmented examples are presented in Figure 8.12.

Asterix does not contain highly occluded images, and the model trained on Asterix fails to recognize the occlusions in the Figure 8.12a. At the same time, Obelix does have highly occluded cars and Figure 8.12b demonstrates that the model recognized occlusions in most cases. That illustrates both the limits of the generalization power of segmentation models and the benefits of using synthetic training data to model rare cases.

8.5 Segmentation with domain adaptation

No matter how photo-realistic the synthetic dataset is, there is always a gap between the synthetic and the real image domain. This problem is known as the domain gap, and methods known as domain adaptation aim to close it. These methods include two main categories – pixel-level adaptation and feature-level adaptation.

Here, we build on a recent feature-level adaptation method Maximum Classifier Discrepancy for Domain Adaptation (MCD_DA) [46]. The algorithm includes a feature generator and two classifiers that are all trained together. The feature generator G creates features from a synthetic and a real image, and both classifiers C1 and C2 are trained for the same task to classify synthetic images in the supervised way. At this point the real images are not used. The key idea of the algorithm is that C1 and C2 learn a different decision boundary on their supervised task if initialized differently. Now we add the real images into the pipeline, and claim that the domain gap is determined on how different the predictions of C1 and C2 are on the real data. By enforcing the predictions to be the same, whether correct or not, we hope to close the gap between the domains.

(a) model trained on the real Asterix

(b) model trained on the synthetic Obelix

Figure 8.12: Segmentation of additionally collected images (not a part of Asterix) with **high occlusion**. Asterix does not contain highly occluded images, and the model trained on Asterix fails to recognize the occlusions in the figure. At the same time, Obelix does have highly occluded cars and the model did recognize occlusions in most cases.

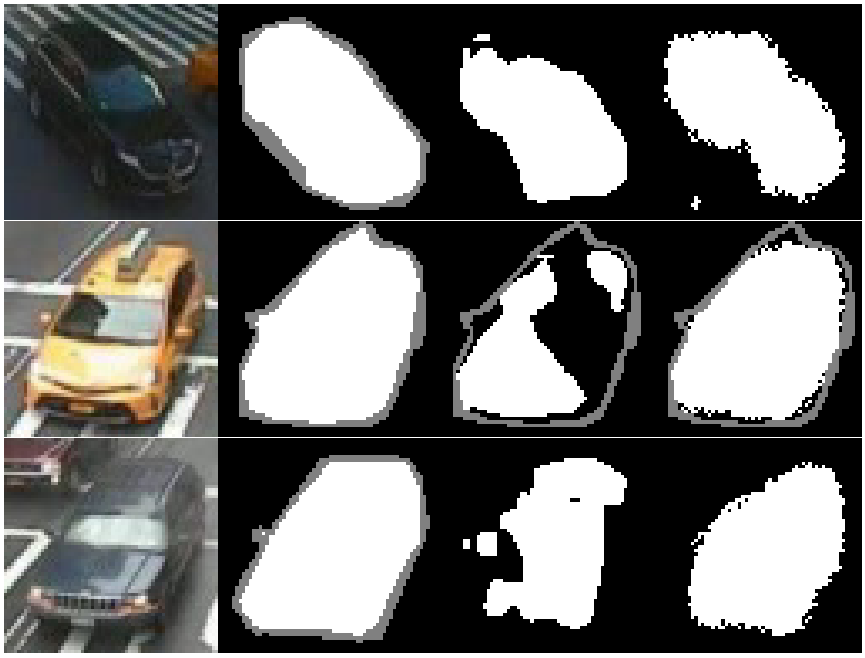


Figure 8.13: Right to left: (a): image; (b): ground truth; (c): model trained on Obelix; (d): model trained on Obelix with domain adaptation to Asterix-`train`.

We train MCD_DA on a subset of synthetic Obelix images and their ground truth masks, while adapting to real Asterix images *without* their ground truth masks. In other words, we adapt from the synthetic to the real domain trying to preserve the semantic information – in our case, the foreground mask. Training is performed with the default values of hyperparameters in MCD_DA package. Exploration of the space of hyperparameters is beyond the scope of this work. It is important to note that the underlying segmentation network is the same as without the adaptation.

As before, the result is evaluated on Asterix-`test`. This time, we modified the ground truth mask in Asterix-`test` so that much of the blurred "don't care" regions become a part of either the background or the mask class. We do so in order to evaluate the effect of domain adaptation algorithm on the quality of the inter-class boundary.

Figure 8.13 dislays selected examples where MCD_DA predictions outperform predictions of the model trained on synthetic images without adaptation. Qualitatively, the last

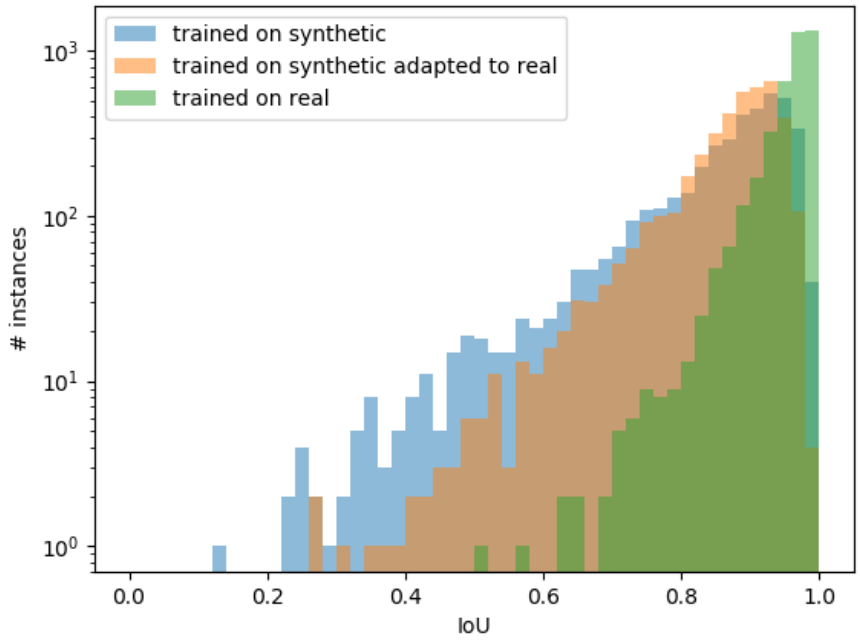


Figure 8.14: Histogram of IoU across images in real-`test`. Dataset is evaluated with a smaller area of "don't care" class (the gray area in Figure 8.13) for the purpose to highlight the non-perfect boundaries of MCD_DA predictions and their effect on the histograms. Note the log scale.

column (d) in Figure 8.13 shows that MCD_DA produces a rough inter-class boundary. This effect can be explained by the adversarial game that the classifiers C1 and C2 play during training.

Quantitatively, Figure 8.14 presents histograms of IoU from the three models: trained on the synthetic data, trained on real data, and trained on synthetic data with domain adaptation to real data. The effect of the domain adaptation is twofold: (1) the number of almost-perfect predictions decreased because of the rough inter-class boundaries and (2) the number of failure cases decreased as well. Assuming it is more important not to make major mistakes, we consider effect (2) to be more important. The overall IoU score increased from 86.2% when training on synthetic data to 88.2% when using domain

Trained on	Avg. IoU, %
Asterix (real)	95.7
Obelix (synthetic)	86.2
Obelix (synthetic) adapted to Asterix (real)	88.2

Table 8.3: Domain adaptation reduces the gap between synthetic and real data.

adaptation, as presented in Table 8.3.

8.6 Conclusion

In this chapter, we consider car foreground segmentation as an example of a task that requires fine-grained understanding of an image. We train a segmentation model on image patches of size 64×64 with one single car in the image. Again, we conduct experiments on synthetic and real data.

The model trained on $4,090$ real images from the Asterix dataset reached the performance of 98.4% in terms of Intersection over Union (IoU). A model trained on about $80,000$ synthetic images from the Obelix dataset displayed the performance of IoU = 95.9%. Though it does not match the performance of the real-data model, the IoU distribution indicates that much of the difference can be attributed to bad generalization of the synthetic-data model to blurred boundaries in real images, while in most cases the mask follows the general shape of the car. The qualitative evaluation of the model on UA-DETRAC dataset confirmed the generalization abilities of the synthetic-data model.

We conducted extensive experiments on evaluating the influence of various steps in the process of synthetic data generation on the result. We found that using a plain gray background dropped the performance by 0.6%, and rendering image only with a cloudy lighting dropped it by 0.3%. At the same time, surprisingly, rendering 3D models with

broken texture, increased the performance by 0.6% instead of decreasing it. That suggests that broken 3D models decrease the bias introduced with rendering 3D models, but future work is planned to explore this effect.

Next, we demonstrate the effectiveness of the synthetic data to learn rare cases, where the real data is not available. In our experiments, we consider drastic car occlusion as a rare case. We collected images with highly-occluded cars and tested the models on them. We found that real-data models (which were not trained on such occlusions) confuse car parts in such images, as opposed to the synthetic-data models.

Finally, we demonstrated the ability of domain adaptation algorithms to reduce the gap between the synthetic and the real data.

Chapter 9

Conclusion

Municipalities around the world are increasingly equipped with low-resolution traffic surveillance cameras. They are used by municipal public services for the purposes of monitoring road conditions, quickly responding to road accidents, and studying patterns of traffic. The public interface to these cameras would typically provide video of low resolution and low framerate. Such video is a valuable source of data for anyone with the goal of training machine learning models for tasks ranging from automatic traffic density recognition to finding free parking spots. Currently, the best performing machine learning models are trained in a supervised way and require ground truth annotations. Commercial automated traffic analysis systems use expensive work of human annotators to collect ground truth labels in videos from different cameras, in different lighting and weather conditions, for various rare road traffic situations and vehicle appearances.

In this thesis, we propose to democratize machine learning and reduce the amounts of required annotations by using synthetic computer-generated data for the specific scenario of traffic surveillance cameras. Below, we first describe the components of the system we are releasing and then discuss the experiments we have performed to demonstrate the usefulness of synthetic data with the help of this system.

9.1 Released system

We propose a system that includes three components: (1) a software system for generating synthetic data, (2) a dataset of synthetic and real images, as well as a dataset of 3D car models, and (3) a dataset management toolbox that facilitates working with annotations.

The system for generating synthetic data TrafficBlender is described in Chapter 4. It creates a synthetic image by building a virtual 3D scene and then rendering it as an image with either a real or synthetic background. In particular, in order to create one image, the system first generates a synthetic scene: several 3D car models are picked from a pool of 3D CAD models and placed into the scene according to a predefined pattern. The pattern may represent traffic in a generic road as in Chapter 4 or it may emulate traffic in a real road seen by a specific traffic camera as in Chapter 7. The 3D scene is then rendered as an image. In Chapter 4, we render 3D models on synthetic textured background, while in Chapter 7, we overlay the rendered car on a real video frame from a traffic camera to get a hybrid synthetic-real image. While most synthetic datasets for the self-driving scenario were created by rendering virtual 3D scenes, our system is the first to support hybrid images in road traffic camera scenarios.

Similar to other engines used to create synthetic datasets, our system is capable of simulating typical weather conditions, namely, a sunny, a cloudy, or a rainy day. The system uses the Blender engine [71] as the backend for rendering. We can generate on the order of $100,000$ images per day on a multicore CPU.

Released datasets: CADillac, Asterix & Obelix. We release three datasets aimed to evaluate the synthetic data for training machine learning models in the scenario of a low resolution road traffic camera.

The first dataset, named **CADillac**, is discussed in Chapter 4. CADillac is a dataset

of more than 1,000 CAD car models that were collected from the 3DWarehouse[1] public repository. It includes high-quality vehicles of various colors, types, and sizes. For example, we assembled a large collection of emergency service vehicles, such as police, ambulance, and fire department trucks. Models are annotated with the information about the car make, year, and model, as well as car dimensions, type, and color. CADillac is to be used in the rendering process with the rendering system discussed above.

The second and third datasets **Asterix & Obelix** are presented in Chapter 6. Asterix and Obelix are complimentary to each other. They both consist of images of individual cars as seen by different traffic cameras, under a variety of traffic, weather, and lighting conditions. Asterix is made of real annotated images, while Obelix is made of synthetic data generated as described in Chapter 4. As compared to existing datasets of videos from surveillance cameras, Asterix & Obelix do not contain full video frames, but rather cropped images of individual cars. Unlike other datasets, we also provide pixel-level annotations in the form of the foreground mask. Moreover, we provide the yaw and pitch angles that together define the car 3D orientation, while other datasets provide only discrete orientation labels, such as "front" or "side."

Dataset management toolbox Shuffler. Finally, collecting images for a dataset and managing an annotation team is a challenge without an effective tool to manage data and annotations. In Chapter 3 we propose (1) a SQL database schema for storing annotations to datasets common in computer vision and (2) a dataset management tool Shuffler, which supports annotations from the moment they are created to the moment they are loaded into a Machine Learning framework. Throughout this thesis, we describe how Shuffler was used for operations with annotations. Chapter 4 discusses the use of Shuffler in building Obelix, and Chapter 5 details processing labels obtained from our annotation team.

[1] `https://3dwarehouse.sketchup.com`

9.2 Experiments

We conduct thorough experiments to evaluate if synthetic image data is useful in training machine learning models. In particular, we consider two machine learning tasks: vehicle detection with bounding boxes and then semantic foreground segmentation for each individual car.

Vehicle detection. We started by training the Faster-RCNN neural network architecture to detect cars as bounding boxes in videos from two cameras in New York City (Chapter 7). We trained the network with real data, with synthetic data, and with their combination and then compared the model performance on different test datasets. The performance was compared with the Intersection over Union (IoU). In these experiments, we used the hybrid images described above for the role synthetic data. The reader is reminded that hybrid images are created by rendering 3D car models on the background of frames from real video.

First, we compared models trained on about 800 real images and $21,800$ synthetic images for one camera and on about 800 real images and $7,440$ synthetic images for another camera. As expected, models trained on synthetic data above performed considerably worse: 79.9% vs 95% for camera #1 and 78.3% vs 88.2% for camera #2. However, when we combined the synthetic and the real data, the performance of the models went up to 96.5% and 90.1%, that is, almost 2% higher than the performance of the models trained on real data alone.

Furthermore, we consider a scenario where the real data is very limited. In particular, we use only 10% of the real dataset of about 800 images for training. In that case, the performance of the models drops by about 10% for both cameras, to 84.1% and 80.8% respectively. However, if we add the full synthetic collection to the training, the performance goes back up to 94.8% and 85.4%, which is almost the original level of 95% for camera #1

and close to the original 88.2% for camera #2.

These experiments demonstrate that synthetic data improves the performance of a detection model even in scenarios when annotated real data is accessible (800 labelled images). At the same time, in those cases when annotated real data is very limited (80 labelled images), e.g., when bootstrapping a model in a new environment, a synthetic dataset can become an effective tool to quickly improve the performance of the model.

Foreground segmentation. Then, in Chapter 8 we proceed to the analysis of individual cars one-by-one, in particular, we consider the task of the foreground semantic segmentation. In order to do that, we train a segmentation model on image patches of size 64×64 with one single car in the image. Again, we conduct experiments on synthetic and real data. In the previous chapter, we described how we built the twin datasets Asterix and Obelisk for exactly this task. We use a Dilated Residual Network as a backbone for the segmentation network and Intersection over Union (IoU) for the "car" class as evaluation metrics.

The model trained on about $8,000$ real images reached the performance of 98.4% of IoU. Next, a model trained on about $80,000$ synthetic images displays the performance of 95.9%. Though that does not match the performance of the real-data model, the data indicated that much of the difference in the performance can be attributed to bad generalization of the synthetic-data models to blurred boundaries in real data, while in most cases the mask follows the general shape of the car. The qualitative evaluation of the model on UA-DETRAC dataset confirmed the generalization abilities of the synthetic-data model.

We conducted extensive experiments on evaluating the influence of various steps in the process of synthetic data generation on the result. For example, we found that using a plain gray background dropped the performance by 0.6%. At the same time, surprisingly, rendering 3D models with broken texture, increased the performance by 0.6% instead of decreasing it. That suggests that broken 3D models decrease the bias introduced with

rendering 3D models, but future work is planned to explore this effect.

Next, we demonstrate the effectiveness of the synthetic data to learn rare cases, where the real data is not available. In our experiments, we consider drastic car occlusion as a rare case. We collected images with highly-occluded cars and tested the models on them. We found that real-data models (which were not trained on such occlusions) confuse car parts in such images, as opposed to the synthetic-data models.

Lastly, we demonstrated the ability of domain adaptation algorithms to reduce the gap between the synthetic and the real data.

9.3 Contributions and future work

The major contributions of the thesis are:

- Designing a system TrafficBlender for producing synthetic and hybrid synthetic-real data for the surveillance traffic camera scenario;
- A dataset of high-quality 3D car models CADillac and the twin datasets of synthetic and real images Asterix & Obelix for foreground segmentation;
- A lightweight toolbox Shuffler for managing annotations in computer vision datasets;
- Comprehensive experiments on the detection and segmentation Machine Learning problems that demonstrated the usefulness of synthetic data, especially in situations when real data is limited.

One direction of research for future work is employing synthetic images for the task of tracking cars between cameras. the problem of matching a car across two images is connected with finding a suitable feature representation which would allow us to separate internal features, such as car style and shape, from external ones, such as orientation or occlusions. Rendering the same 3D car model in two different environments would allow us to learn how to disentangle these features.

Another direction of work is to tackle the high-framerate camera scenario by rendering videos of synthetic scenes as opposed to rendering static images. The gap between a synthetic and real video could be caused not only by the differences between a rendered and a real frame, but also by the synthetic nature of object movements. Exploring this gap and the ways to minimize it would be a step towards a better understanding of how to efficiently use synthetic data.

Bibliography

[1] "Real-Time Traffic Information in New York City." [Online]. Available: http://www.nbcnewyork.com/traffic (Accessed 2019-03-26). 1, 2.3.1, 7, 7.3.1

[2] "As Moscow goes high-tech, so does its surveillance system." [Online]. Available: https://www.washingtonpost.com/world/europe/as-moscow-goes-high-tech-so-does-its-surveillance-system/2017/12/17/3b3ef2cc-da9e-11e7-a241-0848315642d0_story.html (Accessed 2019-03-26). 1

[3] "Traffic Surveillance Cameras Now Installed at Nearly 1,000 Beijing Intersections." [Online]. Available: http://www.thebeijinger.com/blog/2018/05/16/beijing-adds-nearly-1000-traffic-surveillance-camera-citys-intersections (Accessed 2019-03-26). 1

[4] S. F. Smith, G. Barlow, X.-F. Xie, and Z. B. Rubinstein, "SURTRAC: Scalable Urban Traffic Control," 1 2013. [Online]. Available: https://kilthub.cmu.edu/articles/SURTRAC_Scalable_Urban_Traffic_Control/6561035 1

[5] "Video surveillance for traffic." [Online]. Available: https://www.videosurveillance.com/traffic.asp (Accessed 2019-04-06). 1

[6] M. Hejrati and D. Ramanan, "Analyzing 3D objects in cluttered images," in *25th International Conference on Neural Information Processing Systems (NIPS)*. Curran Associates Inc., 2012, pp. 593–601. 1, 7

[7] A. Takeuchi, S. Mita, and D. McAllester, "On-road vehicle tracking using deformable object model and particle filter with integrated likelihoods," in *IEEE IV Intelligent Vehicles Symposium*, June 2010, pp. 1014–1021. 1, 7

[8] R. A. Hadi, G. Sulong, and L. E. George, "Vehicle detection and tracking techniques: A concise review," *Signal and Image Processing: An International Journal (SIPIJ)*, vol. 5(1), pp. 1–12, 2014. 1, 7

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. 1, 2.1.1, 2.1.2

[10] L. Yang, P. Luo, C. C. Loy, and X. Tang, "A large-scale car dataset for fine-grained categorization and verification." in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 3973–3981. 1, 2.3.1, 2.3.4, 6.1, 7

[11] G. J. Brostow, J. Shotton, J. Fauqueur, and R. Cipolla, "Segmentation and recognition using structure from motion point clouds," in *The IEEE European Conference on Computer Vision (ECCV)*, 2008, pp. 44–57. 1, 2.2, 4.1, 6.1

[12] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understand-

ing," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 1, 2.2, 2.4, 3.1, 4.1, 6.1

[13] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *The IEEE European Conference on Computer Vision (ECCV)*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 740–755. 1, 2.2, 3.3, 5, 5.2.3

[14] L. Wen, D. Du, Z. Cai, Z. Lei, M.-C. Chang, H. Qi, J. Lim, M.-H. Yang, and S. Lyu, "Ua-detrac: A new benchmark and protocol for multi-object detection and tracking," arXiv 1511.04136, 2015. 1, 2.3.4, 2.4, 6.1, 8.3

[15] M. Naphade, D. C. Anastasiu, A. Sharma, V. Jagrlamudi, H. Jeon, K. Liu, M. Chang, S. Lyu, and Z. Gao, "The NVIDIA AI City Challenge," in *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Aug 2017, pp. 1–6. 1, 2.3.4, 6.1

[16] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *The International Conference on Neural Information Processing Systems (NIPS-W)*, 2017. 1, 7, 3.7

[17] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://keras.io 1, 7, 3.7

[18] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *The IEEE Conference on Computer Vision and Pattern Recognition*, vol. 1, 2001, pp. I–511–I–518 vol.1. 2.1.1

[19] A. Basharat, A. Gritai, and M. Shah, "Learning object motion patterns for anomaly detection and improved object detection," in *The IEEE Conference on Computer Vision and Pattern Recognition*, June 2008, pp. 1–8. 2.1.1

[20] R. Sa, W. Owens, R. Wiegand, M. Studin, D. Capoferri, K. Barooha, A. Greaux, R. Rattray, A. Hutton, J. Cintineo, and V. Chaudhary, "Intervertebral disc detection in X-ray images using Faster R-CNN," in *2017 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, July 2017, pp. 564–567. 2.1.1

[21] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. Mujica, A. Coates, and A. Y. Ng, "An empirical evaluation of deep learning on highway driving," arXiv 1504.01716, 2015. 2.1.1

[22] Y. Tang, C. Zhang, R. Gu, P. Li, and B. Yang, "Vehicle detection and recognition for intelligent traffic surveillance system," *Multimedia Tools Appl.*, vol. 76, no. 4, pp. 5817–5832, Feb. 2017. [Online]. Available: https://doi.org/10.1007/s11042-015-2520-x 2.1.1

[23] P. F. Felzenszwalb, R. B. Girshick, and D. A. McAllester, "Cascade object detection with deformable part models," in *The IEEE Conference on Computer Vision and Pattern Recognition*, 2010, pp. 2241–2248. [Online]. Available:

https://doi.org/10.1109/CVPR.2010.5539906 2.1.1

[24] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2015. 2.1.1, 7, 7.1, 7.3.2

[25] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes Challenge: A Retrospective," *International Journal of Computer Vision*, vol. 111, no. 1, pp. 98–136, Jan. 2015. 2.1.1, 2.4, 3.1, 3.1, 3.2, 3.3, 6.1

[26] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2999–3007. 2.1.1, 7.1

[27] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes (VOC) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, 2010. 2.1.2, 5.2.5, 5.2.5, 6.2

[28] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, 2015, pp. 3431–3440. [Online]. Available: https://doi.org/10.1109/CVPR.2015.7298965 2.1.2

[29] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs," in *3rd International Conference on Learning Representations, (ICLR), San Diego, CA, USA,*

May 7-9, 2015, Conference Track Proceedings, 2015. 2.1.2

[30] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation," in *IEEE International Conference on Computer Vision (ICCV)*, ser. ICCV '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1520–1528. [Online]. Available: http://dx.doi.org/10.1109/ICCV.2015.178 2.1.2

[31] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, ser. CVPR '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 580–587. [Online]. Available: https://doi.org/10.1109/CVPR.2014.81 2.1.2

[32] F. Yu, V. Koltun, and T. A. Funkhouser, "Dilated residual networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 636–644, 2017. 2.1.2, 8.1

[33] "Miovision." [Online]. Available: https://miovision.com (Accessed 2019-03-26). 2.2, 2.3.1

[34] J. A. Eichel, A. Mishra, N. Miller, N. Jankovic, M. A. Thomas, T. Abbott, D. Swanson, and J. Keller, "Diverse large-scale its dataset created from continuous learning for real-time vehicle detection," arXiv 1510.02055, 2015. 2.2

[35] A. M. López, J. Xu, J. L. Gomez, D. Vázquez, and G. Ros, "From virtual to real world visual perception using domain adaptation - the DPM as example," in *Domain Adaptation in Computer Vision Applications*, ser. Advances in Computer Vision and

Pattern Recognition. Springer, 2017, pp. 243–258. 2.2, 4.1, 6.1

[36] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb, "Learning from simulated and unsupervised images through adversarial training," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. 2.2, 7.3.7

[37] P. Isola, J. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5967–5976, 2017. 2.2

[38] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networkss," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017. 2.2

[39] M.-Y. Liu, T. Breuel, and J. Kautz, Eds., *Unsupervised Image-to-Image Translation Networks*, 2017. 2.2

[40] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf 2.2, 7, 7.3.7

[41] J. Hoffman, E. Tzeng, T. Park, J.-Y. Zhu, P. Isola, K. Saenko, A. Efros, and T. Darrell, "CyCADA: Cycle-consistent adversarial domain adaptation," in *International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and

A. Krause, Eds., vol. 80. Stockholmsmssan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 1989–1998. 2.2

[42] M. Long, H. Zhu, J. Wang, and M. I. Jordan, "Unsupervised domain adaptation with residual transfer networks," in *Advances in Neural Information Processing Systems (NIPS)*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 136–144. 2.2

[43] Y. Taigman, A. Polyak, and L. Wolf, "Unsupervised cross-domain image generation," in *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2017. 2.2

[44] Z. Murez, S. Kolouri, D. J. Kriegman, R. Ramamoorthi, and K. Kim, "Image to image translation for domain adaptation," *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4500–4509, 2018. 2.2

[45] S. Zhao, X. Zhao, G. Ding, and K. Keutzer, "Emotiongan: Unsupervised domain adaptation for learning discrete probability distributions of image emotions," in *ACM International Conference on Multimedia*, ser. MM '18. New York, NY, USA: ACM, 2018, pp. 1319–1327. 2.2

[46] K. Saito, K. Watanabe, Y. Ushiku, and T. Harada, "Maximum classifier discrepancy for unsupervised domain adaptation," in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, 2018, pp. 3723–3732. [Online]. Available: http://openaccess.thecvf.com/content_cvpr_2018/html/Saito_Maximum_Classifier_Discrepancy_CVPR_2018_paper.html 2.2, 8.5

[47] "Real-Time Traffic Information in Pennsylvania." [Online]. Available: http://www.511pa.com/CameraListing.aspx (Accessed 2019-03-26). 2.3.1

[48] "WeatherBug." [Online]. Available: https://www.weatherbug.com/traffic-cam (Accessed 2019-03-26). 2.3.1

[49] "Americas 4,150 traffic cameras, in one map." [Online]. Available: https://www.vox.com/a/red-light-speed-cameras (Accessed 2019-03-26). 2.3.1

[50] S. Zhang, "Deep understanding of urban mobility from CityscapeWebcams," Ph.D. dissertation, Carnegie Mellon University, 2018, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2018-06-19. [Online]. Available: https://search.proquest.com/docview/2043940964?accountid=9902 2.3.1, 2.4, 5, 5.1.1, 5.1, 5.2, 5.3, 6

[51] W. Qiu and A. Yuille, "UnrealCV: Connecting computer vision to unreal engine," in *Proceedings of the European Conference on Computer Vision Workshops (ECCV)*, 2016, pp. 909–916. 2.3.2, 2.3.4

[52] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. Lopez, "The SYNTHIA Dataset: A large collection of synthetic images for semantic segmentation of urban scenes," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 2.3.2, 2.3.4, 4.1, 6.1

[53] A. Gaidon, Q. Wang, Y. Cabon, and E. Vig, "Virtual worlds as proxy for multi-object tracking analysis," in *The IEEE Conference on Computer Vision and Pattern*

Recognition (CVPR), June 2016. 2.3.2, 2.3.4, 4.1, 6.1

[54] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," in *The IEEE European Conference on Computer Vision (ECCV)*, ser. LNCS, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., vol. 9906. Springer International Publishing, 2016, pp. 102–118. 2.3.2, 2.3.4, 4.1, 6.1

[55] A. Shafaei, J. J. Little, and M. Schmidt, "Play and learn: Using video games to train computer vision models," *27th British Machine Vision Conference (BMVC)*, 2016. 2.3.2, 2.3.4

[56] B. C. Russell, A. Torralba, K. P. Murphy, and W. T. Freeman, "Labelme: A database and web-based tool for image annotation," *Int. J. Comput. Vision*, vol. 77, no. 1-3, pp. 157–173, May 2008. [Online]. Available: http://dx.doi.org/10.1007/s11263-007-0090-8 2.4, 3.3, 3.3, 5, 5.2.5, 5.2.5, 6.2

[57] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012. 2.4, 3.1, 3.3, 3.3, 3.4, 3.9, 4.1, 5.2.2, 6.1

[58] H. Xu, Y. Gao, F. Yu, and T. Darrell, "End-to-end learning of driving models from large-scale video datasets," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3530–3538, 2017. 2.4, 3.1, 3.3, 3.3, 4.1, 6.1

[59] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970. [Online]. Available: http://doi.acm.org/10.1145/362384.362685 2.4, 3.2

[60] SQLite Consortium. (2018) SQLite. [Online]. Available: https://www.sqlite.org (Accessed 2018-09-30). 2.4, 3.2

[61] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET '74. New York, NY, USA: ACM, 1974, pp. 249–264. [Online]. Available: http://doi.acm.org/10.1145/800296.811515 3.2

[62] Oracle Corporation. (2018) MySQL 8.0 Reference Manual. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en (Accessed 2018-09-30). 3.2

[63] MariaDB Corporation Ab. (2018) MariaDB Documentation. [Online]. Available: https://mariadb.com/kb/en/library/documentation (Accessed 2018-09-30). 3.2

[64] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009. 3.3, 5, 5.2.5, 6.2

[65] G. Neuhold, T. Ollmann, S. Rota Bulò, and P. Kontschieder, "The mapillary vistas dataset for semantic understanding of street scenes," in *International Conference on Computer Vision (ICCV)*, 2017. [Online]. Available: https://www.mapillary.com/dataset/vistas 3.3, 4.1, 6.1

[66] X. Huang, X. Cheng, Q. Geng, B. Cao, D. Zhou, P. Wang, Y. Lin, and R. Yang, "The apolloscape dataset for autonomous driving," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018. 3.3, 3.3, 4.1

[67] J. Per, V. S. Kenk, R. Mandeljc, M. Kristan, and S. Kovacic, "Dana36: A multi-camera image dataset for object identification in surveillance scenarios," in *2012 IEEE Ninth International Conference on Advanced Video and Signal-Based Surveillance*, Sept 2012, pp. 64–69. 3.3

[68] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. 4, 4.4

[69] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000. 5, 4.4, 5.2.1, 5.2.2

[70] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *1st Annual Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, S. Levine, V. Vanhoucke, and K. Goldberg, Eds., vol. 78. PMLR, 13–15 Nov 2017, pp. 1–16. [Online]. Available: http://proceedings.mlr.press/v78/dosovitskiy17a.html 4.1

[71] Blender Online Community, *Blender - a 3D modelling and rendering package*, Blender Foundation, Blender Institute, Amsterdam, 2018. [Online]. Available: http://www.blender.org 4.2, 4.3, 7, 9.1

[72] G. Varol, J. Romero, X. Martin, N. Mahmood, M. J. Black, I. Laptev, and C. Schmid, "Learning from Synthetic Humans," in *CVPR*, 2017. 4.3

[73] H. Su, C. R. Qi, Y. Li, and L. J. Guibas, "Render for cnn: Viewpoint estimation in images using cnns trained with rendered 3d model views," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015. 4.3

[74] W. Qiu, F. Zhong, Y. Zhang, S. Qiao, Z. Xiao, T. S. Kim, Y. Wang, and A. Yuille, "Unrealcv: Virtual worlds for computer vision," *ACM Multimedia Open Source Software Competition*, 2017. 4.3

[75] M. Johnson-Roberson, C. Barto, R. Mehta, S. N. Sridhar, K. Rosaen, and R. Vasudevan, "Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks?" in *IEEE International Conference on Robotics and Automation*, 2017, pp. 1–8. 4.3, 6.1

[76] W. McKinney, "pandas: a Foundational Python Library for Data Analysis and Statistics," https://www.scribd.com/doc/71048089/pandas-a-Foundational-Python-Library-for-Data-Analysis-and-Statistics, accessed: 2018-09-14. 4.4

[77] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2011 (VOC2011) Results," http://www.pascal-network.org/challenges/VOC/voc2011/workshop/index.html. 5, 6.2

[78] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. New York, NY, USA: Cambridge University Press, 2003. 5.2.1, 5.2.1

[79] M. A. Fischler and R. C. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography," *Commun. ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981. [Online]. Available: http://doi.acm.org/10.1145/358669.358692 5.2.1

[80] Y. Wang, X. Tan, Y. Yang, X. Liu, E. Ding, F. Zhou, and L. S. Davis, "3d pose

estimation for fine-grained object categories," in *The IEEE European Conference on Computer Vision (ECCV) Workshops (1)*, ser. Lecture Notes in Computer Science, vol. 11129. Springer, 2018, pp. 619–632. 5.2.2

[81] R. L.-S. S. M. B. Ricardo Guerrero-Gmez-Olmedo, Beatriz Torre-Jimnez and D. Ooro-Rubio, "Extremely overlapping vehicle counting," in *Iberian Conference on Pattern Recognition and Image Analysis (IbPRIA)*, 2015. 6.1

[82] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014. 7.1

[83] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders, "Selective search for object recognition," *Proceeedings of the International Journal of Computer Vision (IJCV)*, 2013. 7.1

[84] R. B. Girshick, "Fast R-CNN," in *IEEE International Conference on Computer Vision (ICCV)*, 2015. 7.1

[85] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 9, pp. 1904–1916, 2015. 7.1

[86] C. L. Zitnick and P. Dollár, "Edge boxes: Locating object proposals from edges," in *European Conference on Computer Vision (ECCV)*, September 2014. 7.1

[87] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified,

real-time object detection," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 7.1

[88] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *The IEEE European Conference on Computer Vision*, 2016, to appear. [Online]. Available: http://arxiv.org/abs/1512.02325 7.1

[89] H. Hattori, V. N. Boddeti, K. M. Kitani, and T. Kanade, "Learning scene-specific pedestrian detectors without real data." in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. 7.1

[90] P. KaewTraKulPong and R. Bowden, "An improved adaptive background mixture model for real-time tracking with shadow detection," in *2nd European Workshop on Advanced Video Based Surveillance Systems*, vol. 5308, 2001, pp. 135–144. 7.2.1

[91] C. Stauffer and W. Grimson, "Adaptive background mixture models for real-time tracking," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1999. 7.2.1

[92] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015. 7.3.2

[93] "Moscow traffic police admitted a fine issued by mistake," 2016. [Online]. Available: https://www.kommersant.ru/doc/3075725 (Accessed 2019-03-26). 8.1

[94] S. L. Smith, P. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate,

increase the batch size," in *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2018. 8.1