EBERHARD-KARLS-UNIVERSITÄT TÜBINGEN
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl Kognitive Systeme

Bachelorarbeit

Sampling Useful Synthetic Data for UAV Object Detection

David Ott

Betreuer: Prof. Dr. rer. nat. Andreas Zell
Wilhelm-Schickard-Institut für Informatik

Benjamin Kiefer
Wilhelm-Schickard-Institut für Informatik

Begonnen am: 1. Juli 2020

Beendet am: 28. Oktober 2020

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Tübingen am 28. Oktober 2020

_____

David Ott

Abstract. Current object detection models require large amounts of labeled data to be trained. In this work, modifications, adaptations, and improvements to the DeepGTAV framework were made. This simulation environment based on the GTAV computer game, allows the capturing of synthetic object detection training data from a UAV's perspective. Two large scale synthetic datasets are produced and then analyzed with readily available meta data from DeepGTAV.

A state-of-the-art, YOLO based object detection model is trained on real-world and synthetic data. Two training schemes with synthetic data are employed. In training purely with synthetic data reasonable real-world performance is achieved. By pretraining with synthetic data and then training with real-world data the real-world performance is improved for small real-world datasets. Furthermore, improvements in the data generation processes between the two synthetic datasets are shown. The effects of different dataset sizes, model sizes and of the usage of pretrained weights is investigated. Finally, motivated, novel data generation architectures are proposed.

Kurzfassung. Moderne Objekterkennungsmodelle erfordern das Training mit großen Mengen annotierter Trainingsdaten. In dieser Arbeit werden Änderungen, Anpassungen und Verbesserungen am DeepGTAV-Framework vorgenommen. Diese Simulationsumgebung basierend auf dem Computerspiel GTAV erlaubt die Generierung von synthetischen Trainingsdaten für die Objekterkennung in Luftaufnahmen. Zwei große synthetische Datensätze werden erstellt und anhand ihrer leicht verfügbaren Metadaten analysiert.

Ein aktuelles, auf YOLO basierendes Objekterkennungsmodell wird mit synthetischen Daten und Daten aus der realen Welt trainiert. Zwei Trainingskonfigurationen werden verwendet: In einem Training ausschließlich mit synthetischen Daten wird eine akzeptable Genauigkeit in der echten Welt erreicht. Durch Vortraining mit synthetischen Daten und anschließendes Training mit Daten aus der realen Welt wird die Leistung in der realen Welt für kleine Datensätze verbessert. Darüber hinaus werden Verbesserungen des Datengenerierungsprozesses zwischen den beiden synthetischen Datensätzen gezeigt. Die Auswirkung verschiedener Datensatzgrößen, Modellgrößen und der Verwendung von vortrainierten Gewichten wird untersucht. Abschließend werden neue, motivierte Datengenerierungsprozesse vorgeschlagen.

Contents

1. Introduction

In recent years, great improvements have been made to the state-of-the-art in object detection mainly due to the availability of large-scale datasets, compute power and deep learning techniques [ZZXW19]. This work is concerned with achieving good real-world performance of an object detector for UAV camera perspectives by supplying synthetic training data.

In the following, the challenges of object detection from aerial view will be discussed. Then, a review of the state-of-the-art in synthetic training data generation and use will be given, followed by a motivation for the different training and testing conditions employed in this work.

1.1. UAV Object Detection

Object detection from UAV perspective poses some specific problems. First of all, the relative position of the camera to the object is very different from that in datasets classically used for general object detection training (e.g. ImageNet [DDS+09], Pascal-VOC [EVGW+10] or COCO [LMB+14]). Objects are mostly seen from above, as opposed to being seen from the side, which results in vastly different object silhouettes.

Furthermore, the distance between the camera and the objects for UAV images is larger on average. Therefore, the objects in the image are smaller, which poses difficulties to object detection systems.

An additional challenge is posed if real time object detection on the UAV is required. In this case, fast and hardware inexpensive object detection systems are required, to allow real time performance when running on integrated hardware aboard the UAV. For this reason, faster single-stage object detectors like Yolo [RDGF16, RF17, RF18, BWL20] or Faster RCNN [RHGS15] are preferred for this task, as compared to multi-stage object detectors like Cascade R-CNN [CV18].

For this work different real-world datasets for UAV object detection have been considered, namely VisDrone [ZWB+18, ZWD+20], UAVDT [DQY+18], VIRAT [OHP+11], AU-AIR [BK20] and the Stanford campus dataset [RSAS16]. [BK20] also present a comprehensive overview of different datasets for UAV object detection.

1.2. Synthetic Training Data

One of the most resource demanding tasks in building an object detection system is the acquisition of training data. Traditionally, a large sample of images with the required objects is taken which then must be annotated by human annotators through tedious manual work. An alternative to this is the generation of synthetic, simulated training data. The advantage here is that the ground truth labeling data is easily available, as the exact positions of the simulated objects are known to the simulation, and an additional labeling process is not necessary anymore.

One branch of generating such synthetic training data concerns itself with building simulators that produce the required training data for the specific task. One example of this is CARLA

[DRC+17], a simulation environment that is meant to generate training data for an urban driving scenario. Another example is AirSim [SDLK18], an environment that is meant to generate such data for UAVs. A more elaborate, general approach is seen in Meta-Sim [KPL+19]. This system also tries to mimic the underlying distribution of objects from the real world by trying to learn its probabilistic scene grammar.

A different approach is not building the synthetic environment from scratch but using already existing high-fidelity virtual environments in modern computer games. Such approaches can be seen in [RVRK16], [RHK17] and [AEK+18]. Furthermore [JRBM+16], [MSF+17] and [HCW19] have shown that such synthetic training data can in fact significantly increase the performance of real-world object detectors for a car driving environment. In [JRBM+16] it is shown that a large sample of purely synthetic training data (200,000 images with bounding boxes), generated from the computer game GTAV can outperform training with real-world data. In [YWS+18] it is shown that by training an object detector with synthetic LiDAR data from GTAV and real-world data, a better performance of a LiDAR object detector is achieved than by only training with real-world data. Similarly, in [HCW19] it is shown that pretraining an object detector with synthetic LiDAR data generated in GTAV and then training it with real-world data improves the performance of a LiDAR object detector, compared to training only with the real-world data. In general, it seems to be the case that relatively larger sets of synthetic data have to be used, compared to the use of real-world data. In [JRBM+16] it is stated that one possible reason for this is that *"It appears the variation and training value of a single simulation image is lower than that of a single real image. The lighting, color, and texture variation in the real world is greater than that of our simulation and as such many more simulation images are required to achieve reasonable performance."*.

There are some advantages to not building a simulation environment from scratch but modifying current computer games. First, this often proves to be the simpler task, as both the technical environment (e.g. the engine) and the actual simulated environment (e.g. the game world, the models) are already built. Furthermore, computer games often strive for high visual fidelity and realism as a selling point. Many custom-built simulators do not achieve this level of visual realism. Another advantage of computer games is that they are built to simulate a dynamic environment. Entities in this world move around in realistic ways, for example cars in GTAV drive around and abide to traffic laws. This opens the possibility of producing more than single frame training data. For example, video data for object tracking tasks would also be possible. For the detection of pedestrians there is another advantage to those dynamic environments. The different models of people in the game world are rigged and animated to move around in a realistic manner. This produces people in many different poses, which would not be seen in a static environment. It stands to reason that the variability of the pose is a special problem for biological entities, in particular people. While for the purpose of object detection a car can be seen as a rigid body in the world, only changing its position but not its actual form, this is not the case for people, which change both.

There are also some disadvantages to modifying computer games as simulation environments as compared to using simulation environments built from scratch. First, computer games are often not as modifiable as the custom built simulation environments. In many cases their source code is closed source, making modifications difficult. But even if this is not the case, custom built simulators are built to be easy to modify by design, while the only goal in computer game design is to produce the final product. There often still is an adherence to good design principles in computer games, as this reduces the iteration times and speed of development, but this is not a given. Another disadvantage of computer games as simulation environments is that there are often shortcuts taken in the simulation of computer games to achieve real time performance.

One example of such a shortcut is the use of precomputed lighting in baking [SSDS12, Ver17]. Of course, those shortcuts remove some of the visual fidelity that would be present in an actual physical simulation. Such shortcuts are possibly undesirable when wanting to produce training data that matches real-world data as best as possible to achieve the best real-world performance. When computation time is available and real time performance is not required, as is the case for training data generation, then one would have reason to opt for a simulation environment that actually simulates these things as true to reality as possible.

Last but not least, there is a potential problem of the licensing with computer games. While most simulators built for this purpose are distributed with open source and very permissive licenses, computer games are often distributed with restrictive licenses. It must be noted that the question of legality of object detectors trained on proprietary data, where only the object detector is published but the actual data is not, is very relevant in this context but far beyond the scope of this work.

In the case of GTAV the publisher allows non-commercial use of the game footage [RVRK16].

1.3. Goals of this Work

Motivated by those previous works, the goal of this work is to examine the use and usefulness of synthetic data to train a real-world object detector for UAV imaging. Therefore, the goals of this work are as follows: First, the DeepGTAV-PreSIL framework [HCW19] is to be modified in such a way to allow generating training data from a UAV's perspective. Then a state-of-the-art object detector is to be trained with this data. Modifications and improvements of the data generation process are to be made. In different experimental conditions the effect of those improvements is to be shown. In a set of experiments, answers to the following questions shall be found:

Can good performance for the object detection task be achieved by only training with synthetic data?

In an experimental setup similar to [JRBM$^+$16] it shall be tested if one can achieve good performance on a real-world object detection task by only training with synthetic data.

Does synthetic data improve the performance of an object detector by using it to pretrain?

In an experimental setup similar to [HCW19] an object detector shall first be trained only using synthetic data. Then the object detector is further trained on the real-world dataset. If similar results as in [HCW19] are expected, this should improve the performance of the object detector.

Furthermore, the effect of the size of the real-world dataset shall be tested. It stands to reason, that the synthetic training data can mitigate the effects of having a real-world dataset that is too small. Therefore, the size of the real-world dataset shall be varied, the hypothesis being that the synthetic data pretraining has a disproportionately larger effect for smaller real-world datasets.

What is the effect of using weights pretrained on real-world images in conjunction with synthetic data?

It became a standard training procedure in research and industry to not train object detectors from scratch, but to use pretrained weights that have been trained on large scale object detec-

tion datasets such as ImageNet [DDS+09], Pascal-VOC [EVGW+10] or COCO [LMB+14] as initial weights. In general, this results in faster training convergence and better final performance [GDDM14, ZF14]. It is still an open question in research, what exactly is learned in such a pretraining. It stands to reason, that one of the things that is learned is low level feature extraction (e.g. extraction of lines in different angles and basic geometric shapes), similar to the low levels of visual processing in the human visual system [BCDH10]. [SK19] state that pretraining *"is very effective since many image datasets share low-level spatial characteristics that are better learned with big data."* Another possible hypothesis is that some abstract representation of how the physical world works is attained (e.g. how light scatters, how shadows work, etc.). In a similar sense, some understanding about the "messiness" of the real world could be attained providing more resilience for noise in the data. In contrast to the real world, some of those basic properties are not present in virtual environments. In particular, virtual worlds are not fully physically accurate, have less details and are less noisy. For example, virtual environments in general are made of a limited amount of vertices and polygons, resulting in less details and more flat surfaces than in the real world. Therefore, it stands to reason that the world model encoded in pretrained weights trained on real-world data allows a model that is afterwards only trained on synthetic data to also contain these properties, that it otherwise could not learn. A model with initial weights pretrained on real-world data should therefore perform disproportionately better on real-world data than a model that is trained only on synthetic data. To test this hypothesis, the improvement by using pretrained weights on a model trained only with synthetic data shall be compared to the improvement of pretrained weights on a model trained with real-world data. The hypothesis being that the use of pretrained weights results in a larger improvement for the model only trained with synthetic data than for the model trained with real-world data.

Can the object detection performance be improved with a better data capturing process?

Several changes were made to the initial data capturing system. Those were mostly inspired by the attempt to produce a well-balanced dataset representative of the real-world distribution. In short, those changes were: changing the camera heights above the ground from 10-100m to 0-20m, varying the camera angles from $-20°$ to $-90°$ instead of only looking directly downward ($-90°$), balancing the prevalence of the different object classes and increasing the amount of total objects. A detailed description of these changes can be found in section 2.1.2. The effect of these changes on the performance shall be tested.

How does the size of the synthetic dataset relate to performance?

In [JRBM+16] the hypothesis that the information content of a synthetic image is lower than that of a real-world image is stated. Furthermore, it holds in general that more training data results in better performance and [JRBM+16] claim that they have not yet reached a plateau, even with 200,000 training samples. To examine the relationship between the amount of synthetic training data used and the achieved performance, training shall be conducted with different amounts of synthetic data.

Do the performance gains also apply to larger models?

To see if the improvements by using synthetic data hold for larger models the performance of such a larger model trained with synthetic data shall be examined.

1.3.1. Efficient Sampling

One of the main goals of this work was to leverage the possibility of on the fly ground truth data generation combined with accessible meta data to improve the data generation process. Works from the literature have shown, that in a sense the loss is a sort of metric for the information content of a training sample, meaning that there can be learned more from a training sample with a higher loss. For example, in importance sampling [KF18] it can be seen that training converges faster by presenting those samples more often which produce a higher training loss.

For this work in particular, the hypothesis was, that for a specific model, the loss would not be randomly distributed in the environment, but rather clustered, such that regions with higher loss and regions with lower loss would exist. This could be, for example, due to some environment factors (e.g. the model is good at detecting objects on grass, but not on sand), the objects in the environment (e.g. the model is bad at detecting pedestrians, resulting in a high loss around pedestrians), the position in the environment (e.g. the model is bad at detecting the testing samples from a greater height) or the camera angle (e.g. the model is good at detecting objects from a bird's eye view, but not in a 45° angle). In principle, this could be extended to any meta property of one training sample.

This could then be leveraged to produce more training data in such regions where the loss is higher. This would in turn result in better balanced training data. This could also be used for an iterative process: the model is trained on some data, then some data is generated in such regions with high loss, then the model is trained on this data, etc. Such an iterative process would in principle enable endless training to fine tune the model further and further, until the model capacity is reached. A rudimentary description of such a training schedule is given in Alg. 1. Such a training schedule would also be closely related to active learning [Set09].

Algorithm 1 Rudimentary pseudo code of an iterative data generation and training procedure

Require:
model: an initial object detection model, possibly initialized with random weights
remote location: the location on which actual model training is conducted, e.g. a compute cluster

1: **while** true **do**
2: pull the current version of the **model** from the **remote location**
3: generate **data** based on some informed metric of the information content for the current **model**
4: upload **data** to **remote location**
5: train **model** on **remote location**
6: **end while**

The applicability of such a training schedule is heavily dependent on the assumption, that the per sample loss for a specific model is not randomly distributed in the environment, but in some form clustered. Therefore, this hypothesis needed to be tested. To this end, two hypotheses were tested. The first hypothesis is that a model trained primarily on training data recorded from greater heights would perform significantly better on testing data recorded at a greater height than a model trained primarily on data recorded on lower heights and vice-versa. The second hypothesis is that a model trained on data recorded from different camera angles would perform significantly better on testing data recorded at flat camera angles than a model trained only with data recorded directly looking downwards. The detailed procedure for these tests is described in section 2.5.

Although these hypotheses seem to be very intuitive and reasonable, they are the base requirement of such an iterative training procedure. Furthermore, their verification further confirms some theoretical understanding of object detectors.

The initial plan of this work was to then implement an iterative training schedule as described above. Due to technical limitations of the DeepGTAV framework in its current state this was not possible. Nonetheless, in section 4.3 different possible specific implementations of such an iterative training schedule are theoretically discussed.

1.4. Main Contributions

The main contributions of this work are:

1. Modifying the DeepGTAV framework to allow capturing data from UAV perspective.
2. Providing a large synthetic dataset for aerial view object detection.
3. Empirically showing that training an object detection model only with such synthetic data already achieves reasonable performance for real-world object detection tasks.
4. Empirically showing that such synthetic data is useful to improve the performance of a real-world object detector.
5. Proposing novel iterative training procedures.

2. Method

2.1. The DeepGTAV Framework

This work heavily builds upon the DeepGTAV-PreSIL framework[1] [HCW19] which itself is built upon the DeepGTAV framework[2]. Originally, DeepGTAV was meant as a reinforcement learning environment for self driving cars, providing functionality to interact with GTAV through a TCP-server. The DeepGTAV framework builds upon the functionality of SkriptHookV[3] to run inside GTAV.

DeepGTAV-PreSIL integrates the functionality of DeepGTAV with the functionality of GTAVisionExport[4], the technique presented in [JRBM+16], to extract depth and stencil information.

In the following, the combination of those functionalities and further additions will be referred to as DeepGTAV or the DeepGTAV framework. In this section the structure and functionalities of DeepGTAV will be explained in detail. Then, the modifications and improvements that have been made to enable this work will be shown.

2.1.1. Structure of DeepGTAV

DeepGTAV opens a TCP-server at port 8000 to send and receive messages, encoded as JSON. Through those messages a client can remotely control GTAV and receive the data, which was extracted in GTAV, for example with a Python [VRD09] script. To do this, the package **VPilot**[5] is used, providing simple interfaces, e.g., for the communication with DeepGTAV. In the following the functionalities of the most important files are explained:

Scenario.cpp

In this file a **Scenario** class is defined, that manages all the interactions with GTAV through functions from ScriptHookV defined in **natives.h**. Furthermore the **Scenario** calls the data extraction functions in **ScreenCapturer.cpp** and **ObjectDetection.cpp**.

Server.cpp

Server.cpp defines a **Server** class that starts a TCP connection on port 8000. The function **Server::checkRecvMessage()** parses received messages as JSON, decodes the commands and calls the appropriate functions of the **Scenario**. The function **Server::checkSendMessage()** checks if a new message has been generated and in that case sends the new message over the TCP connection, such that the client can receive the message. The JSON parsing is done with the rapdidjson library[6].

[1] https://github.com/bradenhurl/DeepGTAV-PreSIL
[2] https://github.com/aitorzip/DeepGTAV
[3] http://www.dev-c.com/gtav/scripthookv/
[4] https://github.com/umautobots/GTAVisionExport
[5] https://github.com/aitorzip/VPilot
[6] https://github.com/Tencent/rapidjson

A full command flow for sending a command to DeepGTAV as a message from the client in a Python script works as follows: In the data generation script, the client is called with a message command (Alg. 2). In this example the message command is **GoToLocation()** with a target x-, y-, and z-coordinate.

Algorithm 2 Exerpt from drive.py

```
[...]
client.sendMessage(GoToLocation(x, y, z))
[...]
```

This message is encoded as a JSON string according to its class definition seen in Alg. 3, which is sent over the TCP-connection by **client.sendMessage()**.

Algorithm 3 Exerpt from VPilot.deepgtav.messages.py

```
class GoToLocation:
    def __init__(self, x, y, z, speed = 20.0):
        self.x = x
        self.y = y
        self.z = z
        self.speed = speed

    def to_json(self):
        return json.dumps({'GoToLocation':self.__dict__})
```

The message is received by the DeepGTAV **Server** and decoded into a JSON document **d** (Alg. 4). This JSON document is then checked to contain specific commands and, in that case, the appropriate function from **Scenario** is called.

In this case the function **Scenario::goToLocation()** is called which is defined in Alg. 5 This function uses the wrapper function **AI::TASK_VEHICLE_DRIVE_TO_COORD()** defined in **natives.h** to interact with GTAV.

The messages sent from DeepGTAV to the client are decoded as python objects containing fields, for example **message[“frame”]** contains the captured image and **message[“time”]** contains the current in-game time.

ScreenCapturer.cpp

The ScreenCapturer.cpp captures the frame of GTAV that is currently on the screen.

ObjectDetection.cpp

In ObjectDetection.cpp all the functionalities for capturing the object detection data are implemented with the **ObjectDetection** class. The most notable functions are **ObjectDetection::setDepthBuffer()** and **ObjectDetection::setStencilBuffer()** which are used to set the depth and stencil buffers.

The function **ObjectDetection::generateMessage()** is used to export the meta data information.

The function **ObjectDetection::exportDetectionString()** produces a string which contains all the bounding box information in the format shown in Alg. 7.

Algorithm 4 Exerpt from Server.cpp

```
void Server::checkRecvMessage() {
        // Here the message is received as the string json.
        // The specific implementation is omitted for readability
        [...]
        d.Parse(json);
        if (d.HasMember("commands")) {
                [...]
        }
        [...]
        else if (d.HasMember("GoToLocation")) {
        const Value& target = d["GoToLocation"];
        scenario.goToLocation(target["x"].GetFloat(),
                              target["y"].GetFloat(),
                              target["z"].GetFloat(),
                              target["speed"].GetFloat());
        }
        [...]
}
```

Algorithm 5 Exerpt from Scenario.cpp

```
void Scenario::goToLocation(float x, float y, float z, float speed) {
        Hash vehicleHash;
        vehicleHash = GAMEPLAY::GET_HASH_KEY((char*)_vehicle);

        AI::CLEAR_PED_TASKS(ped);
        AI::TASK_VEHICLE_DRIVE_TO_COORD(ped, m_ownVehicle,
                x, y, z, speed, Any(1.f), vehicleHash, _drivingMode,
                2.f, true);
}
```

Constants.h

In **Constants.h** different settings for DeepGTAV are defined.

natives.h

natives.h provides human readable namespaces and function wrappers for functions defined by ScriptHookV found at different memory addresses.

2.1.2. Improvements

It must be noted, that some of the functionality that existed in DeepGTAV was obscured or removed through DeepGTAV-PreSIL and some of the functionality that existed in DeepGTAV-PreSIL has been obscured, removed or modified by the modifications that were made for this work. In general, the code quality at this point is not very good.

The general aim was to make the messages and commands used to communicate with DeepGTAV more understandable, by having more smaller command messages instead of setting all the settings with a *Dataset* and *Scenario* message, which was the case for DeepGTAV-PreSIL. This also allows to change the settings at runtime.

The added commands are shown below with example usages at the end of this section. Furthermore, different other improvements that have been made to the DeepGTAV framework are presented.

Faster data generation

Preliminary tests have shown that the data generation speed was significantly affected by having the export directory (the directory where DeepGTAV would save its recorded data) on either an HDD or an SSD drive. This allowed the conclusion that one of the bottlenecks for the generation was the writing process to the file system.

The option **DONT_COLLECT_IMAGE_AND_BBOXES_TO_FILE** allows to not write the exported data directly to the file system, so it is only sent via the TCP connection. This increases the data generation speed by about 15%. This can be attributed to not pausing the DeepGTAV-scripts in GTAV for the read-write actions, but using the buffered higher level read-write functionality in Python. Furthermore the online data generation allows direct modification and handling of the exported data in Python, without modifying the code of DeepGTAV.

The data generation speed was further improved by only recording the necessary data. To this end the option **ONLY_COLLECT_IMAGE_AND_BBOXES** was added to **Constants.h**, which deactivates all functionality that is not relevant for the capturing of the bounding box information. This also increases the data generation speed by about 15%.

StartRecording / StopRecording

These messages start and stop the recording in DeepGTAV. As the recording process takes approximately six seconds per frame, with recording active constantly, the agent traverses the world very slowly. These functions allow to activate and deactivate the capturing process at will, thereby allowing to traverse the world faster between generating frames. In total this allows to cover more areas of the world for the dataset generation, thereby allowing to generate more balanced training data. An example for the use of StartRecording / StopRecording messages to only export data for two frames at every 10th frame that would be recorded is

shown in Alg. 9. Due to the workings of DeepGTAV, data is exported for the frame where the **StartRecording** message is received, and data is exported for the frame where the **StopRecording** message is received. Therefore, at minimum, two frames must be exported at a time.

SetCameraPositionAndRotation

This command allows to set up the camera position and rotation relative to the player vehicle. This is required to record images from UAV perspective and allows to change the camera perspective at will, to generate more diverse training data.

SetClockTime

This command allows to set the in-game clock time. This enables the capturing of training data at different times of the day, where different shaders are active. Examples can be seen in figure 3.2.

SetWeather

This command allows to set the in-game weather. The different weathers come with different camera effects (e.g. raindrops or snowflakes) and different shaders.

This command has not been used in the data generation process, because VisDrone, the dataset used for comparison in this work, does not have other weathers apart of a clear weather without rain. Additionally, some of the weather conditions produce unwanted interactions with the bounding boxes, e.g. for cloudy weather some objects are occluded by the mist, but the bounding boxes are still recorded. In some preliminary experiments different weather settings produced bad results.

TeleportToLocation

This command teleports the player vehicle to the specified location. This allows to quickly get to a starting position for the data capturing process and to randomize the starting position. This command can not be used multiple times, as the stencil and depth buffers do not get reinitialized properly.

GoToLocation

This command uses the GTAV AI to travel to the specified location.

Algorithm 6 Examples for the usage of the different commands described above.

```
client.sendMessage(SetCameraPositionAndRotation(x_offset, y_offset,
                                    z_offset, rot_x, rot_y, rot_z))
client.sendMessage(SetClockTime(hours, minutes, seconds))
client.sendMessage(SetWeather(weather))
client.sendMessage(TeleportToLocation(x, y, z))
client.sendMessage(GoToLocation(x, y, z))
```

Travel height

When given a way finding task for flying vehicles, the GTAV AI will try to get to a certain travel height and then travel on this height to the target location, only reaching the final specified height when close to the target. For this reason, it is challenging to capture data at a specific height, as the AI always defaults to its predefined travel height. To mitigate this the agent was commanded to go to a closer location in the current direction with the correct height, if its current height above the ground did not match the wanted travel height.

Matching DeepGTAV objects to VisDrone objects

DeepGTAV generates the object bounding boxes in a format as seen below (Alg. 7)

Algorithm 7 Example of a bounding box string as it is extracted by DeepGTAV. The most important fields are marked.

```
Car 0 0 -1.62759 1003 553 1056 584 1.27776 2.08377 4.50658
    2.64644 1.67631 36.7839 -1.55577 3842 1413 134 8.4032 0.0205022
    0.0161763 ninef 0
[...]
Pedestrian 0 0.354545 -1.78954 1077 541 1082 560 1.82406
    1.2 1.44 9.98207 1.71453 79.2913 -1.66431 14082 71 5 1.66095
    -0.0104276 0.000563514 Pedestrian 0
```

The most relevant fields are the first, which refers to the class the object has in GTAV. The fifth, sixth, seventh and eighth field refer to to the bounding box (left, top, right, bottom pixel coordinates) and the second to last value which refers to the in-game model name (marked bold in Alg. 7). The other fields refer to truncation and occlusion information, 3D position of the object in the game world, etc. and are discarded.

The first class field does not match to the VisDrone conventions correctly. Therefore, the correct labels have to be inferred from the object model name (second to last value). This is done when parsing the bounding box string in the client Python script. Extensive functionality to do this has been implemented in **VPilot.utils.BoundingBoxes.py**. There is further functionality implemented that directly converts the bounding boxes to the correct format required by the model employed in this work.

More objects and better class distributions

After initially capturing and analyzing the DeepGTAV_HIGH dataset (described below in section 2.3.2), it became apparent, that there is a rather small amount of objects in the in-game world and there were large class imbalances.

To mitigate those issues, some modifications to the game files have been made: The total amount of traffic vehicles and pedestrians was increased using the mod **Simple Increase Traffic (and Pedestrian)**[7] which was installed using **OpenIV**[8]. To prevent crashes, the mod **Heap Limit Adjuster**[9] was added, which would allow a larger total amount of in-game objects.

[7] https://de.gta5-mods.com/misc/simple-increase-traffic-and-pedestrian
[8] https://openiv.com/
[9] https://de.gta5-mods.com/tools/heap-limit-adjuster-600-mb-of-heap

To balance the class distribution, the files

GTAV\mods\update\update.rpf\x64\levels\gta5\popgroups.ymt

and

GTAV\mods\update\update.rpf\common\data\levels\gta5\popcycle.dat

were edited using **OpenIV**. The file **popgroups.ymt** defines population groups by the contained model names in an XML format. This file was edited in such a way that there existed population groups for each of the classes (pedestrian, car, van, truck, bus, motor). An example of the population group for the class bus can be seen in Alg. 8.

Algorithm 8 Excerpt from **popgroups.ymt** for a population group for the *bus* class that contains the in-game models *bus, airbus* and *coach*.

```
<Item>
  <Name>VEH_DEEPGTAV_BUS</Name>
  <models>
    <Item>
      <Name>bus</Name>
      <Variations type="NULL"/>
    </Item>
    <Item>
      <Name>airbus</Name>
      <Variations type="NULL"/>
    </Item>
    <Item>
      <Name>coach</Name>
      <Variations type="NULL"/>
    </Item>
  </models>
  <flags>POPGROUP_SCENARIO POPGROUP_AMBIENT</flags>
</Item>
```

The file **popcycle.dat** defines the specific amounts of objects that are spawned at a location at different times of day and the relative amounts of the different population groups that are used. The file was modified in such a way, that at all locations at all times of day the same population groups were spawned. Those population groups were defined in popgroups.ymt as described above. Specifically, those population groups were DeepGTAV_peds, which contained all pedestrian models and VEH_DEEPGTAV_CAR, VEH_DEEPGTAV_VAN, VEH_DEEPGTAV_TRUCK, VEH_DEEPGTAV_BUS and VEH_DEEPGTAV_MOTOR. With each of those vehicle classes making up 20% of the total population, the classes should occur equally often in the game world.

Extract meta data

DeepGTAV was modified in such a way that it would send meta data in its message to the client. Those are the location of the player vehicle, the height above the ground of the player vehicle, the in-game time, the camera location and the camera rotation.

2.2. Data Generation

The data generation was conducted on a personal computer with Windows 10 having an AMD Ryzen 5 2600X, 16GB of RAM and a Nvidia GeForce GTX 1080 with 8GB of GDDR5 graphics memory. All Data Generation was conducted with GTAV version 2060.1

2.2.1. Data Cleaning

DeepGTAV exports very small bounding boxes, which can not reasonably be expected to be recognized. For example, this is the case for objects that are very far in the distance or for objects that are almost completely occluded. In preliminary tests, including those bounding boxes for the training resulted in bad performances. Therefore, all bounding boxes which had a width or height smaller than 10 pixels were removed. An example image with the removed bounding boxes can be seen in Fig. 3.5.

Images where no bounding boxes remained after cleaning were fully removed from the dataset.

2.3. Datasets

In this work three datasets were used. The VisDrone dataset was used as a benchmark to assess the real-world performance. Two datasets were generated in DeepGTAV which in the following will be referred to as DeepGTAV_HIGH and DeepGTAV_LOW. Descriptions of those datasets are given below:

2.3.1. VisDrone

As a benchmark for the real-world performance of the trained model, the VisDrone2019-DET [ZWB+18, ZWD+20] dataset has been chosen. This is the subsection for a detection task of the VisDrone2019 dataset. The dataset consists of 6,471 training images, 548 validation images and 3,190 testing images, totaling 10,209 images. The test set is split into two sets, test-dev with 1,610 images and test-challenge with 1,580 images. Ground truth bounding box data has not been published for the test-challenge set. For this reason, only the test-dev set was used in this work, which will be referred to as the VisDrone test set in the context of this work. Images are labeled with bounding boxes for 10 object categories: *pedestrian, people, car, van, bus, truck, motor, bicycle, awning-tricycle, tricycle*. Furthermore, *ignored regions* bounding boxes, regions where no ground truth labeling data was generated because there is too much occlusion or there are too many objects, and *other* object bounding boxes, for special objects like forklifts or tankers, are included.

The *people* class contains people that are not walking or directly taking part in traffic, e.g. people that are sitting or people that are in vehicles. The *motor* class contains all sorts of two-wheeled motorized vehicles, e.g. motorcycles or motor scooters.

The state-of-the-art performance of different object detectors on the VisDrone dataset is reported in [Pai19].

In this work, the *ignored regions* were grayed out and the bounding boxes for *other* were removed. To allow compatibility with the classes extracted in DeepGTAV, some classes were removed or integrated into other classes: The classes *bicycle, awning-tricycle, tricycle* and *other* were removed. The class *people* was integrated into the class *pedestrian*.

Therefore, the final classes on which the object detectors were trained are: *pedestrian, car, van, truck, bus* and *motor*.

2.3.2. DeepGTAV_HIGH

DeepGTAV_HIGH is the first large scale synthetic dataset generated for this work, using the modified DeepGTAV framework. Data generation was conducted in the in-game area with $x \in [-1960, 1900]$ and $y \in [-3360, 2000]$ (see Fig. 2.1), starting from a random location in the area. A random target location in this area was created, to which the in-game way finding AI traveled in a direct path. Upon arrival a new random target location was generated. A constant travel height above the ground was enforced. This travel height was uniformly chosen in between 10m to 100m in-game height. A new random travel height was chosen every 100 captured frames. The camera was set to a pitch angle of -90° relative to the vehicle, directly pointing downwards and with an offset of 3m below the vehicle, resulting in an actual camera height in between 7m to 97m. Due to limitations of the in-game path finding AI the vehicle would in general not travel at the low travel heights. Furthermore, due to the movement of the vehicle, the height and camera rotation have a little bit of additional movement. The true distributions can be seen in section 3.1. The in-game time was set to a random value at the beginning of the data generation and then ran according to the in-game time cycle. Only two frames were captured, for every 10th frame being generated.

Images containing no object bounding boxes were discarded in the data generation. There were approximately 55,000 frames generated, taking around six days. Those images were then cleaned according to section 2.2.1. From the remaining images, 50,000 images were used as the DeepGTAV_HIGH dataset. The first 40,000 images were used as a training set, the remaining 10,000 images as a validation set.

The changes to the game files described in section 2.1.2 have not yet been implemented for this data generation. Pseudocode of the data generation script can be seen in Alg. 9.

2.3.3. DeepGTAV_LOW

In principle, the same data generation process as for DeepGTAV_HIGH (section 2.3.2) was used. The in-game area was reduced to $x \in [-1200, 1400]$ and $y \in [-2200, 1300]$, as the larger capturing area in DeepGTAV_HIGH resulted in many frames to be discarded due to not containing any objects, because the camera was over water or fields (shown in Fig. 2.1). The travel height was randomly generated in between 20m and 40m with a new travel height being generated every 100 captured frames. The camera was offset 20m below the vehicle, resulting in actual camera heights between 0m and 20m. This setup was a workaround to allow low height capturing while still using the in-game path finding AI, which would not travel at low heights. The camera pitch relative to the vehicle was randomly set every 140 captured frames in between -90° and -20°. Again, data generation started at a random in-game time, with the in-game time cycle running. The true height and camera angles were again subject to random movement. The true distributions are shown in section 3.1. Again, for every 10th generated frame two frames were captured. In general, the same data generation script as shown in Alg. 9 was used.

For this data generation some modifications to the game files were made to spawn more in-game objects and to balance the distribution between the different object classes. Those are described in section 2.1.2.

Algorithm 9 Pseudo code of the data generation script.

Require:
AREA: an in-game area for the agent to travel in
h: a range of travel heights
c: a range of camera angles
savedir a directory to save the exported data to, in this work a remote location was used

```
start new TCP-connection with DeepGTAV as client
teleport to random position in AREA
set in-game time to random time
while true do
    count = count + 1
    if count % 10 == 0 then
        client.sendMessage(StartRecording())
    end if
    if count % 10 == 1 then
        client.sendMessage(StopRecording())
    end if
    if count % 500 == 0 then
        set new random travel height in h
    end if
    if count % 700 == 0 then
        set new random camera angle in c
    end if
    if count % 2000 == 150 then
        generate new target location in AREA
    end if
    if agent is close to target location then
        generate new target location in AREA
        go to new target location
    end if
    if true travel height is different from current travel height then
        correct travel height
    end if
    if new message is received from client then
        save message["image"] to savedir/images
        parse message["bounding_box"] and save the parsed bounding box to savedir/labels
        parse meta data from message and save parsed meta data to savedir/meta_data
    end if
end while
```

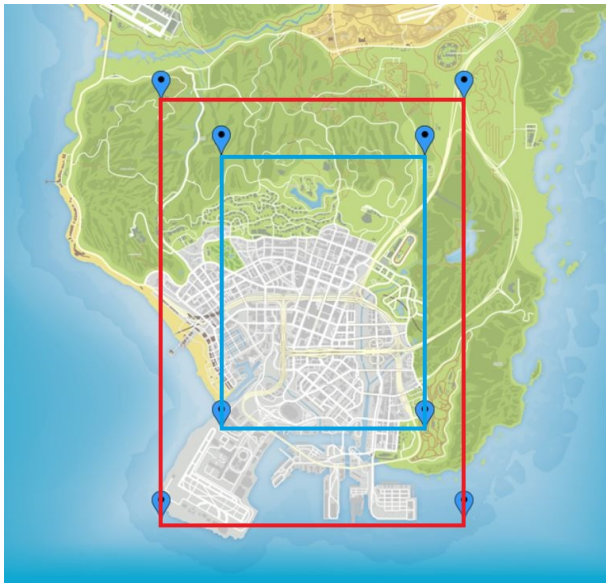

Figure 2.1.: The in-game area in which data was captured marked on the GTAV map. The area surrounded by the outer line was used to capture the DeepGTAV_HIGH dataset. The area surrounded by the inner line was used to capture the DeepGTAV_LOW dataset.

Data generation was conducted to produce approximately 45,000 images. Those were then cleaned according to section 2.2.1. Of the remaining images, 41,500 images were used as the DeepGTAV_LOW dataset. The first 40,000 of those images were used as the training set, while the remaining 1,500 images were used as the validation set.

2.4. Training

2.4.1. YOLO

The object detection system that has been chosen for this work is YOLO, a single-stage object detector. The architecture itself and its training process have been iteratively improved, as can be seen in [RDGF16], [RF17], [RF18] and [BWL20]. The main advantage of this architecture is that it is relatively lightweight compared to other object detectors, which allows relatively fast training times and very fast inference times on relatively small hardware, while still achieving good performance.

The specific implementation chosen for this work is Ultralytics-Yolov5 v1.0 [JCH+20]. This implementation uses models of different sizes (s/m/l/x), differing in the number of layers and the channels per layer.

Those models are similar to the normal *Yolo* models and the *tiny Yolo* models. Size and performance wise Ultralytics-Yolov5s is comparable to Yolov4-tiny and Ultralytics-Yolov5l is comparable to Yolov4 [BWL20].

For this work, mainly due to constraints in the training times, the Ultralytics-Yolov5s model was used if not otherwise specified. The Yolov5l model was used to achieve a good final performance and to validate that the findings probably also hold for larger models.

All models were adapted for 6 classes. Furthermore, the models were adapted for an image size of 1056 x 1056 pixels. Unless otherwise specified, the models were trained with the

standard configuration in the Ultralytics-Yolov5 v1.0 implementation. In the following, the most important of these settings are listed:

The models were trained using the SGD optimizer with nesterov momentum [SMDH13] with an initial learning rate of 0.01, a momentum of 0.937 and weight decay of 0.0005. The learning rate was scheduled according to

$$lr_e = \frac{1}{2}(1+\cos(\frac{e\cdot\pi}{total\ epochs}))\cdot 0.9+0.1$$

where lr_e is the learning rate in epoch e.

At training time, mixed precision training [MNA+17] was used to increase the training speed. This technique uses half-precision floating numbers to do the training, while using single-precision numbers in the final model. This reduces the memory consumption and training speed, while still preserving the final accuracy.

The images were augmented with hue-saturation-value augmentation [SK19], where the hue, saturation and value were randomly enlarged or decreased by a fraction of maximally 0.014, 0.68 and 0.36, respectively. The images were randomly scaled by a maximum factor of 0.5. The input images were augmented with mosaic augmentation [BWL20], where four images are combined into one training sample.

The Yolo5s models were trained with a batch size of 64, the Yolo5l models were trained with a batch size of 24.

All models were initially trained for 300 epochs, which resulted in convergence. For transfer training the best weights from those 300 epochs (by validation mAP) were further trained on the fine-tuning dataset for a total of 500 epochs. For testing the best weights (by validation mAP) from training were used.

2.4.2. Training Conditions

A comprehensive depiction of the different training conditions can be seen in Tab. 3.1.

Baseline: VisDrone Training

In this condition the models were trained using the training and validation set from VisDrone as described in section 2.3.1. The Yolo5s model was trained with 1,000, 2,000, 4,000 and all images from the VisDrone training set (VisDrone1k, VisDrone2k, VisDrone4k and VisDrone6.5k), always using the full VisDrone validation set. Furthermore, the Yolo5l model was trained with the full VisDrone training and validation set. For all these runs initial weights pretrained on COCO were used. The Yolo5s model was furthermore trained with random initial weights on all images from the VisDrone training set with the VisDrone validation set.

Only DeepGTAV Training

In this condition the model was trained on a portion of each, the DeepGTAV_HIGH or the DeepGTAV_LOW training set and the respective full validation set. The Yolo5s model was trained with 10,000 and 40,000 images from the DeepGTAV_HIGH training set (DeepGTAV_HIGH10k and DeepGTAV_HIGH40k). The Yolo5s model was additionally trained with 10,000, 20,000, 30,000 and 40,000 images from the DeepGTAV_LOW training set (DeepGTAV_LOW10k, DeepGTAV_LOW20k, DeepGTAV_LOW30k and DeepGTAV_LOW40k). The Yolo5l model was trained with 40,000 images from the DeepGTAV_LOW training set. For all these runs initial weights pretrained on COCO were used.

The Yolo5s model was furthermore trained with random initial weights on 10,000 images from the DeepGTAV_LOW training set.

Transfer Training

For this condition, the weights trained in *Only DeepGTAV Training* (section 2.4.2) were used as initial weights for training with VisDrone data.

The Yolo5s model was trained with initial weights that were pretrained on COCO and then trained on DeepGTAV_LOW10k with the VisDrone1k, VisDrone2k, VisDrone4k and VisDrone6.5k training set. The same was done for initial weights that were pretrained on COCO and then trained on DeepGTAV_LOW40k.

The Yolo5s model was trained with initial weights that were randomly initialized and then trained on DeepGTAV_LOW10k with the VisDrone1k and the VisDrone6.5k training set.

The Yolo5s model was trained on VisDrone 6.5k with initial weights that were pretrained on COCO and then trained on DeepGTAV_HIGH10k, DeepGTAV_HIGH20k, DeepGTAV_LOW30k or DeepGTAV_LOW40k.

The Yolo5l model was trained on VisDrone 6.5k with initial weights that were pretrained on COCO and then trained on DeepGTAV_LOW40k.

2.4.3. Training Setup

All training was conducted on a single node of the TCML Cluster[10] having an Intel XEON E5-2650 v4 and four GeForce GTX 1080 Ti (11GB GDDR5X memory).

2.5. Testing for an unequal Distribution of the Loss

To show that the per sample loss for a specific model is not equally or randomly distributed in the game world, experiments were conducted in the following way:

Two models were used for these tests: The Yolo5s model with COCO pretrained initial weights trained on DeepGTAV_HIGH10k and the Yolo5s model with COCO pretrained initial weights trained on DeepGTAV_LOW10k. Two different subsets of the DeepGTAV_HIGH and the DeepGTAV_LOW dataset with 10,000 images each were used for experimentation, that neither of the two models had seen in training. On these images the individual predictions of the two different models were made and the according loss per image in relation to the ground truth was calculated. The images on which the loss was calculated were split up along the mean of the camera height above the ground, resulting in 5,000 images with a camera height above the ground greater than the mean and 5,000 images lower than the mean for each dataset. The mean loss for those two groups was compared between each experimentation dataset and each model.

The same procedure was conducted for the camera pitch angles in the DeepGTAV_LOW experimentation dataset.

[10] https://uni-tuebingen.de/fakultaeten/mathematisch-naturwissenschaftliche-fakultaet/fachbereiche/informatik/lehrstuehle/kognitive-systeme/projects/tcml-cluster/

3. Results

3.1. Analysis of the Data from DeepGTAV

In the following, examples of the generated synthetic data are presented. In particular, the high visual fidelity and diversity and possible strengths and weaknesses will be shown. Examples of the different generated object classes are shown in Fig. 3.1. Special attention should be given to the models of pedestrians in-game, which are animated, thereby producing images where pedestrians are shown in many different poses.

DeepGTAV also allows the use of the in-game time cycle. At different in-game times, distinct visual qualities in the images are produced. Different shaders are used for different times of the day, the angle of shadows is different, cars have their headlights on at night, just to name a few of the differences. Examples can be seen in Fig. 3.2.

The capturing of images at different heights is also possible, resulting in distinct perspectives on objects. For example, at greater heights the objects in general are smaller, for lower heights more details on the objects can be recognized. Examples of images captured at 10m, 50m and 100m can be seen in Fig. 3.3.

Furthermore, the DeepGTAV framework allows the free manipulation of the camera angles and camera rotation. Of particular interest for UAV imaging is the pitch angle. Different camera angles result in different distinct features and silhouettes of the objects to be visible. Examples can be seen in Fig. 3.4.

For great heights or relatively flat camera angles DeepGTAV produces very small bounding boxes, due to the objects being far in the distance. Since those bounding boxes could not reasonably be recognized and resulted in bad performances in preliminary training results, such bounding boxes were removed as described in section 2.2.1. An example of a scene with and without those small bounding boxes can be seen in Fig. 3.5.

3.1.1. Weaknesses in DeepGTAV

The DeepGTAV framework has some weaknesses that are mainly due to some errors and bugs in the data generation.

In general, the bounding boxes are quite good and most of the time almost pixel perfect. There are some very small errors in the bounding boxes, probably due to synchronization problems with the stencil and depth buffers. Those errors only really make a difference for very small objects.

There are also some larger errors in the bounding boxes due to bugs. In particular, there sometimes is tearing, where one of the edges is at the wrong position as can be seen in Fig. 3.6 (top-left). There are also some bugs with the pedestrian bounding boxes, where either no bounding box is recognized at all or the bounding box is extremely small (see Fig. 3.6, bottom-right). For trucks with a leading vehicle and a large trailer the bounding box only encompasses the leading vehicle. For smaller truck models the full truck is in the bounding box as it should be.

Figure 3.1.: Examples of images exported from DeepGTAV with bounding boxes. The different classes are showcased, being approximately from left to right, top to bottom *pedestrian, car, van, truck, bus* and *motor*.

Figure 3.2.: Images captured at 0:00, 6:00, 12:00 and 18:00 in-game time (from left to right, top to bottom).

Figure 3.3.: Images captured at 10m, 40m, 70m and 100m in-game height (left to right, top to bottom).

Figure 3.4.: Captured images at $-20°$, $-45°$, $-90°$ and $-90°$ camera pitch angle (from left to right, top to bottom).

Figure 3.5.: A comparison before (left) and after (right) removing small bounding boxes with width or height smaller than 10px

Figure 3.6.: Examples of different errors in the data produced by DeepGTAV. In the top-left image a tearing error is shown, where one of the sides of the bounding box is at the wrong location. In the top-right image an example of a height error is shown. For this frame DeepGTAV exports a height above the ground of 20m while the actual height difference to the street is much larger. This is due to DeepGTAV interpreting the building below to be the ground. In the bottom-left image an example of a clipping error is shown. The camera in this image is inside of a building, in which it should not be. The building walls are invisible from the inside, but the bounding boxes are exported correctly. The bottom-right image shows an example of a bounding box error with pedestrians, where some of the bounding box borders are off.

Some errors in the height calculations arise when moving over buildings. The in-game calculation of the height above the ground recognizes large buildings as ground. In Fig. 3.6 (top-right) such an example is shown, where the extracted height is 20m, while the actual height above the street is much larger. In a similar way when traveling over water, not the height of the water level is used as a baseline for the height above ground calculation, but the floor underwater is set as height zero.

Due to limitations of the in-game path finding AI and the way the camera is set for low height image capturing, there are sometimes clipping problems for very low heights, where the camera is below ground or in a building. An example of this can be seen in Fig. 3.6 (bottom-left). In those cases, the camera can see through the wall, and bounding boxes are still produced correctly.

The extraction of the frames and bounding boxes takes rather long (about six seconds per extracted image). On very rare occasions GTAV crashes when running the data generation (about once every 24 hours). In those cases, data generation has to be restarted. One reason for this could be bugs in the depth calculations.

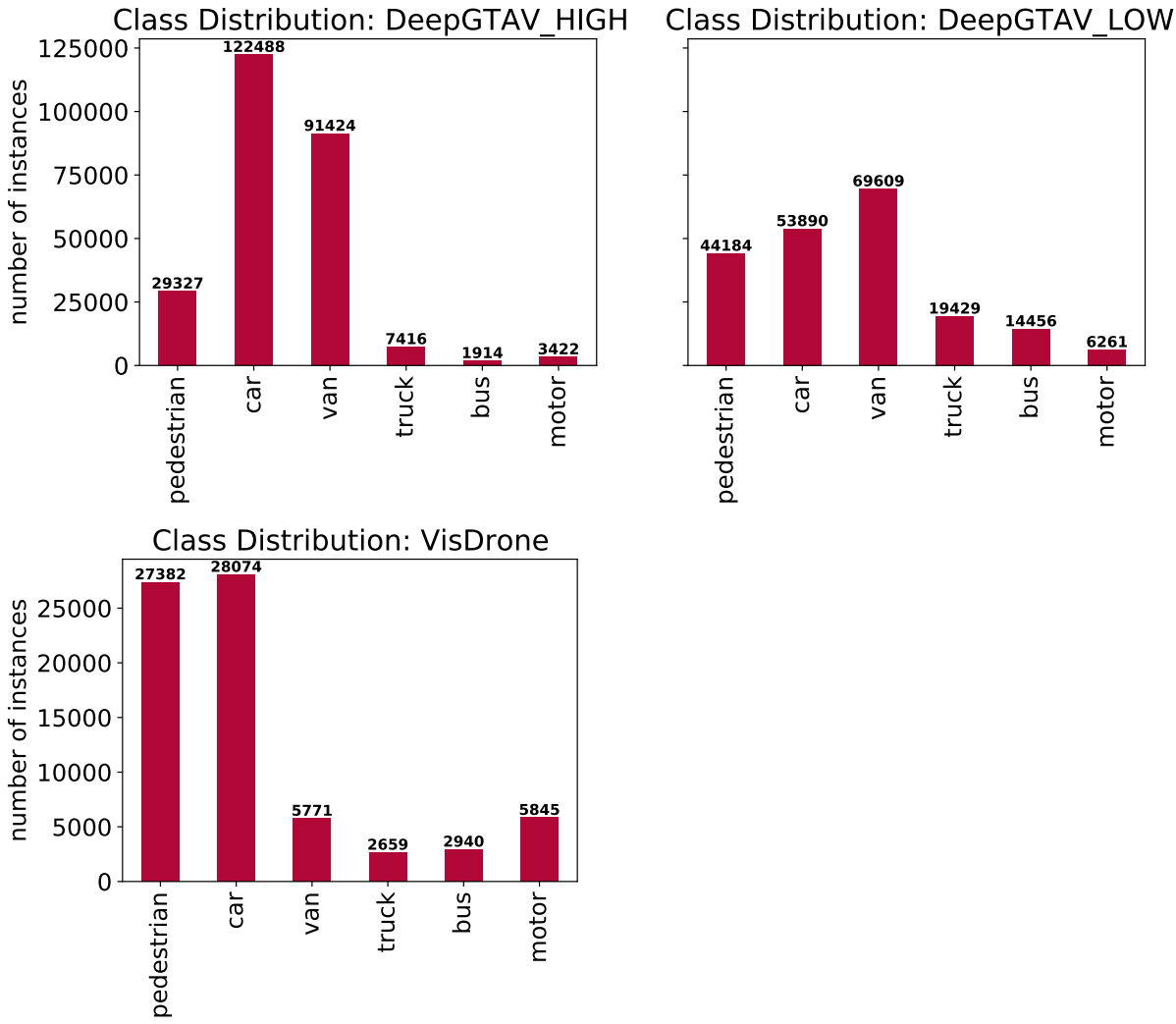

Figure 3.7.: The distribution of the different classes in the full dataset for DeepGTAV_HIGH (top left) DeepGTAV_LOW (top right) and VisDrone (bottom left). For the two datasets from DeepGTAV the training and validation set are included, for VisDrone the training, validation and test set are included.

3.1.2. Statistics and Comparison to VisDrone

The generation of data in the virtual environment allows the easy extraction of meta data. In the following, different statistics of the generated datasets are presented.

The distribution of the different object classes can be seen in Fig. 3.7. The average number of objects per image for DeepGTAV_HIGH is 5.1, while it is 5 for DeepGTAV_LOW and 50.2 for VisDrone.

The distribution of the height above the ground is shown in Fig. 3.8 and the distribution of the camera pitch angles is shown in Fig. 3.9.

3.2. Training Results

In Tab. 3.1 a comprehensive listing of all the training conditions and training results is given. Note that both, the mAP@50 and the mAP are reported. The mAP@50 refers to a mean average

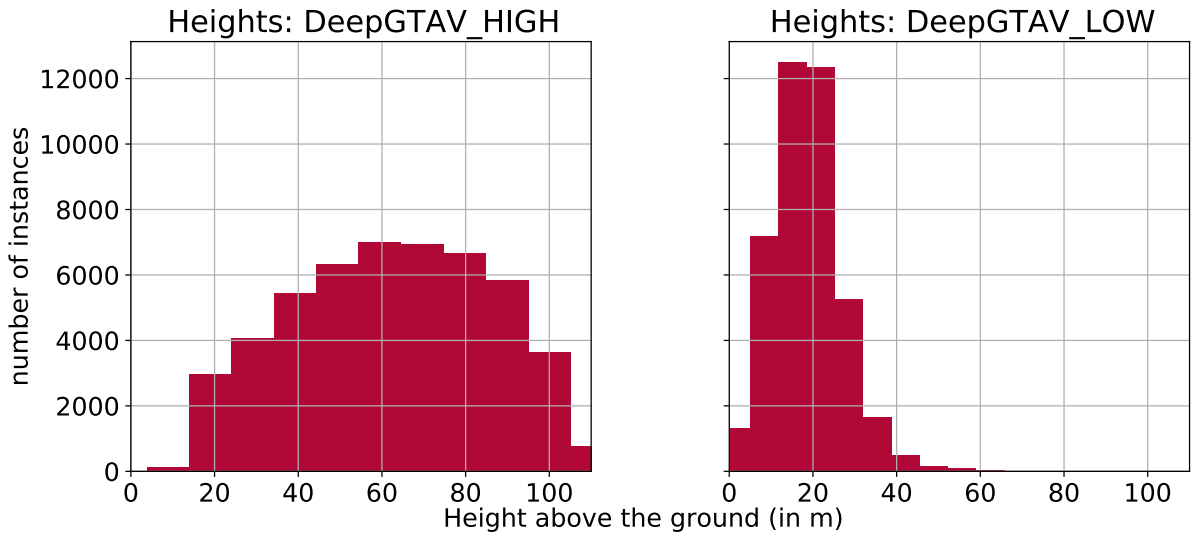

Figure 3.8.: The distribution of the camera height above the ground for DeepGTAV_HIGH (left) and DeepGTV_LOW (right).

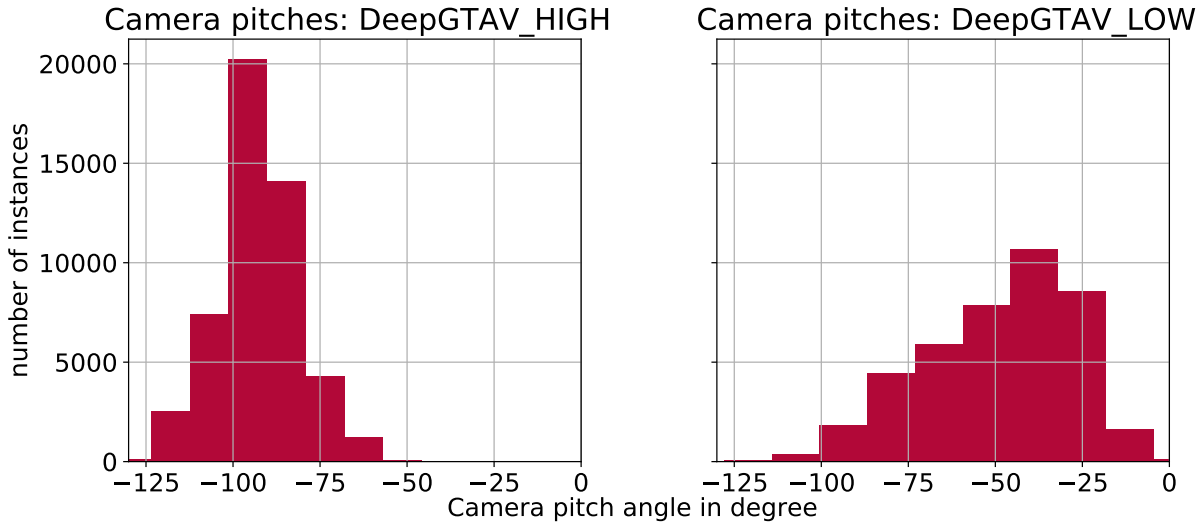

Figure 3.9.: The distribution of the camera pitch angle for DeepGTAV_HIGH (left) and DeepGTAV_LOW (right). A pitch of $-90°$ refers to the camera directly looking downward to the ground, while a pitch of $0°$ refers to the camera looking directly in front of the player vehicle in parallel to the ground plane.

precision with an IoU threshold of 0.5. The mAP refers to the mean average precision at IoU = 0.50:0.05:0.95 as it is described in [LMB+14]. This is the mean of the ten mAPs for IoU thresholds starting at IoU=0.5 and increasing by 0.05 until IoU=0.95 is reached.

In this section different subsets of those results will be presented in graphical comparison. For those comparisons, the mAP@50 on the VisDrone test set achieved by the different models is used.

In Tab. 3.2 the individual class mAPs are reported for some of the models.

In Fig. 3.10 bounding box predictions on the VisDrone test set from the Yolo5l model initialized with weights pretrained on COCO and only trained with DeepGTAV_LOW40k are shown.

Comparing the effect of the improvements in the DeepGTAV datasets

The improvements in the capturing process of DeepGTAV do show a significant improvement in the model performance. For 10,000 training samples the mAP@50 on the VisDrone test set is raised from 6.6% using DeepGTAV_HIGH to 15% using DeepGTAV_LOW. For 40,000 training samples the mAP@50 is raised from 7.5% on DeepGTAV_HIGH to 16.7% on DeepGTAV_LOW.

Comparing the effect of the synthetic dataset size

In Fig. 3.11 the effect of the size of the synthetic training dataset on the testing performance on VisDrone-test is shown.

Comparing the effect of dataset sizes for transfer learning

In Fig. 3.12 the effect of the real-world dataset size and the effect of the synthetic dataset size in the transfer learning condition is shown.

Comparing the effect of pretrained weights

In Fig. 3.13 the effect of using weights pretrained on COCO on the model performance can be seen. The use of weights pretrained on COCO improves the performance of the Yolo5s model trained on DeepGTAV_LOW by an absolute value of 3.7% mAP@50, while it improves the performance of the Yolo5s model trained on VisDrone6.5k by 2.2% mAP@50 and that of the Yolo5s model trained on VisDrone1k by 5%.

Testing for an unequal distribution of the loss

In Fig. 3.14 the mean loss per sample for samples above average height and below average height for the two models trained on DeepGTAV_LOW10k and DeepGTAV_HIGH10k on a subset of DeepGTAV_LOW and DeepGTAV_HIGH are shown. In Fig. 3.15 the mean loss per sample for samples above and below the average camera pitch angle are shown for the models trained on DeepGTAV_LOW10k and DeepGTAV_HIGH10k on subsets of DeepGTAV_LOW and DeepGTAV_HIGH.

Model	Pretrained weights	Training/validation data	Test mAP@50	mAP
Baseline: VisDrone Training				
yolo5s	COCO	VisDrone1k	17.18%	8.71%
yolo5s	COCO	VisDrone2k	22.98%	12.28%
yolo5s	COCO	VisDrone4k	29.33%	16.06%
yolo5s	COCO	VisDrone6.5k	31.88%	17.63%
yolo5l	COCO	VisDrone6.5k	35.36%	20.45%
yolo5s	none	VisDrone6.5k	29.65%	15.91%
Only DeepGTAV Training				
yolo5s	COCO	DeepGTAV_HIGH10k	6.62%	3.45%
yolo5s	COCO	DeepGTAV_HIGH40k	7.46%	3.85%
yolo5s	COCO	DeepGTAV_LOW10k	15.08%	7.66%
yolo5s	COCO	DeepGTAV_LOW20k	15.16%	7.80%
yolo5s	COCO	DeepGTAV_LOW30k	16.12%	8.24%
yolo5s	COCO	DeepGTAV_LOW40k	16.74%	8.65%
yolo5l	COCO	DeepGTAV_LOW40k	19.25%	10.34%
yolo5s	none	DeepGTAV_LOW10k	11.42%	5.48%
Transfer Training				
yolo5s	COCO→DeepGTAV_LOW10k	VisDrone1k	17.78%	9.25%
yolo5s	COCO→DeepGTAV_LOW10k	VisDrone2k	23.28%	12.40%
yolo5s	COCO→DeepGTAV_LOW10k	VisDrone4k	28.65%	15.73%
yolo5s	COCO→DeepGTAV_LOW10k	VisDrone6.5k	31.25%	17.27%
yolo5s	COCO→DeepGTAV_LOW40k	VisDrone1k	19.83%	10.41%
yolo5s	COCO→DeepGTAV_LOW40k	VisDrone2k	23.88%	12.94%
yolo5s	COCO→DeepGTAV_LOW40k	VisDrone4k	29.49%	16.32%
yolo5s	COCO→DeepGTAV_LOW40k	VisDrone6.5k	31.50%	17.50%
yolo5s	none→DeepGTAV_LOW10k	VisDrone1k	16.08%	8.08%
yolo5s	none→DeepGTAV_LOW10k	VisDrone6.5k	28.94%	15.72%
yolo5s	COCO→DeepGTAV_HIGH10k	VisDrone6.5k	31.54%	17.56%
yolo5s	COCO→DeepGTAV_HIGH40k	VisDrone6.5k	31.40%	17.40%
yolo5s	COCO→DeepGTAV_LOW20k	VisDrone6.5k	31.43%	17.44%
yolo5s	COCO→DeepGTAV_LOW30k	VisDrone6.5k	31.36%	17.45%
yolo5l	COCO→DeepGTAV_LOW40k	VisDrone6.5k	35.29%	20.35%

Table 3.1.: A comprehensive depiction of the training conditions and training results. All test mAP@50 and mAP are reported on the VisDrone test set. In the pretrained weights for example *COCO→DeepGTAV_HIGH10k* refers to the model with COCO pretrained weights trained on the first 10000 images of DeepGTAV_HIGH for 300 epochs. The best weights (by validation mAP) are now taken to be trained on the according training/validation data (e.g. VisDrone1k). The number behind the dataset refers to the number of training images of the dataset that were used for the training. For example, for VisDrone2k only the first 2000 images of VisDrone-train are used for training. The enitre validation set of the respective datasets was always used.

Figure 3.10.: Bounding box predictions of the Yolo5l model with initial weights pretrained on COCO, trained purely on DeepGTAV_LOW40k. Bounding boxes are displayed if the model has a confidence greater than a threshold of 0.4.

Model	mAP	pedestrian	car	van	truck	bus	motor
Yolo5l on COCO→DeepGTAV_LOW40k							
	10.34%	4.57 %	27.20%	8.12%	5.03%	14.85%	2.26%
Yolo5s on COCO→VisDrone1k							
	8.71%	5.81%	26.35%	7.68%	2.94%	4.66%	4.79%
Yolo5s on COCO→DeepGTAV_LOW40k→VisDrone1k							
	10.41%	6.21%	28.71%	11.48%	3.89%	6.75%	5.43%

Table 3.2.: The total mAP and the individual class mAPs of different models. The convention *"Yolo5s on COCO→DeepGTAV40k→VisDrone1k"* describes, that the Yolo5s model was initialized with weights pretrained on COCO, then trained on DeepGTAV_LOW40k and then trained on VisDrone1k.

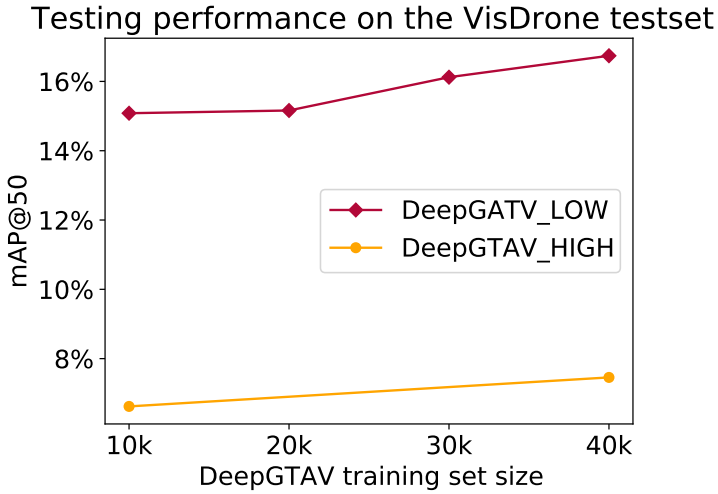

Figure 3.11.: The reached mAP@50 on the VisDrone test set against the size of the synthetic training set. All shown models used initial weights pretrained on COCO.

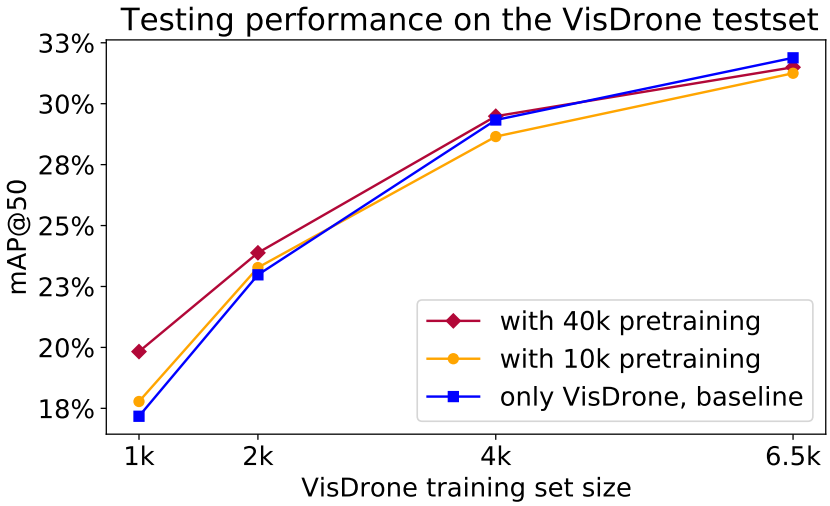

Figure 3.12.: The reached mAP@50 on the VisDrone test set against various sizes of subsets of the VisDrone training set used for fine tuning. In the condition "only VisDrone, baseline" weights pretrained on COCO were used and no further pretraining on DeepGTAV was conducted. For "with 10k pretraining" and "with 40k pretraining" models were initialized with weights pretrained on COCO. Then they were trained with 10,000/40,000 images from the DeepGTAV_LOW dataset and then those weights were further trained on the respective portion of the VisDrone training set.

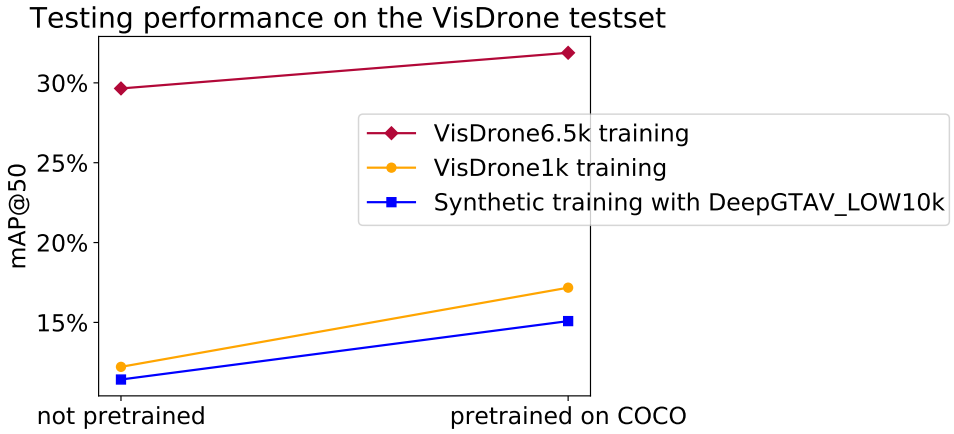

Figure 3.13.: The reached mAP@50 on the VisDrone test set when either using initial weights pretrained on COCO or random initial weights. The improvements are shown for the Yolo5s model trained on VisDrone6.5k, VisDrone1k and DeepGTAV_LOW10k.

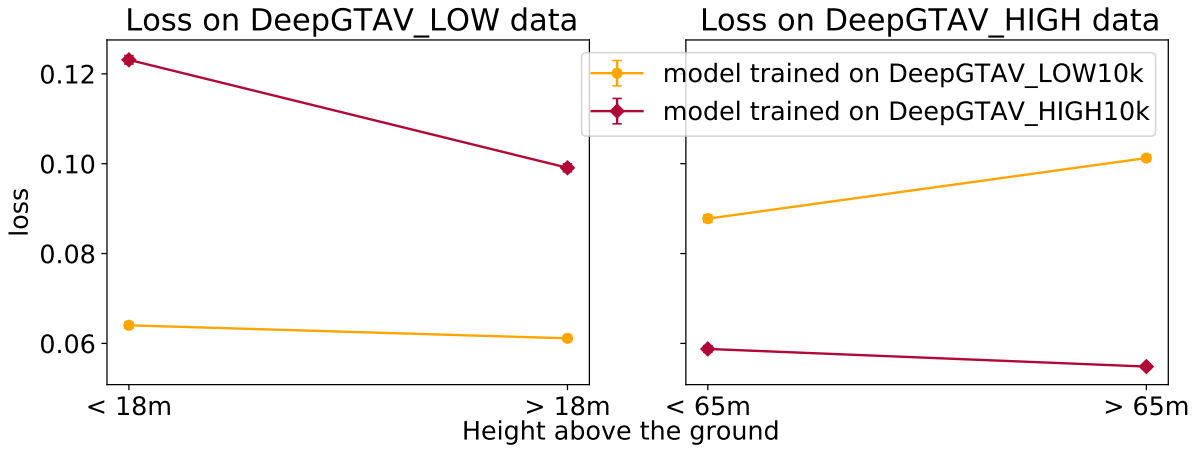

Figure 3.14.: Differences of the mean loss for samples below and above the mean camera height. Note that all means are plotted with error bars for the standard error of the mean, which are too small to be visible in the figure.

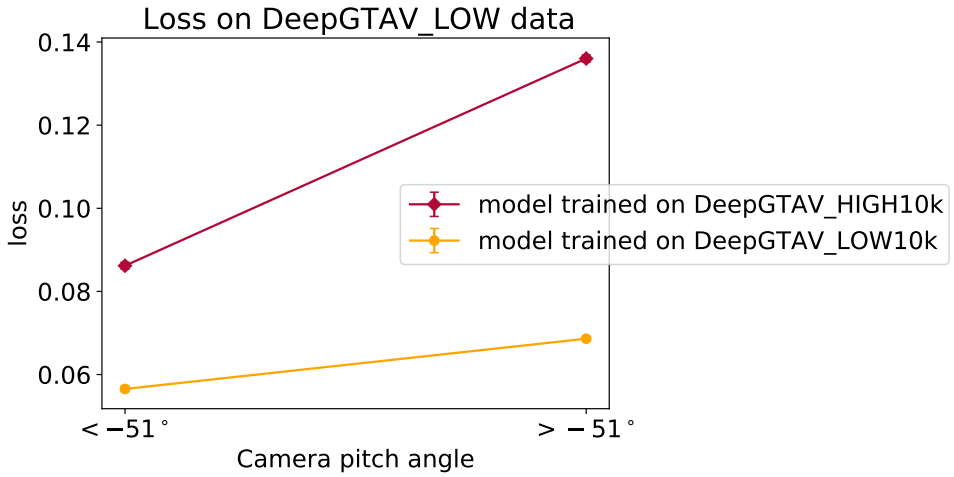

Figure 3.15.: Differences of the mean loss for samples with a camera pitch below and above the mean camera pitch. Note that all means are plotted with error bars for the standard error of the mean, which are too small to be visible in the figure.

4. Discussion

4.1. Analysis of the Datasets

When comparing the class distributions of DeepGTAV_HIGH and DeepGTAV_LOW (Fig. 3.7) it can be seen that the changes that were made do in fact result in a more balanced class distribution in DeepGTAV_LOW, which better resembles the class distribution of VisDrone.

Although modifications to the game files were made in such a way to balance the different classes evenly, they are in fact not evenly balanced. This is due to limitations in the in-game spawning systems, due to which specific objects can not be spawned in specific areas. In particular trucks, buses and motorcycles were spawned less often. However, the lower prevalence of the classes *truck, bus* and *motor* does match the distribution in VisDrone. On the other hand, the high amount of *van* objects in DeepGTAV_LOW compared to VisDrone probably introduces a bias. In general, if the class distribution of the training set is different from that of the test set this produces a disadvantageous bias [HYKL04, SKW06]. It stands to reason that the performance of the training in this work could be significantly improved by fitting the class distribution of the synthetic training data to the real-world dataset.

In spite of the changes that were made to increase the number of in-game objects (see section 2.1.2), the average number of objects per image actually decreased slightly from DeepGTAV_HIGH to DeepGTAV_LOW (on average 5.1 objects per image to 5 objects per image). This is mostly due to the lower travel height resulting in less objects per image on average. Therefore, the changes that were made to increase the number of spawned objects are obscured by this difference in travel height, but those changes did in fact have a positive effect and resulted in more objects captured. There are some other limiting factors regarding the average number of objects in an image. Some different settings in the game files that limit the number of objects at any given time have not been modified yet. Furthermore, the game heap has a finite size, therefore only a limited number of objects can be spawned in the game world at any given time. Both of these issues could be solved by thorough examination and modification of the GTAV game files.

In total, however, the average number of objects per image from DeepGTAV is still much lower than that of VisDrone (around 5 objects per image on average vs. 50.2 objects per image). This could potentially result in the DeepGTAV images being much less informative than those from VisDrone.

In Fig. 3.8 it can be seen that the changes made in the data generation scripts produced different, distinct distributions of the camera height as desired. The distributions show, that the true travel heights are not the same as the specified travel heights in the data generation scripts (7-97m and 0-20m, respectively). This is due to the in-game path finding AI sometimes optimizing the paths and traveling at different heights. Furthermore, when moving on uneven ground (e.g. upwards or downwards cliffs) the travel height is different for a short period until the correct travel height is assumed again.

Fig. 3.9 shows that the changes made to the data generation scripts to produce DeepGTAV_LOW did in fact have the desired effect of producing training samples with more diverse camera pitch angles. Again, the camera angles do not exactly map the angles specified in the

data generation scripts. This is due to the random vehicle movement in the game.

From visual inspection the camera angle and height distribution in DeepGTAV_LOW is more similar to that of VisDrone. VisDrone has images at rather low heights and with diverse camera angles (see Fig. 3.10 for VisDrone images), which is properly achieved by DeepGTAV_LOW.

4.2. Analysis of the Training Results

First of all, it has to be noted that the models employed in this work do in fact achieve close to state-of-the-art performance for the VisDrone-DET task. The Yolo5l model does in fact achieve performance comparable to that of an ensemble model of Yolov3 and faster R-CNN called EnDet which is reported in [Pai19] with an mAP@50 of 37.27%. The Yolo5l model has an mAP@50 of 35.36% after training on VisDrone6.5k.

It must be noted that the performance for Yolo5l is reported only for the 6 classes specified in this work, while the performance of EnDet is reported for all ten classes of VisDrone. Furthermore, the performance of Yolo5l is tested on the VisDrone test-dev dataset, while the performance of EnDet is tested on the VisDrone test-challenge set, of which the ground truth labels have not been published. Those two differences should not contradict the assessment that the models used in this work are state-of-the-art.

The most impressive finding of this work is that the Yolo5l model achieves reasonable real-world performance with an mAP@50 of 19.25% on the VisDrone test set. This is achieved by training with the DeepGTAV_LOW40k dataset without ever training on any real-world data. Tab. 3.2 shows that this model achieves at least some performance for all the classes used. The good performance of the Yolo5l model also indicates that the findings discussed in the following would also scale for larger models.

Comparing the effect of the improvements in the DeepGTAV datasets

The improvements in the capturing process between the datasets DeepGTAV_HIGH and DeepGTAV_LOW do in fact significantly increase the model performance on the VisDrone testing set, when not training further on VisDrone (see Fig. 3.11). This is most likely due to the fact that DeepGTAV_LOW better captures the underlying distribution of VisDrone as discussed above in section 4.1. To discover the different effects of the specific changes additional tests would have to be made.

Visual inspection of the predicted labels shows that *car* objects in VisDrone often are misclassified as *van* (see Fig. 3.10). This is probably due to the different class distributions of *car* and *van* in DeepGTAV_LOW and VisDrone, as the *van* class is much more prevalent in DeepGTAV_LOW. Furthermore, there is some ambiguity between the *car* and *van* class in VisDrone. Large cars are labeled as *van*, but the difference between those cars can only be seen at relatively low heights with flat camera angles. Those same objects are labeled as *car* at greater heights. In DeepGTAV those objects are still labeled as *van* at those greater heights. Furthermore, the car models in GTAV are fictitious and only roughly based on real-world car models. This could result in an additional distribution gap. It is to be expected that a relatively large performance increase could be achieved by matching the class distribution better and resolving the ambiguity between *van* and *car*.

In Fig. 3.10 it can also be seen that the model performs reasonably well for images recorded at day and at night. This is probably due to DeepGTAV using the in-game time cycle of GTAV producing good training images for all lighting conditions. The model also already shows

reasonably good performance for the *pedestrian* class, but only for standing people. Sitting people are generally not recognized that well.

Comparing the effect of the dataset size for synthetic training

There seems to be a positive effect of the size of the synthetic dataset on the testing performance. This is the case for both, the DeepGTAV_HIGH and the DeepGTAV_LOW dataset. These effects do however appear to be rather small in relation to the amount of additional training data necessary, having as absolute mAP@50 increase of 0.9% from 10,000 to 40,000 images on DeepGTAV_HIGH and an absolute mAP@50 increase of 1.7% from 10,000 to 40,000 images on DeepGTAV_LOW. Those improvements are also rather small when set in relationship to the computational resources required to achieve them, as the data generation time and the model training time both are multiplied by a factor of four for the four times larger synthetic training dataset. On the other hand, as there are only computational resources needed to do this, in a sense these improvements come very cheap. Furthermore, those improvements come without any undesired changes to the model, e.g. a decrease in model inference speed or an increase in model size, so for an actual deployment it seems reasonable to generate as much synthetic data as possible until no further improvements are possible.

Why the effect is not as great as the one reported in [JRBM+16]

In general, the performance achieved for the real-world application by only training on synthetic data is already quite good, considering that only computational resources were required and no recording or labeling of real-world data had to be done. Nonetheless, the performance is definitely worse than the one achieved in [JRBM+16], where synthetic training even outperforms training with a different real-world dataset. There are some justifications as to why this could be the case:

First, there is only one specific class mAP@70 for *car* reported in [JRBM+16], while here the total mAP@50 of six classes is reported. Additionally, the car class reported in [JRBM+16] is comparably relatively easy, being the class with the best state-of-the-art mAP@70 in KITTI and also the class with the best reported class-mAP across all models in [Pai19].

Second, VisDrone seems to be a generally harder problem than KITTI, which can be seen in different top performances achieved by state-of-the-art-models: The current state-of-the-art mAP@70 for the car class of KITTI 2D is around 95%[1]. The state-of-the-art mAP@50 performances for the pedestrian and cyclist classes are both around 80%. On the other hand, the state-of-the-art mAP@50 performance for all classes of VisDrone is around 55% [Pai19].

Finally, the generation process of KITTI and the virtual data from [HCW19] is standardized to a very specific setup with the same camera perspective and position on a car that only moves on the ground [GLU12]. This results in many less parameters that can be varied than the free camera perspective in the air that is prevalent in VisDrone. For this reason, producing a simulated dataset that correctly matches the setup of KITTI 3D is an easier task than producing such a dataset that correctly maps VisDrone. This can already be seen from the different generation procedures used for DeepGTAV_HIGH and DeepGTAV_LOW and their vastly different achieved performances.

[1] see `http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=2d` for a current leaderboard

Comparing the effect of the dataset size for transfer training

The effect of the real-world dataset size and the synthetic dataset size can be seen in Fig. 3.12. There is no significant positive effect from pretraining on the synthetic data for the VisDrone4k and VisDrone6.5k dataset. This is the case for the Yolo5s model and the Yolo5l model, being pretrained on DeepGTAV_LOW or DeepGTAV_HIGH with any synthetic dataset size. However, there is a positive effect of synthetic pretraining when using only 1,000 or 2,000 images from the VisDrone training set. As expected, the effect of using synthetic data to pretrain is larger, the smaller the real-world dataset is. This is most likely due to the small real-world dataset not containing enough information on itself, thereby allowing an improvement from the synthetic data. It stands to argue that there are two effects: on the one hand the synthetic data provides additional information, that is not contained in the real-world data. On the other hand, the synthetic data provides information that is conflicting with the real-world data, thereby leading to unfavorable configurations. For small real-world dataset sizes the first effect outweighs the second, for larger real-world datasets the second effect is larger than the first. This would result in the achieved performances, where there is an improvement from using synthetic data for the small datasets (VisDrone1k, VisDrone2k) but there is a decrease in performance for the larger real-world datasets (VisDrone4k, VisDrone6.5k). The effect of unfavorable configurations could be due to optimizations of configurations that are good for the synthetic data but result in local optima for the real-world data, which can not be left in the fine-tuning training. It could also be due to favorable configurations in the initial weights pretrained on COCO being "unlearned". There is an effect of the larger real-world datasets resulting in better performance, with a curve of diminishing returns. Similarly, there is an effect that the larger synthetic dataset used for pretraining results in better performance than the smaller synthetic dataset. Both of these effects of larger dataset sizes resulting in better model performances were to be expected for a deep learning task.

In Tab. 3.2 a positive effect on all the individual class mAPs of a Yolo5s model trained on VisDrone1k can be seen for pretraining with DeepGTAV_LOW40k. This indicates that a performance increase is possible for all classes in VisDrone.

The effects reported here are not as large as the ones reported in [HCW19] when comparing training on a real-world dataset of 4,000 instances with or without pretraining, using a synthetic pretraining set of 40,000 instances. [HCW19] report an absolute increase of the average precision for the pedestrian class of KITTI 3D [GLU12] by 4.26%, 2.07% and 4.5% for moderate, easy, and hard objects, respectively. The smaller effects in this work could be due to some different factors, similar though those noted above for the worse performance than in [JRBM$^+$16]:

First, in [HCW19] only one class mAP is reported while six classes are examined in this work. Second, KITTI 3D still seems to be a generally easier problem than VisDrone. The state-of-the-art performance of the mAP@70 for the car class in KITTI 3D is around 80%, while the mAP@50 for the pedestrian and cyclist classes in KITTI 3D are around 45% and 70% respectively. The state-of-the-art total mAP@50 performance for VisDrone on the other hand is around 55% [Pai19]. Third, the KITTI 3D data generation process is standardized as described above. Another difference is that in [HCW19] the model was trained and tested on 3D LiDAR data, while the model in this work was trained and tested with 2D image and bounding box data. Finally, the dataset examined in DeepGTAV-PreSIL is smaller (KITTI4k) than VisDrone6.5k, so maybe this negative effect of using synthetic data when using larger real-world datasets can not yet be seen. Although, there already is no performance improvement to be seen for pretraining VisDrone4k with synthetic data (see Fig. 3.12).

In total, the nonexistent or negative effect of pretraining with synthetic data for larger real-world datasets relativizes its use for real-world applications. If real-world performance was actually important for the specific application, the additional cost to produce a large real-world dataset can be taken, making synthetic pretraining in this context useless.

Comparing the effect of pretrained weights

The hypothesis that the effect of using weights pretrained on COCO would be larger for synthetic training data than for real-world training data appears not to hold. In Fig. 3.13 the effect of using weights pretrained on COCO on VisDrone1k training, VisDrone6.5k training and DeepGTAV_LOW10k training are shown.

Although there is a larger difference for using pretrained weights on DeepGTAV_LOW10k training than there is on VisDrone6.5k training (3.66% vs. 2.23% absolute mAP@50 difference), this is most likely due to the lower final performance reached in DeepGTAV_LOW10k training. This is confirmed by the also high difference of 4.96% absolute mAP@50 for VisDrone1k training. It is assumed, that the pretrained weights contain some information that is not contained in the training data, thereby improving the performance. Therefore, it stands to reason that with an increased performance (meaning more information learned from the training data) some of this newly learned information overlaps with the one from the pretrained weights, which results in the advantages of using the pretrained weights being not as large anymore.

It seems that the hypothesis that pretrained real-world weights would disproportionately "enable" synthetic training to produce better real-world performance does not hold. Nonetheless the use of initial weights pretrained on COCO still produces a better final performance and faster convergence times in all conditions.

Other improvements on the training

There are some venues that could be taken to further improve the model performance.

First of all, apart from the purely synthetic training scheme and the pretraining scheme as described in this work, another training scheme intuitively seems reasonable: One could train the object detector in conjunction with synthetic data and real-world data simultaneously. This would be similar to the training scheme employed in [YWS+18]. Preliminary results of such a training scheme have shown that the addition of small samples of synthetic data (5,000 instances) would improve the real-world performance of the object detector (by around 3% absolute mAP@50), but the addition of larger synthetic samples (10,000 or more) would decrease the performance. This is most likely due to overfitting on the disproportionately larger synthetic data. Furthermore, the improved performance could also be attributed to the mosaic data augmentation that was employed. Additional images without any objects would produce additional contexts for the mosaic augmentation which would result in better generalization properties. A combined training scheme would have to solve the problem of decreasing performance to optimizing for the synthetic data while neglecting real-world data. One way to solve this would be to ignore the gradients of synthetic images at training time if they contradicted with those of real-world data. Another approach would be to weight in the gradients of real-world data stronger or to present the real-world instances with a higher probability at training time.

Other works try to improve the real-world performance by solving the problem of the domain mismatch between the synthetic training data and the real-world testing data [CLS+18]. This is done by introducing domain adaptation components. This definitely is also a venue that is promising to improve the real-world performance and applicability of synthetic training data.

Further works should also investigate whether the usefulness of synthetic data still exists for larger object detectors. To this end similar training schedules as proposed in this work could be employed for larger object detection models or multi-stage object detectors like [CV18]. The results that have been achieved in this work for Yolo5l are promising indicators that the usefulness of synthetic data will in fact persist.

4.3. Motivated Data Generation

Testing for an unequal distribution of the loss

The expected hypotheses concerning the loss distribution for different models are confirmed by the differences of the means shown in Fig. 3.14 and Fig. 3.15. In particular, this shows that an object detector only trained on samples from greater height performs better for greater heights and an object detector trained on samples with lower heights performs better on lower heights. The same is true for an object detector trained with flat camera angles, which is more resistant to testing data with flat camera angles than an object detector that was only trained on images with steep camera angles. From this follows that there would be further knowledge to be gained from the underrepresented domain, e.g. an object detector trained on greater heights would learn disproportionately more from additional samples at lower height than from additional samples at the great height, which would ultimately improve the performance.

The difference in general model performance (The model trained on DeepGTAV_LOW generally performs better on DeepGTAV_LOW and vice-versa) can be explained with the general difference of the distribution between the two datasets. Apart of the shown criteria, e.g. height above the ground in Fig. 3.14, the datasets also differ in the camera angle and location due to the data generation processes used. This results in the differences of the general performance, meaning that for example the means of the loss of the model trained on DeepGTAV_LOW tested on DeepGTAV_LOW are generally smaller.

From the general downward trend of the loss for greater heights it would be tempting to conclude, that samples from greater height are easier (see Fig. 3.15). In the same way it would be tempting to conclude that flatter pitch angles are generally harder, from the general upwards trend of the loss for flatter pitch angles (see Fig. 3.15). Both of these conclusions are invalid, since one can not distinguish whether the worse performance is due to the different conditions being actually more challenging or due to the models not having learned the different conditions sufficiently well. It is to be stressed that conclusions can only be made about the relations of two models, which are conclusions like *"for changes in condition X model A performs better than model B"*. From such a statement it can be concluded that model A has more knowledge about a specific condition than model B and that model B could potentially learn more about this condition from some of the training data of model A, as has been done above. In general, such relations between the models are seen as two-way-interactions in the graphs.

For further understanding of the distribution of the loss in the environment further tests should be conducted. Of particular interest would be the distribution of the loss dependent on the x/y-location. But tests for such an unequal distribution of the loss dependent on the location are confronted with some challenges: First of all, since the environment is not static (objects like cars move around), one would have to control those changes. A much greater problem is that the underlying assumption of a monotonous relation between the varied meta variable and the loss, which made the analysis conducted above relatively simple, probably does not hold. For example, for the height it can be assumed, that a model that performs good on low heights and bad on higher heights performs even worse for even greater heights, and that

the performance is monotonically decreasing with increasing height. Such an assumption does not seem reasonable for the location. Therefore, different tests would need to be applied. One experimental setup that could test for different knowledges of the models for different environments would be the following: Two datasets would have to be generated, one that only contains imaging in an urban housing area and one that contains imaging from a desert area. Then two models would be trained on each of those datasets. A third testing dataset would have to be generated, that contains imaging from both environments (urban housing and desert) that are approximately divided by a border. Again, the individual sample losses of both models would be calculated on this testing dataset. The testing dataset would be grouped into the two areas and the means of the losses for those two groups would be computed. With this data a similar analysis as described above for the height and camera pitch angle could be made. A similar two-way-interaction as seen in the analysis above would be expected.

Iterative sampling

After establishing, that the individual sample loss for a specific model is in fact not distributed equally in the game world, and that further knowledge can be obtained by the model from training samples generated in the regions with higher loss, it is reasonable to use this for an iterative training loop, as introduced in section 1.3.1. Such an iterative training should in fact result in faster training, the production of a better dataset and a better final performance. The main problem of such an iterative training system is the step 3 of Alg. 1:

3: generate **data** based on some informed metric of the information content for the current **model**

In the following I will present three procedures that are promising to yield good results for this step: local search with momentum and random restarts and two reinforcement learning approaches.

The local search with momentum and random restarts would start at a random state in the environment. In this context the state entails the location and the camera angles ($s = (x, y, z, camrot_x, camrot_y, camrot_z)$). The momentum is a tuple of small changes to the current state ($m = (\Delta x, \Delta y, \Delta z, \Delta camrot_x, \Delta camrot_y, \Delta camrot_z)$). The environment is traversed by moving from the current state in the direction of the momentum and an additional small direction. This additional small direction is added to allow changes in the direction, otherwise the momentum would always point in the same direction. After each step, the momentum is updated according to the change of the loss. If the loss was increased, the momentum gets increased in this direction, resulting in more movement towards this direction. After generating a limited amount of data in this way, a new episode is started by generating a new random starting state and a new random momentum. Pseudo code of the local search with momentum and random restarts is shown in Alg. 10.

The problem of learning the distribution of the loss in the environment and then leveraging this knowledge to maximize the loss in the samples produced can also be formulated as a reinforcement learning problem. This can be done in the following way: The loss is considered to be the reward at a specific state in the environment. The states are again considered to be tuples of the location and camera rotation ($s = (x, y, z, camrot_x, camrot_y, camrot_z)$). The actions are small changes in the state ($a = (\Delta x, \Delta y, \Delta z, \Delta camrot_x, \Delta camrot_y, \Delta camrot_z)$) similar to the momentum above.

By discretizing the environment this problem could now be solved with traditional RL algorithms like for example an Epsilon-Greedy-approach [Wat89]. Alternatively, more elaborated

Algorithm 10 Pseudocode for a local search with momentum and random restarts.

Require:
Environment: the environment in which data generation shall be conducted
λ**:** a parameter limiting the size of the momentum updates
model: current iteration of the object detection model

$s \leftarrow$ random state $\in$ *Environment*
$m \leftarrow$ random initial momentum
while true **do**
  $r \leftarrow$ random movement in random direction
  Go to $s = s' + m + r$
  $instance \leftarrow$ export data at the current location
  $loss_{current} \leftarrow model(instance)$
  $\Delta m = (loss_{current} - loss_{last}) \cdot (s - s')$
  $m \leftarrow m + \lambda \cdot \Delta m$
  if episode length is reached **then**
    $s \leftarrow$ random state $\in$ *Environment*
    $m \leftarrow$ random initial momentum
  end if
  $s' \leftarrow s$
end while

systems that are able to deal with continuous state spaces like Deep-Q-Learning [SQAS15, MKS+13, MKS+15] could be used.

Instead of the model loss, the size of the gradient or the mAP score of the sample could be used as a metric of its information content. One general problem of all these approaches is, that it is generally not possible to differentiate between the metric being high due to the model not having learned the sample yet, or due to the sample being generally hard. The gradient size should be the metric that is most robust to this problem, since it is relative to the actual changes that are made in the model, which are relative to the amount learned from the specific sample.

Another problem with such methods could be, that they are generally instable to errors in the labeling. Such methods would especially seek out such samples with errors, since their loss or gradient would be high, and their mAP would be low. This does however only hold for systematic errors in the labeling (e.g. a specific car model is always labeled wrong) and not for random errors.

In the current state of the DeepGTAV framework it was not possible to implement those data generation schemes, as the data generation procedure took too long to do a feasible search in the environment, and since random restarts were not possible. This is because after teleportation in the game world, the ground truth exportation would not work anymore. Therefore, the game would have had to be closed and relaunched for every random restart, which would have taken too much time.

4.4. On the Advantages of Synthetic Data

In this work it has been shown again that synthetic training data is a viable alternative to real-world training data. In particular, it can be seen that with relatively little effort the already existing high fidelity simulations in computer games can be leveraged. In a similar way this approach could be applied to different virtual worlds in other video games.

On the price of producing training data

One of the advantages of synthetic data is the relatively cheap price compared to real-world data. The synthetic data only requires computational resources, while the real-world data requires expensive labeling work by human annotators. To put this in some perspective, the labeling of a dataset comparable in size to DeepGTAV_LOW with 50,000 images and a total of 250,000 bounding boxes would cost around 12,000$ when labeling with only one annotator per sample [2]. It is a standard practice for labeling tasks to use multiple labelers for the same label to increase label accuracy, which would result in a multiplication of the total price by the number of labelers used. It could be argued that in the current state DeepGTAV also does not produce perfect labels, but this can be reasonably expected to be fixed. Furthermore, the cost of recording such a dataset in the real world has not been considered for this calculation.

On the other hand, the computational costs of the data generation process to produce such a synthetic dataset are marginally small. In this case it would be the cost of running a personal computer for six days, which is rather negligible. For a fair comparison, the costs of adapting the DeepGTAV framework should also be considered, although they are somewhat intangible.

In general, human annotated real-world data is prone to having labeling errors. For synthetic data this is not the case. Although the DeepGTAV framework in the current state produces inexact bounding boxes this is only the case due to errors in the extraction code. Those can be solved, yielding pixel perfect annotations.

Accessibility of meta data

The easy accessibility of meta data enables better means of analysis and introspection of the generated data. For many real-world datasets, this data has not been captured at recording time and therefore is not available.

Such meta data could be used in production environments to analyze performance problems as well as in future scientific research. Examples for such analysis are the ones that were made in this work, in particular in section 3.1, section 3.2, section 4.1 and section 4.3.

An example of the applicability for scientific research would be the following: if one wanted to build an ensemble system of UAV object detectors, where different object detectors are used at different heights to improve the performance, one could first test this hypothesis with purely synthetic data at a fraction of the cost, before actually testing it with real-world data.

Synthetic training data frameworks can also be used to produce different kinds of ground truth data in conjunction, producing very rich datasets. With almost no modifications, the DeepGTAV framework can also be used to extract other ground truth label data. In particular, instance segmentation data, depth data and LiDAR data can be extracted with minimal modifications.

4.5. Possible Improvements

There are many routes that can be taken to further improve the DeepGTAV framework. First of all, the errors in the data generation process as discussed above should be fixed. In a similar

[2] The prices are calculated for the Google AI Platform Data Labeling Service (`https://cloud.google.com/ai-platform/data-labeling/pricing`) and Amazon SageMaker (`https://aws.amazon.com/sagemaker/groundtruth/pricing/`) with prices as of the 20. Oct. 2020. The prices are calculated for 250000 $units \cdot 49\$/1000 units = 12250\$$ and 50000 $objects \cdot 0.08\$/object + 250000\ units \cdot 0.036\$/unit = 13000\$$, respectively.

vein, the code quality of the DeepGTAV framework should be improved and some code testing should be implemented.

In its current state the DeepGTAV framework requires a lot of time to extract the labeling data with around six seconds per frame. The main bottleneck are the long calculations of the ray-casting that is required for the retrieval of depth and occlusion information. Currently those calculations are done on the CPU with little optimization. It is to be expected that the speed can be increased a lot with some optimization and by running those calculations on the GPU instead of the CPU. If real time performance were to be achieved, this would even enable DeepGTAV to be used as a versatile environment to train and test reinforcement learning agents. Additionally, real time performance or at least faster performance would also be necessary to allow active training scenarios, as described in section 1.3.1 and section 4.3

With small modifications the framework can be adapted to produce higher resolution data, e.g. 4k images. This is especially relevant for UAV object detection, as the small object size from great heights is one of the limiting factors for the object detectors performance.

With little modifications the DeepGTAV framework could also export video data. Video data is very expensive to label in real-world datasets, but this would not be a problem using this framework. Such video data could then be used e.g. for object tracking tasks or object detectors that also leverage temporal information [ZDYW18, ZXD+17, ZWD+17, KLY+17, KOLW16, HKP+16].

Since the DeepGTAV framework exports the objects by in-game model name, with little modifications the framework can be used to export different or new object classes. Classes for which ground truth labeling data could be produced with almost no modifications are boats, airplanes, helicopters, and construction vehicles. The ground truth data could even be modified in such a way to output the specific car models, allowing to train an object detector on them individually. Similarly, detections of people by age or gender would also be possible.

With minimal modification, segmentation data, depth data and LiDAR data could also be exported. Such training data could be used in many different machine learning tasks beyond object detection.

As discussed above in section 4.1 the average number of objects in the DeepGTAV images is still quite low compared to that of VisDrone. To counteract this the game files could further be modified to allow the spawning of more objects. Additional modifications would have to be made to increase the game memory and heap to allow stable performance. Also, by modifying the game files as discussed above the class distributions could be balanced better, resulting in better training performances and generalization capabilities.

The in-game agents in GTAV already have different states (e.g. sitting, running, fleeing, attacking, etc.). Those states could be used to also extract the activities of the agents. Combined with video capturing this could be used to produce a dataset similar to [OHP+11] to train a detector for different events (e.g. car crashes) or different activities (e.g. leaving a car).

GTAV has a vast modding community of users which produce custom modifications. Of particular interest in this context are graphics modifications and improvements. With readily available modifications the visual fidelity and realism of the generated images can be greatly improved, only requiring computational resources. Furthermore, modifications exist that add new objects (e.g. more cars, new pedestrians) to the game world. This could be used to produce even more diverse training data. A proof of concept for the possibility of modifying the game in this way can already be seen in the improved datasets and training performances from the modifications discussed in section 2.1.2.

With some modifications objects could be spawned in different contexts (e.g. people in the water), allowing the creation of entirely new datasets for vastly different domains.

5. Conclusion

This work produced further evidence for the usefulness of synthetic training data in object detection training. In particular, it has been shown that this approach yields good results in the new domain of object detection from aerial perspective. The usefulness of modifying current computer games to produce this synthetic data has been presented and the DeepGTAV framework was adapted for this use.

It must be stressed again, that when faced with the task of building an object detector for a specific context, instead of expensively capturing real-world imaging and undertaking labeling, one could also adapt such virtual environment for the task and produce training data at a fraction of the cost.

The data generation process with DeepGTAV does still have some problems, as presented in section 3.1.1. Those weaknesses mainly come from errors, bugs, or unfavorable implementations in the DeepGTAV code and can be mitigated with some programming efforts.

Nonetheless, it has been shown that it is promising to pursue the further development of the DeepGTAV framework to allow a cheap acquisition of training data for many different domains. Different venues for further research have been presented in section 4.2 and section 4.5.

A. List of Abbreviations

AI	Artificial Intelligence
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GTAV	Grand Theft Auto 5 - Computer Game
IoU	Intersection over Union
JSON	JavaScript Object Notation
LiDAR	Light Detection and Ranging
mAP	Mean Average Precision
SGD	Stochastic Gradient Descent
TCP	Transmission Control Protocol
UAV	Unmanned Aerial Vehicle
XML	Extensible Markup Language
YOLO	You Only Look Once - Object Detector

Bibliography

[AEK+18] Matt Angus, Mohamed ElBalkini, Samin Khan, Ali Harakeh, Oles Andrienko, Cody Reading, Steven Waslander, and Krzysztof Czarnecki. Unlimited road-scene synthetic annotation (URSA) dataset. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 985–992. IEEE, 2018.

[BCDH10] Alexandre Benoit, Alice Caplier, Barthélémy Durette, and Jeanny Hérault. Using human visual system modeling for bio-inspired low level image processing. *Computer vision and Image understanding*, 114(7):758–773, 2010.

[BK20] Ilker Bozcan and Erdal Kayacan. AU-AIR: A Multi-modal Unmanned Aerial Vehicle Dataset for Low Altitude Traffic Surveillance. *arXiv preprint arXiv:2001.11737*, 2020.

[BWL20] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv preprint arXiv:2004.10934*, 2020.

[CLS+18] Yuhua Chen, Wen Li, Christos Sakaridis, Dengxin Dai, and Luc Van Gool. Domain adaptive Faster R-CNN for object detection in the wild. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3339–3348, 2018.

[CV18] Zhaowei Cai and Nuno Vasconcelos. Cascade R-CNN: Delving into high quality object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6154–6162, 2018.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[DQY+18] Dawei Du, Yuankai Qi, Hongyang Yu, Yifan Yang, Kaiwen Duan, Guorong Li, Weigang Zhang, Qingming Huang, and Qi Tian. The unmanned aerial vehicle benchmark: Object detection and tracking. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 370–386, 2018.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[EVGW+10] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (VOC) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[GDDM14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[GLU12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[HCW19] Braden Hurl, Krzysztof Czarnecki, and Steven Waslander. Precise synthetic image and LiDAR (PreSIL) dataset for autonomous vehicle perception. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2522–2529. IEEE, 2019.

[HKP+16] Wei Han, Pooya Khorrami, Tom Le Paine, Prajit Ramachandran, Mohammad Babaeizadeh, Honghui Shi, Jianan Li, Shuicheng Yan, and Thomas S Huang. Seq-NMS for video object detection. *arXiv preprint arXiv:1602.08465*, 2016.

[HYKL04] Kaizhu Huang, Haiqin Yang, Irwin King, and Michael R Lyu. Learning classifiers from imbalanced data based on biased minimax probability machine. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 2, pages II–II. IEEE, 2004.

[JCH+20] Glenn Jocher, Liu Changyu, Adam Hogan, Lijun Yu, changyu98, Prashant Rai, and Trevor Sullivan. ultralytics/yolov5: Initial Release. June 2020.

[JRBM+16] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks? *arXiv preprint arXiv:1610.01983*, 2016.

[KF18] Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. *arXiv preprint arXiv:1803.00942*, 2018.

[KLY+17] Kai Kang, Hongsheng Li, Junjie Yan, Xingyu Zeng, Bin Yang, Tong Xiao, Cong Zhang, Zhe Wang, Ruohui Wang, Xiaogang Wang, et al. T-CNN: Tubelets with convolutional neural networks for object detection from videos. *IEEE Transactions on Circuits and Systems for Video Technology*, 28(10):2896–2907, 2017.

[KOLW16] Kai Kang, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. Object detection from video tubelets with convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 817–825, 2016.

[KPL+19] Amlan Kar, Aayush Prakash, Ming-Yu Liu, Eric Cameracci, Justin Yuan, Matt Rusiniak, David Acuna, Antonio Torralba, and Sanja Fidler. Meta-Sim: Learning to generate synthetic datasets. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4551–4560, 2019.

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[MNA+17] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[MSF+17] Mark Martinez, Chawin Sitawarin, Kevin Finch, Lennart Meincke, Alex Yablonski, and Alain Kornhauser. Beyond grand theft auto V for training, testing and enhancing deep learning in self driving cars. *arXiv preprint arXiv:1712.01397*, 2017.

[OHP+11] Sangmin Oh, Anthony Hoogs, Amitha Perera, Naresh Cuntoor, Chia-Chih Chen, Jong Taek Lee, Saurajit Mukherjee, JK Aggarwal, Hyungtae Lee, Larry Davis, Eran Swears, Xiaoyang Wang, Qiang Ji, Kishore Reddy, Mubarak Shah, Carl Vondrick, Hamed Pirsiavash, Deva Ramanan, Jenny Yuen, Antonio Torralba, Bi Song, Anesco Fong, Amit Roy-Chowdhury, and Mita Desai. A large-scale benchmark dataset for event recognition in surveillance video. In *CVPR 2011*, pages 3153–3160. IEEE, 2011.

[Pai19] Dheeraj Reddy Pailla. VisDrone-DET2019: The Vision Meets Drone Object Detection in Image Challenge Results. 2019.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[RF17] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[RF18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[RHGS15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[RHK17] Stephan R Richter, Zeeshan Hayder, and Vladlen Koltun. Playing for benchmarks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2213–2222, 2017.

[RSAS16] Alexandre Robicquet, Amir Sadeghian, Alexandre Alahi, and Silvio Savarese. Learning social etiquette: Human trajectory understanding in crowded scenes. In *European conference on computer vision*, pages 549–565. Springer, 2016.

[RVRK16] Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European conference on computer vision*, pages 102–118. Springer, 2016.

[SDLK18] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer, 2018.

[Set09] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[SK19] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

[SKW06] Yanmin Sun, Mohamed S Kamel, and Yang Wang. Boosting for learning multiple classes with imbalanced class distribution. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 592–602. IEEE, 2006.

[SMDH13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[SSDS12] Henry Schäfer, Jochen Süßmuth, Cornelia Denk, and Marc Stamminger. Memory efficient light baking. *Computers & Graphics*, 36(3):193–200, 2012.

[Ver17] Geert J Verhoeven. Computer graphics meets image fusion: The power of texture baking to simultaneously visualise 3d surface features and colour. *ISPRS Annals of Photogrammetry, Remote Sensing & Spatial Information Sciences*, 4, 2017.

[VRD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[Wat89] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[YWS+18] Xiangyu Yue, Bichen Wu, Sanjit A Seshia, Kurt Keutzer, and Alberto L Sangiovanni-Vincentelli. A LiDAR point cloud generator: from a virtual world to autonomous driving. In *Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval*, pages 458–464, 2018.

[ZDYW18] Xizhou Zhu, Jifeng Dai, Lu Yuan, and Yichen Wei. Towards high performance video object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7210–7218, 2018.

[ZF14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[ZWB+18] Pengfei Zhu, Longyin Wen, Xiao Bian, Ling Haibin, and Qinghua Hu. Vision Meets Drones: A Challenge. *arXiv preprint arXiv:1804.07437*, 2018.

[ZWD+17] Xizhou Zhu, Yujie Wang, Jifeng Dai, Lu Yuan, and Yichen Wei. Flow-guided feature aggregation for video object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 408–417, 2017.

[ZWD+20] Pengfei Zhu, Longyin Wen, Dawei Du, Xiao Bian, Qinghua Hu, and Haibin Ling. Vision Meets Drones: Past, Present and Future. *arXiv preprint arXiv:2001.06303*, 2020.

[ZXD+17] Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. Deep feature flow for video recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2349–2358, 2017.

[ZZXW19] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 30(11):3212–3232, 2019.