Deep learning project documentation - Advanced traffic light detection

Course: Autonomous Systems - Deep Learning
Participant: Tim Dettling
Student-ID: 208453
Subject: Advanced traffic light detection
Required_packages_additional_to_dl:

- **cv2:** 4.5.2 (OpenCV)
- **future:** 0.18.2
- **jsonschema:** 3.2.0
- **numpy:** 1.19.5
- **object-detection:** 0.1 (Tensorflow object detection API)
- **pandas:** 1.2.4
- **Pillow:** 8.2.0
- **texttable:** 1.6.3
- **tensorflow:** 2.5.0 (with all dependencies such as tensorboard)
- **tkinter:** 8.6
- **tqdm:** 4.61.0

Additionally, the complete list of modules included in the virtual environment unsed for the project and notebook on hand can be seen in the file `VirtualCondaEnvironment.txt` within this folder. As discussed with Andreas Schneider, the conda environment `dl2` is set up to run this notebook on the dgx station in the HHN lab already. Please activate using `conda activate dl2` .

1 - Introduction and project scope

In recent years, the automotive industry is dominated by the megatrends "Electrification" and "Autonomous Driving". For the megatrend Autonomous Driving, an exact perception and understanding of the environment around the vehicle is needed. Additional to the representation of other traffic participants as vehicles, pedestrians and bicycles, the perception of stationary traffic regulations such as traffic lights and traffic signs is crucial, as the autonomous vehicle must follow the regulations in any given traffic situation. Especially, in the transition phase, in which autonomous vehicles share the road with traditional, human driven vehicles, the compliance and safety of autonomous vehicles is crucial for the acceptance of the new technology.

To percept traffic lights, typically camera based systems are used. While the classification and object detection can be realized with traditional computer vision technology, the growing complexity of driving situations as well as the increasing performance of neural networks pushes the development towards deep learning based object detection and classification systems. However, due to the variety of real life image data induced by weather, illumination and the variety of traffic lights with their respective pictograms and states, this task is not trivial and shall therefore be adressed by deep learning within the project on hand.

The project on hand shall furtheron be integrated in the autonomous reasearch vehicle of the University of applied Sciences Heilbronn (HHN). The vehicle is a 2019 Passat B8, which is equiped with three radars, several ultrasonics and a monocamera in its series-configuration. Additional to the series sensorset, the vehicle is retrofit with three Lidars, several cameras and several computers. The vehicle is shown in the images below.


Left source:
https://cdn0.scrvt.com/5b9bbd140a15e188780a6244ebe572d4/c7a4c842e80f9a07/ff8932c55ee4/v/57d91179959 (https://cdn0.scrvt.com/5b9bbd140a15e188780a6244ebe572d4/c7a4c842e80f9a07/ff8932c55ee4/v/57d9117995 (03.04.2021)
Right source:
https://cdn0.scrvt.com/5b9bbd140a15e188780a6244ebe572d4/0b94db604fca2427/a38ce10ad85c/v/d9fae48fbce (https://cdn0.scrvt.com/5b9bbd140a15e188780a6244ebe572d4/0b94db604fca2427/a38ce10ad85c/v/d9fae48fbc (03.04.2021)

The integration of the traffic light detection into an existing ROS-based autonomous driving framework has the long-term goal to realize new autonomous driving functions such as stopping at red lights with predictive, smooth and energy efficient breaking towards distant red traffic lights or autonomous drive-off as soon as the relevant the traffic light switches to green again. The project on hand shall not only detect vehicle relevant traffic lights but also traffic lights relevant to pedestrians and cyclists to enhance the interpretation of driving situations containing vulnerable road users and enable safety functions for e.g. pedestrians.

Finally, the overall goal is to detect traffic lights and traffic signs from one single network. For this, the deep learning project of Nico Hessenthaler shall be merged with this project. More information on the merging strategy and result is given briefly at the end of this notebook. With the resulting network, the functions mentioned above can be extended further, for example stopping the autonomous vehicle at "STOP" signs based on the additional detections of traffic signs.

2 - Datasets

As the given task is a object detection task, generating labeled data is a very cumbersome process, as there are generally multiple objects within one image and not only the classification, but also the position and size of the object needs to be specified. Hence, the generation of labeled data cannot be managed within the project solely. Thererfore, an hybrid approach is used to obtain a sufficient amount of training, evaluation and test data. The data is mostly taken from a publically available dataset, which is further described in **section 2.1**. Furthermore, to create robustness and provide training data from the target hardware, a small amount of recorded images from the HHN Passat was labeled manually. This manually labeled data is mainly used in the cross-project work described in **chapter 11**.

2.1 - DriveU Traffic light dataset v2.0 (DLTD)

2.1.1 General

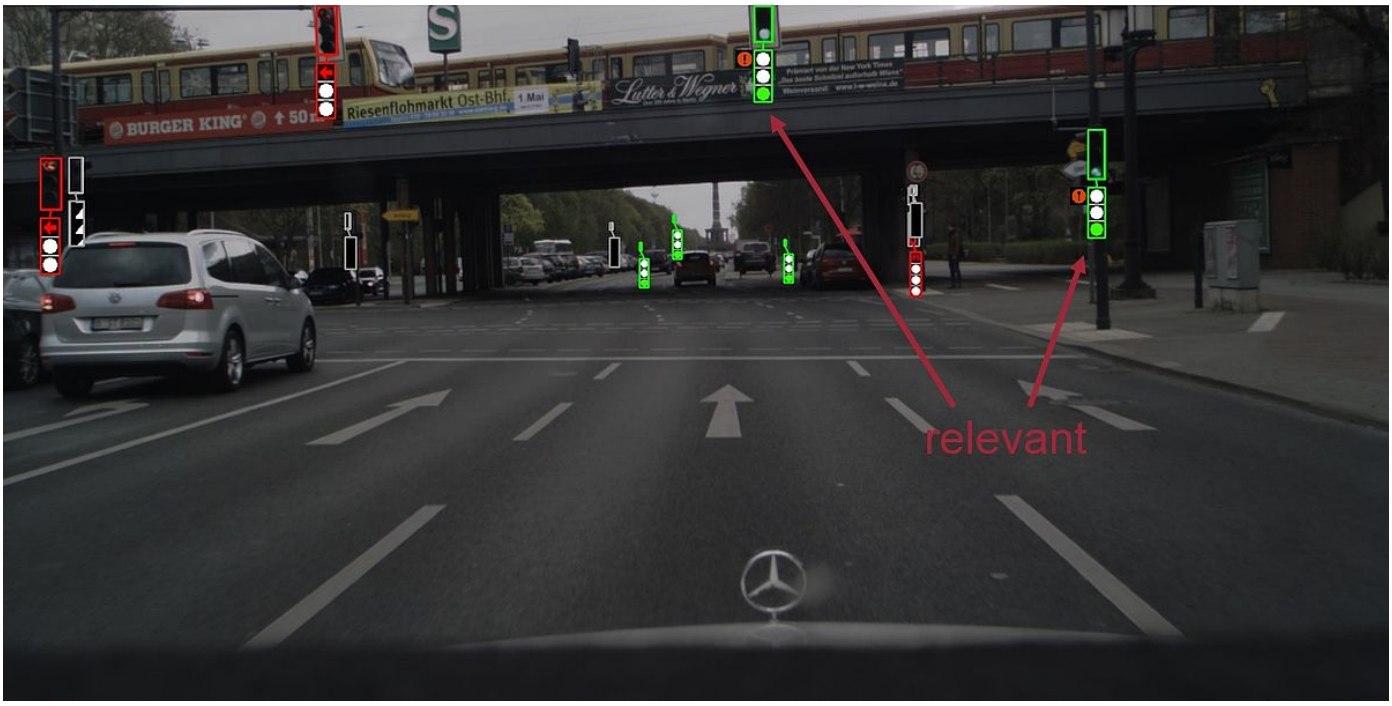
> **Remark:** The DLTD dataset can be downloaded for free in for educational / research purposes only. You need to create an account in order to get access to the data at the official website: https://www.uni-ulm.de/en/in/driveu/projects/driveu-traffic-light-dataset/ (https://www.uni-ulm.de/en/in/driveu/projects/driveu-traffic-light-dataset/)

The DriveU Traffic Light Dataset (DLTD) is a dataset of german traffic lights which is originally provided by the University of Ulm in cooperation with Daimler. It consists of data from **10 german cities** and contains **~48000 images** of **2048x1024 resolution** with **~292000 annotated traffic lights** in the latest released version label version (v2.0). Furthermore, the dataset provides a sophisticated label concept, providing **620 classes** of traffic lights. Please refer to **section 2.1.2** for details on the label concept. In the graphic below, a comparision with other popular traffic light datasets shows that the DLTD dataset is by far the biggest and most variant dataset available.

	LARA	LISA	BSTLD	DriveU (v1.0)	DriveU (v2.0)
Resolution [WxH]	640x480	1280x960	1280x720	2048x1024	2048x1024
Depth [bit]	8	8	8, 12	8, 16	8, 16
Frame Rate [Hz]	25	16	?	15	15
Annotations	9 168	51 826	24 242	232 039	292 245
Cities	1	1	1	10	10
Disparity data	✗	✗	✗	✓	✓
Classes	4	7	15, 4	423	620

Source: https://www.uni-ulm.de/fileadmin/_processed_/b/7/csm_dtld_table_6daefff493.png (https://www.uni-ulm.de/fileadmin/_processed_/b/7/csm_dtld_table_6daefff493.png) (29.06.2021)

The images of the dataset are recorded from a stereo camera. For each recorded scene, a disparity image and a single image from the left camera is provided. In the picture below, a example image with the visualization of the annotated traffic lights is provided. More information on the label structure is provided in **chapter 2.1.2**.



Source: https://www.uni-ulm.de/fileadmin/_processed_/d/9/csm_relevance_280e6943af.jpg (https://www.uni-ulm.de/fileadmin/_processed_/d/9/csm_relevance_280e6943af.jpg) (29.06.2021)

The data is stored in a location-based structure at the top level.. Below this structure, each city folder is divided into several routes, which are furtheron split into folders representing the subsequent intersections on the route. These folders then contain the images of the scenes. Each scene consists of the image of the left camera and the relating disparity image, both provided in BAYER format as a `.tiff` file. The following graphic shows the structure of the complete DLTD data package.

```
├── DTLD
    ├── Berlin           # Contains all Routes of Berlin
        ├── Berlin1           # First route
            ├── 2015-04-17_10-50-05     # First intersection
                ├── Image_1_k0.tiff        # First left camera image
                ├── Image_1_nativeV2.tiff  # First disparity image
                ├── Image_2_k0.tiff        # Second left camera image
                ├── Image_2_nativeV2       # Second disparity image
                ├── ...
            ├── 2015-04-17_10-50-41     # Second intersection
            ├── ...
        ├── Berlin2           # Second route
        ├── Berlin3           # Third route
        ├── ...
    ├── Bochum           # Contains all routes of Bochum
    ├── Bremen           # Contains all routes of Bremen
    ├── Dortmund         # Contains all routes of Dortmund
    ├── Duesseldorf      # Contains all routes of Duesseldorf
    ├── Essen            # Contains all routes of Essen
    ├── Frankfurt        # Contains all routes of Frankfurt
    ├── Fulda            # Contains all routes of Fulda
    ├── Hannover         # Contains all routes of Hannover
```

```
├── Kassel           # Contains all routes of Kassel
├── Koeln            # Contains all routes of Cologne
├── DTLD_labels_v1.0 # Old labels (v1.0) in yml-format
├── DTLD_labels_v2.0 # New labels (v2.0) in json-format
├── LICENSE          # License
└── README.md        # Readme
```

Source: https://github.com/julimueller/dtld_parsing (https://github.com/julimueller/dtld_parsing) (29.06.2021)

2.1.2 Label structure

Since the release v2.0 of the DLTD dataset, labels are given in a `.json` format. The label structure follows the above mentioned location-based structure of the dataset. For each of the 10 cities, a single `.json` file containing all labels of all images within the city folder is provided.

Furthermore, three additional `.json` files are included, which represent the suggested data split and the entity of the dataset:

- `DLTD_all.json`
- `DLTD_train.json`
- `DLTD_test.json`

It is worth noting, that the dataset is split only in two partitions, `train` and `test`, whereas for training, evaluation and test purposes, three partitions are needed, `train`, `eval` and `test`. Please refer to **chapter 3.1.2** on how the provided split is further adjusted to cope with this problem. The following section show the folder and file structure of the labels in detail:

```
├── DTLD_labels_v2.0      # New labels (v2.0) in json-format
    ├── DLTD_all.json     # Contains all labels of the dataset
    ├── DLTD_train.json   # Contains all labels of the train data
    ├── DLTD_test.json    # Contains all labels of the test data
    ├── Bochum.json       # Contains all labels of Bochum
    ├── Bochum.json       # Contains all labels of Bochum
    ├── Bremen.json       # Contains all labels of Bremen
    ├── Dortmund.json     # Contains all labels of Dortmund
    ├── Duesseldorf.json  # Contains all labels of Duesseldorf
    ├── Essen.json        # Contains all labels of Essen
    ├── Frankfurt.json    # Contains all labels of Frankfurt
    ├── Fulda.json        # Contains all labels of Fulda
    ├── Hannover.json     # Contains all labels of Hannover
    ├── Kassel.json       # Contains all labels of Kassel
    └── Koeln.json        # Contains all labels of Cologne
```

Source: https://github.com/julimueller/dtld_parsing (https://github.com/julimueller/dtld_parsing) (29.06.2021)

Generally, each of the provided `.json` files has a very similar internal structure, which will be discussed in the following section. The following code shows a customly created example of a `.json` label structure containing only one image and one object within this image. The specific attributes will be described in detail below.

only one image and one object within this image. The specific attributes will be described in detail below.


```
{
    "images": [
        {
            "disparity_image_path": "./Berlin/Berlin1/2015-04-17_10-50-05/example_
pic_nativeV2.tiff",
            "image_path": "./Berlin/Berlin1/2015-04-17_10-50-05/example_pic_k0.tif
f",
            "latitude": 52.5116,
            "longitude": 13.3077,
            "time_stamp": 1429260613.633939,
            "velocity": 11.0,
            "yaw_rate": -0.0001745,
            "labels": [
                {
                    "attributes": {
                        "aspects": "three_aspects",
                        "direction": "front",
                        "occlusion": "not_occluded",
                        "orientation": "vertical",
                        "pictogram": "circle",
                        "reflection": "not_reflected",
                        "relevance": "relevant",
                        "state": "red"
                    },
                    "track_id": "Trafficlight_1",
                    "unique_id": 1,
                    "w": 9,
                    "h": 24,
                    "x": 1254,
                    "y": 217
                }
            ]
        }
     ]
}
```


The outer structure `images` contains the list of images, whereas the image of the left camera and the corresponding disparity image are treated as one image and are therefore represented in a single structure. For each image item in the `images` list, general data is provided, containing both the relative paths to the refered images:

- disparity_image_path
- image_path

Secondly, information on the host vehicle movement and global position is provided:

- latitude
- longitude
- time_stamp
- velocity

- yaw_rate

The inner structure `labels` contains a variable amount of labeled objects within the image with their fixed attributes. Here, first the bounding box pixel size and pixel position is provided by the following attributes:

- w
- h
- x
- y

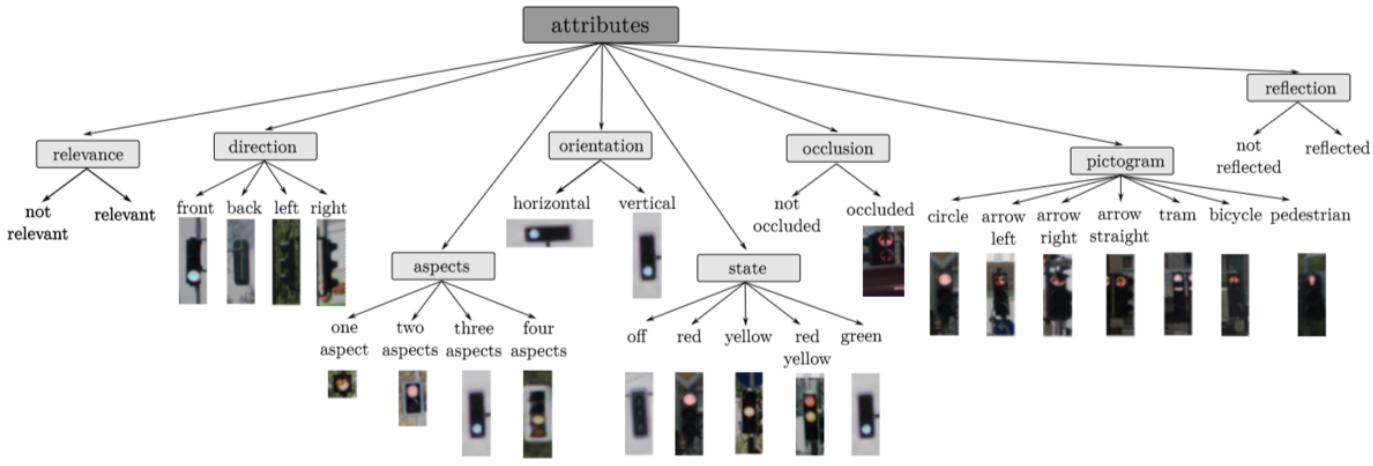
Furtheron, a tracking ID and unique ID is assigned to each object. These attributes are intended to track objects over several subsequent images. However as this task is not within the scope of the project on hand, these attributes are not actively used here. For the sake of completness, the refered attributes are:

- track_id
- unique_id

Finally, the inner structure `attributes` provides information on the object classification. Other than usual label strategies, the dataset does not provide a class string naturally but provides **8 different class attributes for each object** which are labeled seperately. The attributes are the following:

- relevance
- direction
- aspects
- orientation
- state
- occlusion
- pictogram
- reflection

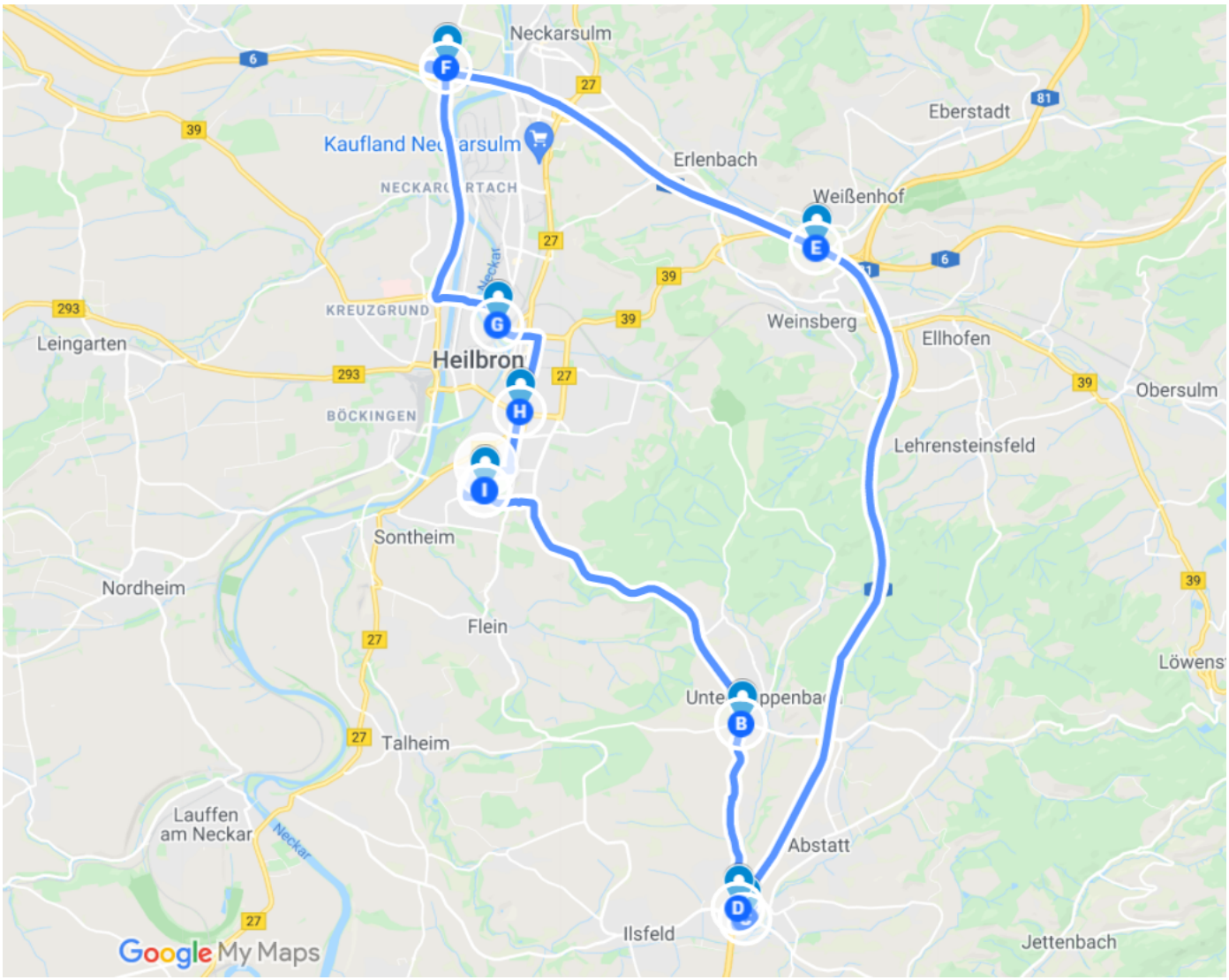
While this structure is very detailed and allows the user to select the desired objects on very specific level, it is not usable out of the box. The final target classes need to be defined and build up from the provided attributes for the project on hand. lease refer to **chapter 3.1.1** on more details, how the relevant classes are parsed from the provided attributes. The possible values for each of the eight attributes are given in the graphic below.



Source: https://www.uni-ulm.de/fileadmin/_processed_/d/a/csm_dtld_attributes_17605fc9d1.png (https://www.uni-ulm.de/fileadmin/_processed_/d/a/csm_dtld_attributes_17605fc9d1.png) (29.06.2021)

2.2. Manually labeled data from HHN passat

The long term goal of the project on hand is to use the network predictions to realize advanced autonomous driving functions in the HHN Passat described in **chapter 1**. As the vehicle is equiped with different camera hardware as the vehicle used to generate the DLTD dataset, data from the target hardware shall also be included in the training process. For this, a one-hour route around Heilbronn was driven and an image of the **120° front camera** was taken every **10 seconds**. Below, the reference route is depicted. The manually labeled is only used in the cross-project task described in **chapter 11**.



Source: own image, Google Maps

3 - Dataset preprocessing

3.1 DLTD

3.1.1 Preprocess image data

The images of the DLTD dataset are originally provided in **BAYER** format and in `.tiff` file format. Hence, they need to be preprocessed and converted to `.jpg` in order to work with them conveniently.

> **Remark:** The DLTD dataset itself is large and cannot be provided with this notebook. All operations within this notebook will therefore be performed on a reduced portion of the dataset, labeled in the provided file `./DemoConfig/DLTD_demo.json` and manually created for this notebook. The raw image data of the demo partition is located in `./DLTD/Berlin/` and stored in the original DLTD file structure. Please refer to the official website given in **chapter 2** to access the complete dataset.

> **Remark: It is recommended to take a look into `./DLTD/` and `./DLTD/Berlin/Berlin1/2015-04-17_10-50-05/` at this point, as additional content will be created within these folders during the execution of this notebook.**

Initially, some libraries need to be imported. Please refer to the additional required libs to the lecture environment `dl` or the environment packages list at the very top of this notebook.

In [2]:

```
#!interpreter
# -*- coding: utf-8 -*-

"""
Tool to convert DLTD pictures originally provided in BAYER format to BGR.
- Confidential -
"""

# ######### Built-in/Generic imports ##########
import json
import os
import copy
import csv
import cv2
import numpy as np
import tkinter as tk
from tkinter import filedialog
from matplotlib import pyplot as plt

# ######### Project specific imports ##########

__author__      =     'Tim Dettling'
__copyright__   =     'Copyright 2021, Advanced Traffic Light Detection Deep Learning'
__credits__     =     ['-']
__license__     =     'Confidential'
__version__     =     '1.0.0'
__maintainer__  =     'Tim Dettling'
__email__       =     'tdettling@stud.hs-heilbronn.de'
__status__      =     'Development'
```

Furtheron, the path containing the images is defined.

In [3]:

```
# ------------------------------------------------------------------------------------------
# Define path containing the images
# ------------------------------------------------------------------------------------------

# The relative path to the folder containing the images
imageDirectory = "./DLTD/Berlin/"
```

Now, the conversion algorithm can be started. Within the ***imageDirectory***, all `.tiff` files are read in, converted from **BAYER** to **BGR** format and then saved as a `.jpg` with the same name as the input file.

> **Remark:** The general conversion between **BAYER** and **BGR** format is specified in a supporting repository provided with the DLTD dataset. Smaller code samples are taken from there and marked within the code below. Please refer to the source below for details.

Source: https://github.com/julimueller/dtld_parsing (https://github.com/julimueller/dtld_parsing) (29.06.2021)

> **Remark:** As the images are later on used in context of the Tensorflow object detection API, which is normalizing the images natively while processing, no further normalization is applied to the images.

In [12]:

```
# Matlotlib inline for plotting the pictures
%matplotlib inline

# ------------------------------------------------------------------------------------------
# Walk the files and see if there is a .tiff image
# ------------------------------------------------------------------------------------------

# Create a list of images for plotting
images = []

# Walk over the files in the folder
for structure in os.walk(imageDirectory):

    # Loop over all files in substructure
    for file in structure[2]:

        # Check if the file is a .tiff and does not conatin "native"
        # This way, disparity images are ignored as they contain "native"
        if file.endswith(".tiff") and "native" not in file:

            # Set up the image paths
            filePathTiff = structure[0] + "/" + file
            filePathJpg = filePathTiff.replace(".tiff", ".jpg")

            # ************************************
            # Conversion code provided by DLTD.
            # Source: https://github.com/julimueller/dtld_parsing

            # Load the image unchanged
            img = cv2.imread(filePathTiff, cv2.IMREAD_UNCHANGED)

            # Convert from BAYER to BGR
            img = cv2.cvtColor(img, cv2.COLOR_BAYER_GB2BGR)

            # Images are saved in 12 bit raw -> shift 4 bits
            img = np.right_shift(img, 4)
            img = img.astype(np.uint8)

            #************************************

            # Save the image for plotting
            images.append(copy.deepcopy(img))

            # Save the original image
            cv2.imwrite(filePathJpg, img)

# Visualize the first image
plt.imshow(cv2.cvtColor(images[0], cv2.COLOR_BGR2RGB))
```

Out[12]:

```
<matplotlib.image.AxesImage at 0x7f40d11486d0>
```

After this step, all image data is avaliable as `.jpg` in the `./DLTD/Berlin/Berlin1/2015-04-17_10-50-05/` specified above. Above you can see a fully preprocessed example image.

3.1.2 Map classes

As stated before, the class definition for each object is given by **eight different attributes** with several possible states. Officially, the dataset states to contain **620 different combinations** of these attributes (refered to as classes on the website) but the total amount of possible combinations of the eight attributes and their possible states is **19200** (this number will be shown as a output of the script lateron).

However, this high number of different classes is problematic for several reasons. First, the very specific and detailed class description leads to a low amount of labels per class. This might lead to bad detection performance on these classes due to lack of training data. Furthermore, a high number of classes is not useful in real life applications, as usually very basic information of the traffic light is needed, e.g. the current `state` (**red, yellow, green, etc.**) or the shown `pictogram` (**circle, arrow left, arrow right, etc.**). Some labeled objects in the DLTD dataset are even completely irrelevant for the application in an autonomous vehicle, for example objects with the `direction` **back, left or right**, as these traffic lights are not facing towards the ego vehicle and therefore do not carry relevant information for the driver.

Knowing this, it becomes obvious, that a mapping of the possible provided classes to the desired target classes needs to be set up. As the possible amount of classes is high and during the development, several different mappings are tested, this process must be automated. The general concept is, that the mapping is saved as a `.json` file with the following format.

```
{
    "target_class_1":
        {
            [
                "possible_class_combination_1_mapped_to_target_class_1",
                "possible_class_combination_2_mapped_to_target_class_1",
                "possible_class_combination_3_mapped_to_target_class_1",
                ...
            ]
        },
    "target_class_2":
        {
            [
                "possible_class_combination_1_mapped_to_target_class_2",
                "possible_class_combination_2_mapped_to_target_class_2",
                "possible_class_combination_3_mapped_to_target_class_2",
                ...
            ]
        },
    ...
}
```

Initially, a few libs need to be imported.

In [18]:

```
#!interpreter
# -*- coding: utf-8 -*-

"""
Tool to map provided subclass attribute combinations to target classes.
- Confidential -
"""

# ######### Built-in/Generic imports ##########
import json
import os
import itertools
import pprint
from itertools import combinations

# ######### Project specific imports ##########

__author__      =     'Tim Dettling'
__copyright__   =     'Copyright 2021, Advanced Traffic Light Detection Deep Learning'
__credits__     =     ['-']
__license__     =     'Confidential'
__version__     =     '1.0.0'
__maintainer__  =     'Tim Dettling'
__email__       =     'tdettling@stud.hs-heilbronn.de'
__status__      =     'Development'
```

In the next step, the **eight attributes** described in **chapter 2.1.2** and their possible states are set up as a dictionary.

Remark: The DLTD website mentions certains states per attribute. However, during the development of this project it became obvious, that some additional states are occuring in the dataset which are not mentioned in the official documentation. These states are marked by a comment below.

In [19]:

```
# ------------------------------------------------------------------------------------------
# Set up a dictionary containing the attributes and their possible states
# ------------------------------------------------------------------------------------------
labelStructure = {

        'relevance':
            [
                "not_relevant",
                "relevant"
            ],
        'direction':
            [
                "front",
                "back",
                "left",
                "right"
            ],
        'aspects':
            [
                "unknown",            # not mentioned in official documentation
                "one_aspect",
                "two_aspects",
                "three_aspects",
                "four_aspects"
            ],
        'orientation':
            [
                "horizontal",
                "vertical"
            ],
        'state':
            [
                "unknown",            # not mentioned in official documentation
                "off",
                "red",
                "yellow",
                "red_yellow",
                "green"
            ],
        'occlusion':
            [
                "not_occluded",
                "occluded"
            ],
        'pictogram':
            [
                "unknown",             # not mentioned in official documentation
                "circle",
                "arrow_left",
                "arrow_right",
                "arrow_straight",
                "arrow_straight_left", # not mentioned in official documentation
                "tram",
                "bicycle",
                "pedestrian",
                "pedestrian_bicycle"
            ],
        'reflection':
            [
```

```
                "not_reflected",
                "reflected"
            ]
}
```

In the next step, the actual mapping is defined. In general, the mapping is defined by three structures:

- ***irrelevantAttributes***
- ***dontCareAttributeStates***
- ***offStatePictogramMerging***

The list ***irrelevantAttributes*** contains attributes, which are completely ignored for the combination algorithm as it is irrelevant for the given application, how the state of the object for this attribute is set. The attributes coosen for this are:

- **orientation** - gives information on whether the object is oriented horizontally or vertically. Not relevant for given application.
- **aspects** - gives information on whether on the number of lights of the object (e.g. a single yellow light or a typical three-lighted traffic light). Not relevant for given application, as the traffic lights relevance is usually independant of the number of aspects.
- **relevance** - gives information on whether the object is relevant for the ego vehicles lane. This attribute is interesting, but as the focus of the project is the detection and classification and not the interpretation of the driving situation, this attribute is ignored.

The list ***dontCareAttributeStates*** contains states of attributes, which classify the object as irrelevant for the given application. The classes containing the keywords are mapped to a `dontCare` class, which is later sorted out of the dataset for training, evaluation and test.

- **reflected** - object reflected e.g. in vehicle windows.
- **tram** - object is only relevant for trams.
- **occluded** - object is occluded and not completely visible.
- **back** - object is visible from the back.
- **left** - object is visible from the left.
- **right** - object is visible from the right.
- **unknown** - no information on the state given.

The dictionary ***offStatePictogramMerging*** tackles the problem, that traffic lights in the `state` "off" usually all look the same (black), the pictogram is not visible to the human eye. Therefore, a mapping algorithm was introduced to map all vehicle relevant pictograms to one class. The vehicle relevant pictograms are given in this dictionary:

- **circle**
- **arrow_right**
- **arrow_left**
- **arrow_straight**
- **arrow_straight_left**

In [20]:

```
# ------------------------------------------------------------------------------------------
# Define mapping parameters
# ------------------------------------------------------------------------------------------
irrelevantAttributes =      ["orientation", "aspects", "relevance"]
dontCareAttributeStates =   ["reflected", "tram", "occluded", "back",\
                             "left", "right", "unknown"]
offStatePictogramMerging =  {'vehicle_relevant':
                             ["circle", "arrow_right", "arrow_left",\
                              "arrow_straight", "arrow_straight_left"]}
outputDirectory =           "./DLTD/"
fileName =                  "targetClassesTrafficLights.json"
mapOffPictograms = True
```

The last preparation step is to set up a small helper function to dump a json structure to a file.

In [21]:

```
# ------------------------------------------------------------------------------------------
# Function to dump a dict data to a json file
# ------------------------------------------------------------------------------------------
def writeJsonToFile(data, filePath):

    # Write the information to the defined json file
    with open(filePath, 'w+') as outfile:
        json.dump(data, outfile)

    # Close the output file
    outfile.close()
```

Now, the main algorithm is introduced. The general approach is to first create a list of all possible classes. The possible classes hereby are combinations of all attributes and their possible states in form of concatinated strings. To set up the combinations, the python native method **itertools.product()** is used. A simple example for a possible class would be `one_aspect-front-not_occluded-horizontal-circle-not_reflected-not_relevant-red` .

Next, all desired target classes are created using the same approach, but sorting out the attributes defined in the list ***irrelevantAttributes***. The related target class to the possible class mentioned above would be `front-not_occluded-circle-not_reflected-red` .

In the next steps, the possible classes are assigned to the target classes by matching splitting the class string and comparing the substrings of the possible class with the ones of the target class.

> **Remark:** For the mapping and merging of the "off" traffic light labels (as the pictogram is not visible), a bigger code block is needed. Due to the length if the notebook, this algorithm is not described here, but is included and commented in the below code.

In [25]:

```
# Main method
if __name__ == '__main__':

    # ------------------------------------------------------------------------------------
    # STEP 1: Set up a list of all possible classes
    # ------------------------------------------------------------------------------------

    # Initially, set up all combinations
    allLabelAttributes = [labelStructure[key]for key in sorted(labelStructure.keys())]
    allLabelCombinations = list(itertools.product(*allLabelAttributes))
    allClasses = ["-".join(tup) for tup in allLabelCombinations]

    # ------------------------------------------------------------------------------------
    # STEP 2: Set up the relevant target labels
    # ------------------------------------------------------------------------------------

    # Initially, set up all combinations with respect to mapping
    targetLabelAttributes = [[item for item in labelStructure[key] \
                              if item not in dontCareAttributeStates] \
                              for key in sorted(labelStructure.keys()) \
                              if key not in irrelevantAttributes]
    targetLabelCombinations = list(itertools.product(*targetLabelAttributes))
    targetClasses = ["-".join(tup) for tup in targetLabelCombinations]

    # Switch
    if mapOffPictograms:

        # Loop over the classes and filter OFF-labels
        for idx, targetClass in enumerate(targetClasses):

            # Check if the class contains "off"
            if "off" in targetClass:

                # Loop over all head merge pictograms
                for headMergePictogram in offStatePictogramMerging.keys():

                    # Loop over all items in the head merge pictogram
                    for pictogram in offStatePictogramMerging[headMergePictogram]:

                        # Check ig the pictogram is in the class
                        if "-" + pictogram + "-" in targetClass:

                            # Remove the class from the target classes
                            targetClasses.remove(targetClass)

                            # Set up the adjusted class by replacing the
                            # pictogram with the head merge pictogram
                            targetClassAdjusted = targetClass.replace(pictogram,\
                                                                      headMergePictogram)

                            # Add the head merge class to target classes
                            # if it is not already in the list of target classes
                            if not targetClassAdjusted in targetClasses:
                                targetClasses.append(targetClassAdjusted)

    # Append the additional artificial classes
    targetClasses.append("dontCare")

    # ------------------------------------------------------------------------------------
```

```
    # STEP 3: Set up the json
    # ------------------------------------------------------------------------------------
    jsonData = {"target_classes": {targetClass: [] for targetClass in targetClasses}}


    # ------------------------------------------------------------------------------------
    # STEP 4: Map the provided labels into the target classes
    # ------------------------------------------------------------------------------------

    # Loop over all possible classes
    for possibleClass in allClasses:

        # ------------------------------------------------------------------------------------
        # STEP 4.1: Check if a dontCare Attribute is in the class string
        # ------------------------------------------------------------------------------------

        # Initially assume, that the class matches
        possibleClassIsDc = False

        # Loop over all dont care attributes
        for dcAttribute in dontCareAttributeStates:

            # Split the possible class again
            possibleClassAttributes = possibleClass.split("-")

            # Check if the dont care attribute is in the possible class string
            if dcAttribute in possibleClassAttributes:

                # Append the class to the Json data and continue to next possible class
                jsonData['target_classes']['dontCare'].append(possibleClass)
                possibleClassIsDc = True
                break

        # Continue outer loop if possible class was categorized already in this step
        if possibleClassIsDc: continue


        # ------------------------------------------------------------------------------------
        # STEP 4.2: Map relevant possible classes to the target classes
        # ------------------------------------------------------------------------------------

        # Loop over all target classes
        for targetClass in targetClasses:

            # Split the target class again
            targetClassAttributes = targetClass.split("-")
            possibleClassAttributes = possibleClass.split("-")

            # Extend the
            if mapOffPictograms:

                # Loop over all head merge pictograms
                for headMergePictogram in offStatePictogramMerging.keys():

                    # Check if the key is in the target class attributes
                    if headMergePictogram in targetClassAttributes:

                        # Loop over all items in the head merge pictogram
                        for pictogram in offStatePictogramMerging[headMergePictogram]:

                            # Perform the mapping for each
```

```
                        targetClassAttributes = targetClass.split("-")
                        targetClassAttributes.remove(headMergePictogram)
                        targetClassAttributes.append(pictogram)

                        # Initially assume, that the class matches
                        classMatches = True

                        # Loop over the targetClassAttributes
                        for attribute in targetClassAttributes:

                            # Logically AND link
                            classMatches &= (attribute in possibleClassAttributes)

                        # Check if target class matches to possible class
                        if classMatches:

                            # Append the possible class to the
                            # target class Json data and continue
                            jsonData['target_classes'][targetClass].\
                                                append(possibleClass)
                            break


                else:
                    # Initially assume, that the class matches
                    classMatches = True

                    # Loop over the targetClassAttributes
                    for attribute in targetClassAttributes:

                        # Logically AND link
                        classMatches &= (attribute in possibleClassAttributes)

                    # Check if target class matches to possible class
                    if classMatches:

                        # Append the possible class to the
                        # target class Json data and continue
                        jsonData['target_classes'][targetClass].append(possibleClass)
                        break
            else:

                # Initially assume, that the class matches
                classMatches = True

                # Loop over the targetClassAttributes
                for attribute in targetClassAttributes:

                    # Logically AND link
                    classMatches &= (attribute in possibleClassAttributes)

                # Check if target class matches to possible class
                if classMatches:

                    # Append the possible class to the target class Json data and continue
                    jsonData['target_classes'][targetClass].append(possibleClass)
                    break


    # ------------------------------------------------------------------------------
    # STEP 5: Check if the number of mapped values is matching
```

```
    # ------------------------------------------------------------------------------------

    # The completeness of mapped classes
    allMappedClasses = [item for key in jsonData['target_classes'].keys()\
                        for item in jsonData['target_classes'][key]]

    # All possible classes
    print("Number of all possible comibinations of classes:     ",\
          len(allClasses))

    # All distinct possible classes
    print("Number of distinct possible comibinations of classes:",\
          len(set(allClasses)))

    # The number of successfully mapped classes
    print("Number of successfully mapped classes:               ",\
          len(allMappedClasses))

    # The number of successfully mapped distinct classes
    print("Number of successfully mapped distinct classes:      ",\
          len(set(allMappedClasses)))

    # Number of target classes
    allTargetClasses = list(jsonData['target_classes'].keys())
    print("Number of all target classes:                        ",\
          len(allTargetClasses))

    # Number of distinct target classes
    print("Number of distinct target classes:                   ",\
          len(set(allTargetClasses)))


    # ------------------------------------------------------------------------------------
    # STEP 5: Dump the mapping to the json
    # ------------------------------------------------------------------------------------

    # Dump the target classes to json file
    writeJsonToFile(jsonData, outputDirectory + fileName)
```

```
Number of all possible comibinations of classes:      19200
Number of distinct possible comibinations of classes: 19200
Number of successfully mapped classes:                19200
Number of successfully mapped distinct classes:       19200
Number of all target classes:                         37
Number of distinct target classes:                    37
```

As you can see from the output, the complete amount of possible combinations is **19200 classes**, which are all mapped to **37 remaining target classes**. Also, all classnames are distinct, so no class name is occuring twice in the mapping structure. The output is now stored in `./DLTD/targetClassesTrafficLights.json` and can be checked as a file. To visualize the generated `.json` structure quickly, two random target classes and their mapped possible classes can be printed by acessing the generated list in python:

In [26]:

```
# ------------------------------------------------------------------------------------
# Print the target class mapping for two arbitrary classes
# ------------------------------------------------------------------------------------
print(allTargetClasses[0], ":\n-------------------------------------------")
pprint.pprint(jsonData['target_classes'][allTargetClasses[0]])
print("\n")
print(allTargetClasses[12], ":\n-----------------------------------------------------")
pprint.pprint(jsonData['target_classes'][allTargetClasses[12]])
```

```
front-not_occluded-circle-not_reflected-red :
-------------------------------------------
['one_aspect-front-not_occluded-horizontal-circle-not_reflected-not_relevant
-red',
 'one_aspect-front-not_occluded-horizontal-circle-not_reflected-relevant-re
d',
 'one_aspect-front-not_occluded-vertical-circle-not_reflected-not_relevant-r
ed',
 'one_aspect-front-not_occluded-vertical-circle-not_reflected-relevant-red',
 'two_aspects-front-not_occluded-horizontal-circle-not_reflected-not_relevan
t-red',
 'two_aspects-front-not_occluded-horizontal-circle-not_reflected-relevant-re
d',
 'two_aspects-front-not_occluded-vertical-circle-not_reflected-not_relevant-
red',
 'two_aspects-front-not_occluded-vertical-circle-not_reflected-relevant-re
d',
 'three_aspects-front-not_occluded-horizontal-circle-not_reflected-not_relev
ant-red',
 'three_aspects-front-not_occluded-horizontal-circle-not_reflected-relevant-
red',
 'three_aspects-front-not_occluded-vertical-circle-not_reflected-not_relevan
t-red',
 'three_aspects-front-not_occluded-vertical-circle-not_reflected-relevant-re
d',
 'four_aspects-front-not_occluded-horizontal-circle-not_reflected-not_releva
nt-red',
 'four_aspects-front-not_occluded-horizontal-circle-not_reflected-relevant-r
ed',
 'four_aspects-front-not_occluded-vertical-circle-not_reflected-not_relevant
-red',
 'four_aspects-front-not_occluded-vertical-circle-not_reflected-relevant-re
d']


front-not_occluded-arrow_straight-not_reflected-red :
-----------------------------------------------------
['one_aspect-front-not_occluded-horizontal-arrow_straight-not_reflected-not_
relevant-red',
 'one_aspect-front-not_occluded-horizontal-arrow_straight-not_reflected-rele
vant-red',
 'one_aspect-front-not_occluded-vertical-arrow_straight-not_reflected-not_re
levant-red',
 'one_aspect-front-not_occluded-vertical-arrow_straight-not_reflected-releva
nt-red',
 'two_aspects-front-not_occluded-horizontal-arrow_straight-not_reflected-not
_relevant-red',
 'two_aspects-front-not_occluded-horizontal-arrow_straight-not_reflected-rel
evant-red',
```

```
 'two_aspects-front-not_occluded-vertical-arrow_straight-not_reflected-not_r
elevant-red',
 'two_aspects-front-not_occluded-vertical-arrow_straight-not_reflected-relev
ant-red',
 'three_aspects-front-not_occluded-horizontal-arrow_straight-not_reflected-n
ot_relevant-red',
 'three_aspects-front-not_occluded-horizontal-arrow_straight-not_reflected-r
elevant-red',
 'three_aspects-front-not_occluded-vertical-arrow_straight-not_reflected-not
_relevant-red',
 'three_aspects-front-not_occluded-vertical-arrow_straight-not_reflected-rel
evant-red',
 'four_aspects-front-not_occluded-horizontal-arrow_straight-not_reflected-no
t_relevant-red',
 'four_aspects-front-not_occluded-horizontal-arrow_straight-not_reflected-re
levant-red',
 'four_aspects-front-not_occluded-vertical-arrow_straight-not_reflected-not_
relevant-red',
 'four_aspects-front-not_occluded-vertical-arrow_straight-not_reflected-rele
vant-red']
```

The final mapping file can now be used to generate the final inputs for training, evaluation and test.

3.1.3 Splitting the dataset

In order to train, evaluate and test the neural network training, the dataset needs to be split into three seperate partitions. The `train` part of the dataset will be used to train the network, the `eval` part will be used to monitor the training process and e.g. detect overfitting of the network. The `test` part of the set is used to test the performance of the final network at the end of the project.

As described in **chapter 2.1.2**, the DLTD dataset naturally provides a split between `train` and `test` in the files:

- `DLTD_train.json`
- `DLTD_test.json`

For the project on hand, the `DLTD_train.json` dataset will be used completely for the `train` partition. The `DLTD_test.json` will be split equally to provide both a `eval` and `test` dataset. In this step, the data will also be converted to a `.csv` format, which is needed for the Tensorflow object detection API, which will be used for the training and is described in **chapter 4**.

As the process normally involves the complete dataset, this notebook will show the general workflow based on the small portion of data from Berlin provided with this notebook. Instead of the `.json` files mentioned above, the customly created file `./DemoConfig/Berlin_demo.json` will be used.

Initially, some libs need to be imported:

In [30]:

```
#!interpreter
# -*- coding: utf-8 -*-

"""
Tool to adapt labels from DTLD dataset (traffic lights) to MTSD dataset (traffic signs)
- Confidential -
"""

# ######### Built-in/Generic imports ##########
import json
import os
import copy
import csv
import tkinter as tk
from tkinter import filedialog
import texttable

# ######### Project specific imports ##########

__author__      =     'Tim Dettling'
__copyright__   =     'Copyright 2021, Advanced Traffic Light Detection Deep Learning'
__credits__     =     ['-']
__license__     =     'Confidential'
__version__     =     '1.0.0'
__maintainer__  =     'Tim Dettling'
__email__       =     'tdettling@stud.hs-heilbronn.de'
__status__      =     'Development'
```

Initially, a few helper functions to read `.json` files and handle dictionaries are created:

In [31]:

```
# ------------------------------------------------------------------------------------------
# Function to read a json file
# ------------------------------------------------------------------------------------------
def readJsonFile(filePath):

    # Load the file
    with open(filePath) as json_file:
        labelList = json.load(json_file)

    # Return the loaded list
    return labelList

# ------------------------------------------------------------------------------------------
# Function to securely increment a int value of a specific key in a dict or create new key
# ------------------------------------------------------------------------------------------
def securelyIncrementValueInDict(dict, key):

    # Check if the key is already in the dict
    if key in dict.keys():

        # Increment value
        dict[key] += 1

    else:

        # Initialize value with 1
        dict[key] = 1
```

As described above, the data shall be represented in a `.csv` format after this step. To define the headlines and initialize the file, another helper functions is defined. The headlines within the file are:

```
- filename  file containing the labeled object
- width     width of the labeled object [pixel]
- height    height of the labeled object [pixel]
- class     class string of the labeled object (mapped)
- xmin      minimal x position of the labeled object [pixel]
- ymin      maximal x position of the labeled object [pixel]
- xmax      minimal y position of the labeled object [pixel]
- ymax      maximal y position of the labeled object [pixel]
```

In [32]:

```
# ------------------------------------------------------------------------------------------
# Function to initialize the output .csv file
# ------------------------------------------------------------------------------------------
def initCsv(filePath):

    # Open the csv file in append mode
    with open(filePath, 'w', newline='') as csvfile:

        # Set up theb writer
        writer = csv.writer(csvfile, delimiter=',',\
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
        writer.writerow(["filename", "width", "height",\
                         "class", "xmin", "ymin", "xmax", "ymax"])
```

Each labeled object is then represented as a single line in the output file. To write the objects to the file, another function is created. Other than the simple file writing, also a consistency test is performed for each label. This has been introduced, as a few mistakes in the provided DLTD labels have been found during the development. The consistency check consistis of two stages:

Initially, the **dontCare** class objects are sorted out. As stated below, this class includes objects, which are for example occluded or refleced in a window and shall therefore be excluded from training, evaluation and test.

Secondly, the dimensions of the labels are compared to the size of the image itself, as corrupted label data containing annotations outside the image dimensions is causing problems in training later.

In [34]:

```
# ------------------------------------------------------------------------------------------
# Function to write a defined json structure the output .csv file
# ------------------------------------------------------------------------------------------
def writeJsonToCsv(data, filePath):

    # Open the csv file in append mode
    with open(filePath, 'a', newline='') as csvfile:

        # Set up the writer
        writer = csv.writer(csvfile, delimiter=',', quotechar='|',\
                            quoting=csv.QUOTE_MINIMAL)

        # Write each labeled object into csv
        for obj in data['objects']:

            # Exclude "dontCare" class
            if obj['label'] != "dontCare":

                # Check if the label dimensions are ok
                if obj['bbox']['xmin'] >= 0 and \
                obj['bbox']['xmax'] <= 2048 and \
                obj['bbox']['ymin'] >= 0 and \
                obj['bbox']['ymax'] <= 1024:

                    # Write the row to the csv file
                    writer.writerow([data['imagepath'], data['width'],\
                                     data['height'], obj['label'], \
                                     obj['bbox']['xmin'], obj['bbox']['ymin'],\
                                     obj['bbox']['xmax'], obj['bbox']['ymax']])
```

Additional to the `.csv` file, a so-called labelmap file is needed in the file format `.pbtxt`. This file includes all classes along with a assigned number to each class. The labelmap file for the complete dataset can be seen here:

```
item {
    id: 1
    name: 'front-not_occluded-circle-not_reflected-yellow'
}

item {
    id: 2
    name: 'front-not_occluded-arrow_right-not_reflected-red_yellow'
}

item {
    id: 3
    name: 'front-not_occluded-arrow_straight_left-not_reflected-yellow'
}

item {
    id: 4
    name: 'front-not_occluded-arrow_straight_left-not_reflected-green'
}

item {
    id: 5
    name: 'front-not_occluded-arrow_straight-not_reflected-yellow'
}

item {
    id: 6
    name: 'front-not_occluded-bicycle-not_reflected-red_yellow'
}

item {
    id: 7
    name: 'front-not_occluded-arrow_right-not_reflected-red'
}

item {
    id: 8
    name: 'front-not_occluded-pedestrian_bicycle-not_reflected-off'
}

item {
    id: 9
    name: 'front-not_occluded-pedestrian-not_reflected-off'
}

item {
    id: 10
    name: 'front-not_occluded-circle-not_reflected-green'
}

item {
```

```
    id: 11
    name: 'front-not_occluded-bicycle-not_reflected-red'
}

item {
    id: 12
    name: 'front-not_occluded-bicycle-not_reflected-green'
}

item {
    id: 13
    name: 'front-not_occluded-pedestrian_bicycle-not_reflected-red'
}

item {
    id: 14
    name: 'front-not_occluded-pedestrian-not_reflected-green'
}

item {
    id: 15
    name: 'front-not_occluded-circle-not_reflected-red_yellow'
}

item {
    id: 16
    name: 'front-not_occluded-circle-not_reflected-red'
}

item {
    id: 17
    name: 'front-not_occluded-arrow_left-not_reflected-green'
}

item {
    id: 18
    name: 'front-not_occluded-arrow_straight-not_reflected-red_yellow'
}

item {
    id: 19
    name: 'front-not_occluded-pedestrian_bicycle-not_reflected-green'
}

item {
    id: 20
    name: 'front-not_occluded-arrow_left-not_reflected-red'
}

item {
    id: 21
    name: 'front-not_occluded-arrow_straight_left-not_reflected-red'
```

```
}

item {
    id: 22
    name: 'front-not_occluded-pedestrian_bicycle-not_reflected-yellow'
}

item {
    id: 23
    name: 'front-not_occluded-pedestrian-not_reflected-yellow'
}

item {
    id: 24
    name: 'front-not_occluded-pedestrian-not_reflected-red'
}

item {
    id: 25
    name: 'front-not_occluded-bicycle-not_reflected-off'
}

item {
    id: 26
    name: 'front-not_occluded-pedestrian_bicycle-not_reflected-red_yellow'
}

item {
    id: 27
    name: 'front-not_occluded-arrow_straight-not_reflected-red'
}

item {
    id: 28
    name: 'front-not_occluded-bicycle-not_reflected-yellow'
}

item {
    id: 29
    name: 'front-not_occluded-arrow_left-not_reflected-yellow'
}

item {
    id: 30
    name: 'front-not_occluded-arrow_straight-not_reflected-green'
}

item {
    id: 31
    name: 'front-not_occluded-arrow_right-not_reflected-green'
}
```

```
    item {
        id: 32
        name: 'front-not_occluded-arrow_right-not_reflected-yellow'
    }

    item {
        id: 33
        name: 'front-not_occluded-arrow_left-not_reflected-red_yellow'
    }

    item {
        id: 34
        name: 'front-not_occluded-vehicle_relevant-not_reflected-off'
    }

    item {
        id: 35
        name: 'front-not_occluded-pedestrian-not_reflected-red_yellow'
    }
```

To create this file, the following function helper function is created:

In [35]:

```
# ------------------------------------------------------------------------------------------
# Function to dump all labels to the labelmap file (pbtxt)
# ------------------------------------------------------------------------------------------
def writeLabelMapFile(data, filePath):

    # Write the labels to a file in pbtxt format
    f = open(filePath, "w+")

    # Write labels to .pbtxt file in correct order
    for i, lbl in enumerate(data):

        f.write("item {\n")
        f.write("\tid: {}\n".format(i+1))
        f.write("\tname: \'" + lbl +"\'\n" )
        f.write("}\n\n")

    # Close the file
    f.close()
```

The last external function is a function to parse the provided label format to a custom json format. Mainly, the mapping of the originally provided attributes of the label to the target class is executed here. For this, the function takes the class mapping, which was created previously in **chapter 3.1.2**. Other than that, all relevant attributes are parsed to the custom json format. Based on this json format, a label file per image can be exported if needed. The custom json format is originally based on the format of the **MTSD dataset** (contains traffic signs around the word) and was introduced for the cross-project work mentioned in **chapter 11** of this notebook.

Source: https://blog.mapillary.com/update/2019/06/27/mapillary-traffic-sign-dataset.html (https://blog.mapillary.com/update/2019/06/27/mapillary-traffic-sign-dataset.html) (11.07.21)

In [36]:

```
# ------------------------------------------------------------------------------------------
# Function to convert a DTLD label dict to a json format (MTSD)
# ------------------------------------------------------------------------------------------
def convertToMtsdJsonFormat(label, classMapping):

    # Derive the filename from the path
    fileName = label['image_path'].split("/")[-1][:-5] + ".json"
    filePath = label['image_path'][:-5].replace('./', './DLTD/') + ".jpg"

    # Initialize a blank label with the desired attributes
    jsonLabel = {}

    # Fill the general attributes
    jsonLabel['filename'] = fileName
    jsonLabel['imagepath'] = filePath
    jsonLabel['width'] = 2048
    jsonLabel['height'] = 1024
    jsonLabel['ispano'] = False
    jsonLabel['objects'] = []

    # Append objects to the correct format
    for labeledObject in label['labels']:

        # Set up a blank dictionary
        objectDict = {}

        # Map the required general attributes
        objectDict['key'] = labeledObject['unique_id']

        # Map the label
        classAttributes = [labeledObject['attributes'][key]\
                           for key in sorted(list(labeledObject['attributes'].keys()))]
        classString = ("-").join(classAttributes)
        headClass = {item: key for key in classMapping.keys() \
                     for item in classMapping[key]}[classString]
        objectDict['label'] = headClass

        # Map the required bounding box attributes
        objectDict['bbox'] = {}
        objectDict['bbox']['xmin'] = labeledObject['x']
        objectDict['bbox']['ymin'] = labeledObject['y']
        objectDict['bbox']['xmax'] = labeledObject['x'] + labeledObject['w']
        objectDict['bbox']['ymax'] = labeledObject['y'] + labeledObject['h']

        # Map the required properties attributes
        objectDict['properties'] = {}
        objectDict['properties']['barrier'] = False
        objectDict['properties']['occluded'] = not "not_occluded" in objectDict['label']
        objectDict['properties']['out-of-frame'] = False
        objectDict['properties']['exterior'] = False
        objectDict['properties']['ambiguous'] = False
        objectDict['properties']['included'] = False
        objectDict['properties']['direction-or-information'] = False
        objectDict['properties']['highway'] = False
        objectDict['properties']['dummy'] = False

        # Append the objectDict to the output
        jsonLabel['objects'].append(copy.deepcopy(objectDict))
```

```
    # Return the mtsd json formated data and the file name
    return jsonLabel, fileName
```

As a last step before the main algorithm, some parameter definition is needed. First, the path to the previously created class mapping file and the outputpath needs to be defined. Most importantly, also the ***mode*** must be defined. The mode defines, how

- **TRAIN** - Exports all labels in the json. Should be applied to `DLTD_train.json`
- **EVAL** - Exports first half of the labels in the json. Should be applied to `DLTD_test.json`
- **TEST** - Exports second half of the labels in the json. Should be applied to `DLTD_test.json`

This notebook does not provide the correct files `DLTD_train.json` and `DLTD_test.json`, as the data is to big for a demonstration. However, all modes work with the provided `DLTD_demo.json`. The original files can be accessed seperately to this notebook on request if needed.

> **Remark:** The following algorithm will only work if the class mapping file was created in prior. Please make sure to execute all previous cells, especially the ones in **chapter 3.1.2**.

In [58]:

```
# ----------------------------------------------------------------------------------------
# Define mapping parameters
# ----------------------------------------------------------------------------------------
classMappingFile = "./DLTD/targetClassesTrafficLights.json"
labelInputDirectory = "./DLTD/DemoConfig/"
outputDirectory = "./DLTD/"
mode = "TRAIN"                                                  # TRAIN, TEST or EVAL
```

Finally, the main algorithm can be started. All `.json` files in the labelInputDirectory are read and processed.

In [60]:

```
# Main method
if __name__ == '__main__':

    # ------------------------------------------------------------------------------------
    # STEP 1: Load the class mapping file
    # ------------------------------------------------------------------------------------

    # Read the class mapping file
    classMapping = readJsonFile(classMappingFile)

    # ------------------------------------------------------------------------------------
    # STEP 2: Loop over the json files
    # ------------------------------------------------------------------------------------

    # Initialize the csv file
    initCsv(outputDirectory + mode + ".csv")

    # Initialize a list of labels
    allLabels = []
    numLabelOccuranced = {}

    # Sort out non json files
    for idx, file in enumerate(os.listdir(labelInputDirectory)):

        # Check if the file is actually a label file (.json file)
        if file.endswith('.json'):

            # Parse the json file using the defined function
            labelList = readJsonFile(labelInputDirectory + "/" + file)

            # Loop over the label list of this file
            for idx, label in enumerate(labelList["images"]):

                # Separate test and eval
                if idx >= len(labelList["images"]) / 2 and mode == "TEST":
                    continue

                # Separate test and eval
                if idx < len(labelList["images"]) / 2 and mode == "EVAL":
                    continue

                # Convert the label to MTSD json format
                mtsdJsonLabel, fileName = convertToMtsdJsonFormat(label,\
                                                  classMapping["target_classes"])

                # Append all labels to the global list
                for obj in mtsdJsonLabel['objects']:
                    if obj['label'] != "dontCare":
                        allLabels.append(obj['label'])

                    # Increment the class count
                    securelyIncrementValueInDict(numLabelOccuranced, obj['label'])

                # Dump the label to csv file
                writeJsonToCsv(mtsdJsonLabel, outputDirectory + mode + ".csv")

    # ------------------------------------------------------------------------------------
    # STEP 3: Create the labelmap file of all seen labels
    # ------------------------------------------------------------------------------------
```

```
    # Get all distinct classes by using a set
    distinctClasses = set(allLabels)

    # Create the labelMap file
    if mode == "TRAIN":
        writeLabelMapFile(distinctClasses, outputDirectory + "/labelMap.pbtxt")

    # Generate an overview
    keysSorted = sorted(list(numLabelOccuranced.keys()))
    table = texttable.Texttable()
    table.set_cols_align(["l", "c"])
    table.set_cols_valign(["m", "m"])
    rows = [["class", "occurances"]]
    for key in keysSorted:
        rows.append([key, numLabelOccuranced[key]])
    table.add_rows(rows)
    print("Classes and respective occurances in the processed data for mode: ", mode, "\n")
    print(table.draw())
    print("\nNumber of distinct classes without dontCare:", len(distinctClasses))
```

```
Classes and respective occurances in the processed data for mode:  TRAIN

+-----------------------------------------------------------+------------+
|                          class                            | occurances |
+===========================================================+============+
| dontCare                                                  |    146     |
+-----------------------------------------------------------+------------+
| front-not_occluded-circle-not_reflected-red               |     57     |
+-----------------------------------------------------------+------------+
| front-not_occluded-pedestrian-not_reflected-off           |     11     |
+-----------------------------------------------------------+------------+
| front-not_occluded-pedestrian-not_reflected-red           |     12     |
+-----------------------------------------------------------+------------+
| front-not_occluded-vehicle_relevant-not_reflected-off     |     8      |
+-----------------------------------------------------------+------------+

Number of distinct classes without dontCare: 4
```

As a console output, the classes and their respective occurances in the processed partition of the dataset is printed. You can try to alter the ***mode*** in the parameter section and see, how the included classes and their respective occurances change for the given partition of the dataset.

As the main output, two new files have been created in `./DLTD/` :

- `labelMap_TRAIN.pbtxt`
- `TRAIN.csv`

Together with the initially provided files (which can be created the same way, altering the **mode** in the parameter section), the folder now contains all relevant data for the next steps.

- `labelMap_TRAIN.pbtxt`
- `TRAIN.csv`
- `EVAL.csv`
- `TEST.csv`

For the complete training process, the described process is applied to the complete dataset. This results in the

following numbers for the partitions:

Partition	Number of images	Number of labeled objects
TRAIN	28525	114078
EVAL	6226	24829
TEST	6226	24602

Furthermore, the occurances of classes should be equally distributed among the partitions, meaning all classes should appear in each partition weighted according to the general size proportions of the partitions. The following lists of classes and their respective occurances amoung the partitions are created by the same code as shown above for the smaller `DLTD_demo.json` partition.

It is obvious, that while the partitions are not perfectly equally distributed, the split is still relatively good for the high variety of classes and the large dataset. However, when looking at the `TRAIN` set, some classes only appear very rarely. For example, the class `front-not_occluded-arrow_straight_left-not_reflected-yellow` is only occuring two times within the `TRAIN` data. While this is caused by the natural distribution of traffic lights in real life (`yellow` light is rare and a `straight_left` arrow is even more rare), the performance on this class is still expected to be weak. Please refer to **chapter 9** for a discussion of the dataset distributions and other factors on the final performance.

TRAIN dataset:

class	occurances
dontCare	86142
front-not_occluded-arrow_left-not_reflected-green	2013
front-not_occluded-arrow_left-not_reflected-red	7353
front-not_occluded-arrow_left-not_reflected-red_yellow	194
front-not_occluded-arrow_left-not_reflected-yellow	357
front-not_occluded-arrow_right-not_reflected-green	628
front-not_occluded-arrow_right-not_reflected-red	464
front-not_occluded-arrow_right-not_reflected-red_yellow	21
front-not_occluded-arrow_right-not_reflected-yellow	136
front-not_occluded-arrow_straight-not_reflected-green	3264
front-not_occluded-arrow_straight-not_reflected-red	1771
front-not_occluded-arrow_straight-not_reflected-red_yellow	96
front-not_occluded-arrow_straight-not_reflected-yellow	181
front-not_occluded-arrow_straight_left-not_reflected-green	120
front-not_occluded-arrow_straight_left-not_reflected-red	10
front-not_occluded-arrow_straight_left-not_reflected-yellow	2
front-not_occluded-bicycle-not_reflected-green	1041
front-not_occluded-bicycle-not_reflected-off	66
front-not_occluded-bicycle-not_reflected-red	896
front-not_occluded-bicycle-not_reflected-red_yellow	73
front-not_occluded-bicycle-not_reflected-yellow	35
front-not_occluded-circle-not_reflected-green	43361
front-not_occluded-circle-not_reflected-red	23864
front-not_occluded-circle-not_reflected-red_yellow	1214

front-not_occluded-circle-not_reflected-yellow	2925
front-not_occluded-pedestrian-not_reflected-green	8807
front-not_occluded-pedestrian-not_reflected-off	607
front-not_occluded-pedestrian-not_reflected-red	6862
front-not_occluded-pedestrian-not_reflected-red_yellow	317
front-not_occluded-pedestrian-not_reflected-yellow	339
front-not_occluded-pedestrian_bicycle-not_reflected-green	887
front-not_occluded-pedestrian_bicycle-not_reflected-off	63
front-not_occluded-pedestrian_bicycle-not_reflected-red	553
front-not_occluded-pedestrian_bicycle-not_reflected-red_yellow	37
front-not_occluded-pedestrian_bicycle-not_reflected-yellow	35
front-not_occluded-vehicle_relevant-not_reflected-off	5529

EVAL dataset:

class	occurances
dontCare	23930
front-not_occluded-arrow_left-not_reflected-green	455
front-not_occluded-arrow_left-not_reflected-red	1219
front-not_occluded-arrow_left-not_reflected-red_yellow	33
front-not_occluded-arrow_left-not_reflected-yellow	150
front-not_occluded-arrow_right-not_reflected-green	80
front-not_occluded-arrow_right-not_reflected-red	206
front-not_occluded-arrow_right-not_reflected-red_yellow	1
front-not_occluded-arrow_right-not_reflected-yellow	18
front-not_occluded-arrow_straight-not_reflected-green	600
front-not_occluded-arrow_straight-not_reflected-red	142
front-not_occluded-arrow_straight-not_reflected-red_yellow	8
front-not_occluded-arrow_straight-not_reflected-yellow	35
front-not_occluded-bicycle-not_reflected-green	581
front-not_occluded-bicycle-not_reflected-red	280
front-not_occluded-bicycle-not_reflected-red_yellow	20
front-not_occluded-bicycle-not_reflected-yellow	7
front-not_occluded-circle-not_reflected-green	9982
front-not_occluded-circle-not_reflected-red	4541
front-not_occluded-circle-not_reflected-red_yellow	299
front-not_occluded-circle-not_reflected-yellow	555
front-not_occluded-pedestrian-not_reflected-green	2260
front-not_occluded-pedestrian-not_reflected-off	217
front-not_occluded-pedestrian-not_reflected-red	1366

```
| front-not_occluded-pedestrian-not_reflected-red_yellow     |     98     |
+------------------------------------------------------------+------------+
| front-not_occluded-pedestrian-not_reflected-yellow         |     94     |
+------------------------------------------------------------+------------+
| front-not_occluded-pedestrian_bicycle-not_reflected-green  |    153     |
+------------------------------------------------------------+------------+
| front-not_occluded-pedestrian_bicycle-not_reflected-red    |     29     |
+------------------------------------------------------------+------------+
| front-not_occluded-pedestrian_bicycle-not_reflected-yellow |     20     |
+------------------------------------------------------------+------------+
| front-not_occluded-vehicle_relevant-not_reflected-off      |    1389    |
+------------------------------------------------------------+------------+
```

TEST dataset:

class	occurances
dontCare	18608
front-not_occluded-arrow_left-not_reflected-green	427
front-not_occluded-arrow_left-not_reflected-red	1978
front-not_occluded-arrow_left-not_reflected-red_yellow	29
front-not_occluded-arrow_left-not_reflected-yellow	85
front-not_occluded-arrow_right-not_reflected-green	107
front-not_occluded-arrow_right-not_reflected-red	77
front-not_occluded-arrow_right-not_reflected-red_yellow	2
front-not_occluded-arrow_right-not_reflected-yellow	34
front-not_occluded-arrow_straight-not_reflected-green	851
front-not_occluded-arrow_straight-not_reflected-red	303
front-not_occluded-arrow_straight-not_reflected-red_yellow	16
front-not_occluded-arrow_straight-not_reflected-yellow	58
front-not_occluded-bicycle-not_reflected-green	137
front-not_occluded-bicycle-not_reflected-off	10
front-not_occluded-bicycle-not_reflected-red	141
front-not_occluded-bicycle-not_reflected-red_yellow	5
front-not_occluded-bicycle-not_reflected-yellow	3
front-not_occluded-circle-not_reflected-green	8932
front-not_occluded-circle-not_reflected-red	5714
front-not_occluded-circle-not_reflected-red_yellow	207
front-not_occluded-circle-not_reflected-yellow	573
front-not_occluded-pedestrian-not_reflected-green	1599
front-not_occluded-pedestrian-not_reflected-off	89

```
| front-not_occluded-pedestrian-not_reflected-red                |    1598    |
+----------------------------------------------------------------+------------+
| front-not_occluded-pedestrian-not_reflected-red_yellow         |     14     |
+----------------------------------------------------------------+------------+
| front-not_occluded-pedestrian-not_reflected-yellow             |     2      |
+----------------------------------------------------------------+------------+
| front-not_occluded-pedestrian_bicycle-not_reflected-green      |    299     |
+----------------------------------------------------------------+------------+
| front-not_occluded-pedestrian_bicycle-not_reflected-off        |     22     |
+----------------------------------------------------------------+------------+
| front-not_occluded-pedestrian_bicycle-not_reflected-red        |    107     |
+----------------------------------------------------------------+------------+
| front-not_occluded-pedestrian_bicycle-not_reflected-red_yellow |     21     |
+----------------------------------------------------------------+------------+
| front-not_occluded-vehicle_relevant-not_reflected-off          |    1169    |
+----------------------------------------------------------------+------------+
```

3.2 Data from the HHN Passat

To label the recorded images from the HHN Passat, a existing and widely used label tool "labelImg" was used. The general label process is to draw bounding boxes around the desired objects and label the object with a class definition out of a predefined list. To reduce the length of the notebook on hand, this process will not be described in detail here. More information on the cross-project work can be found in **chapter 11**.

In-depth information on how to install and use the tool can be found in the below sources:

> **Sources LabelImg:**
> - https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html#preparing-the-dataset (https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html#preparing-the-dataset) (20.07.21)
> - http://tzutalin.github.io/labelImg/ (http://tzutalin.github.io/labelImg/) (20.07.21)

The process of labeling images is depicted below. The bounding boxes can be drawn within a GUI and a class string is assigned to each bounding box. The labels can then be exported to a `.xml` file.

Source: Own creation by Tim Dettling & Nico Hessenthaler (21.07.2021)

4 - Tensorflow Object Detection API

The tensorflow object detection API is a toolset provided by Google which currently supports `tensorflow==2.5.0` in its latest release. The API is not naturally provided with the standard tensorflow installation and must therefore be installed manually on the training maschine. The API however depends on tensorflow. The installation routine can be found here:

> **Installation guide:** https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html (https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html)

The API provides methods to train, evaluate and test neural networks for object detection problems as well as pre-trained models of the Tensorflow object detection model zoo and is commonly used for object detection using tensorflow. Additionally, the API provides predefined methods for data augmentation, training configurations such as learning rate decay, dropout etc.

As the process of installation is a cumbersome task and several other setups need to be performed during the process, the installation has been performed on two seperate lab PCs of the HHN already. If needed, a meeting can be set up to show the installation, training and wvaluation process on these lab PCs, as proposed by Prof. Stache.

5 - Neural Network selection - Faster R-CNN ResNet101 V1 1024x1024

As mentioned in **chapter 4**, the Tensorflow Object Detection API provides a "Model Zoo" containing popular object detection neural network designs. The Model Zoo can be found here:

> **TF2 detection model zoo:**
> https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md
> (https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md)

The given task of detecting traffic lights consists mainly of the detection of relatively small objects. Additionally, as the network shall be included in a real time system within the HHN Passat, the networks computation speed is also a huge factor for the decision. For the project on hand, the tradeoff between a fast, real time capable network, a high performance epecially relatively small objects is relevant. Furthermore, the networks input image size must be sufficiently high, as the releatively small features on traffic sign objects would not be represented well on input images with lower resolution. For the project on hand, a input resolution of **1024x1024 pixels** was choosen, again beeing a tradeoff between computational effort and expected performance.

Comparing the popular networks within the Tensorflow Model Zoo under the mentioned assumption, the following networks show potential for usage in the given task based on their computation time and mean average Precision (mAP) on the COCO dataset, which is a commonly used reference dataset for object detection:

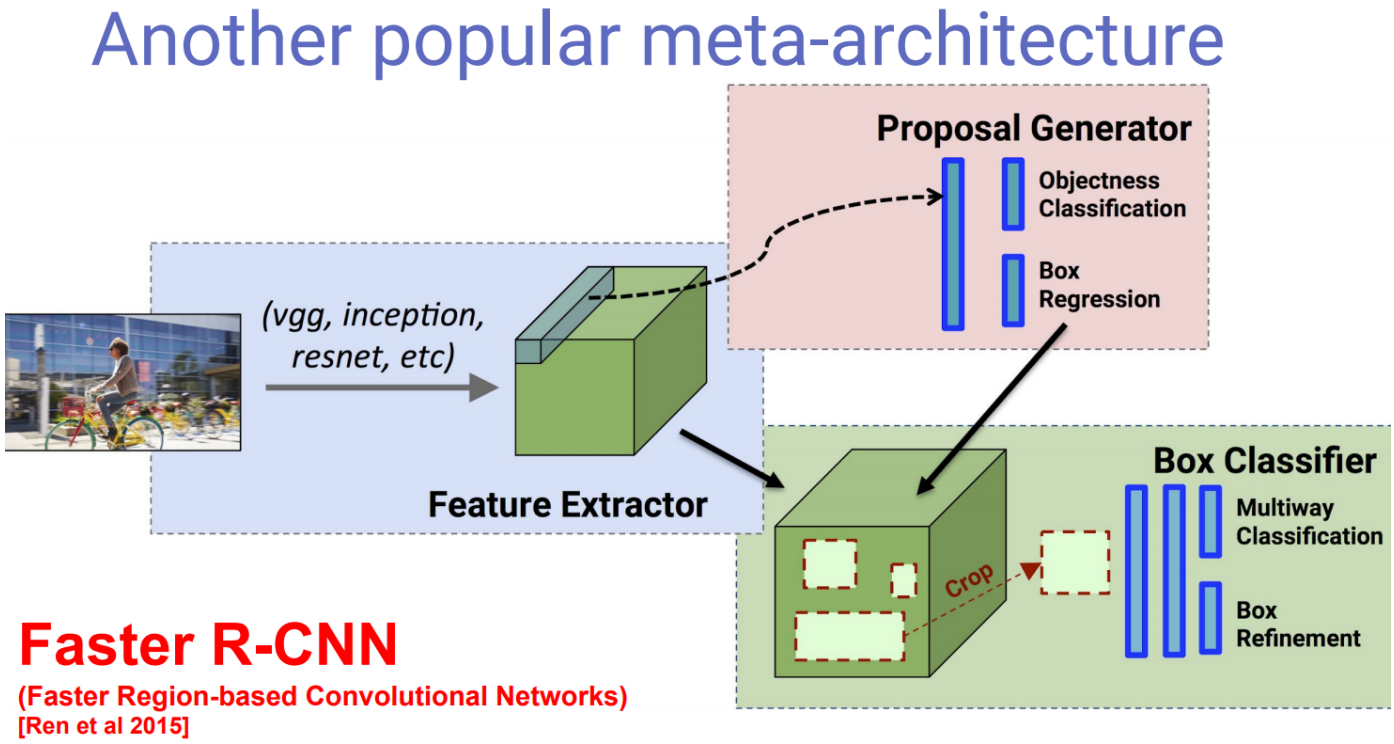
Network	Computation time (ms)	COCO mAP
SSD ResNet101 V1 FPN 1024x1024 (RetinaNet101)	104	39.5
EfficientDet D4 1024x1024	133	48.5
Faster R-CNN ResNet50 V1 1024x1024	65	31.0
Faster R-CNN ResNet152 V1 1024x1024	85	37.6
Faster R-CNN ResNet101 V1 1024x1024	**72**	**37.1**

Source:
https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md
(https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md)
[02.07.2021]

The best mAP is achieved by the **EfficientDet D4 1024x1024** network with **39.5**, but at the cost of a very high computation time with **133ms**. The best computation speed is achieved by the **Faster R-CNN ResNet50 V1 1024x1024** network, but at a comparably bad mAP value of **31.0**.

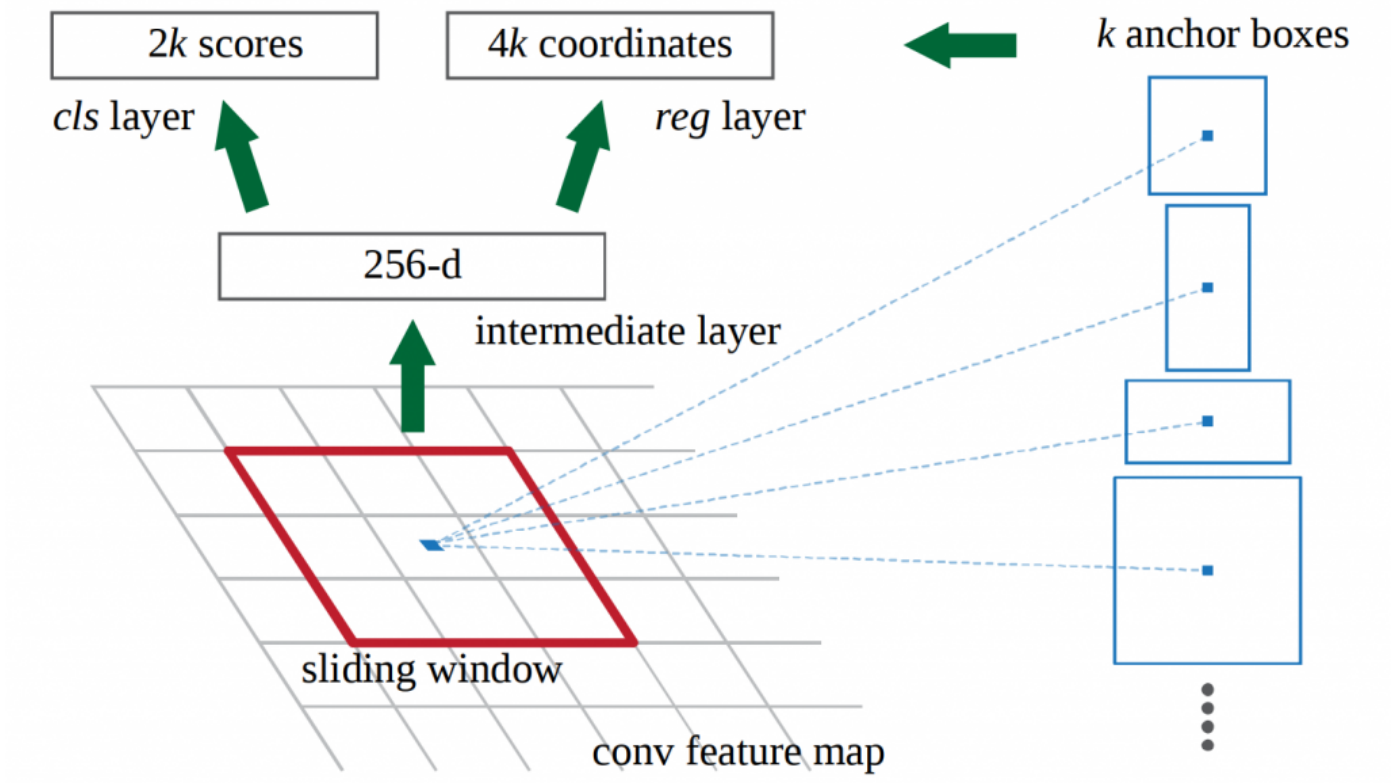
For the project on hand, the **Faster R-CNN ResNet101 V1 1024x1024** was selected, due to its relatively high speed of **72ms** and very good COCO mAP of **37.1**. The structure of the selected network will be described briefly in the following section.

The detection algorithm of Faster R-CNN networks generally consist of three main modules. First, a Feature Extractor provides features to a Proposal Generator (RPN). The output of these two modules is then used in the Box Classifier. While the Feature Extractor can be realized in multiple forms, for the used network within this project, the Feature Extractor is realized with a ResNet101. The figure below shows the general system architecture.



Source: https://clear.ml/blog/the-battle-of-speed-accuracy-single-shot-vs-two-shot-detection/ (https://clear.ml/blog/the-battle-of-speed-accuracy-single-shot-vs-two-shot-detection/) (28.07.2021)

The Proposal Generator works using a fully connected layer to predict anchor boxes with predefined scale and aspect ratios, while the number of anchor boxes can be defined by a Hyperparameter. The result after the first two computation steps (Feature Extractor and Proposal Generator) is a intermediate feature map with related box proposals, which is then together fed into the Box Classifier, as shown in the image above.



Source: https://i.stack.imgur.com/xFUUt.png (https://i.stack.imgur.com/xFUUt.png) (28.07.2021)

Within the last step, features are cropped from the intermediate feature map using the box proposals. Within the last step, box refinements and a classification is then calculated for each box proposal resulting in detection boxes with a respective classification as the output of the network.

6 - Training setup and configurations

6.1 - Creating tfrecords

In order to train the network with the tensorflow object detection API, so-called **tfrecords** are needed. A tfrecord is a tensorflow specific file format which contains binary records containing both the binary image data itself as well as the annotation data. As the images are stored within the record, it is easily understandable, that the files grow to huge sizes, especially considering the size of the DLTD dataset containing 48000 images.

To start the training, two seperate tfrecords are needed:

- `TRAIN.tfrecord`
- `EVAL.tfrecord`

For the testing of the network in the end, another tfrecord is needed:

- `TEST.tfrecord`

To generate the files, mainly the `.csv` files created in **chapter 3.1.3** are needed:

- `TRAIN.csv`
- `EVAL.csv`
- `TEST.csv`

Additional to that, the labelmap file is needed:

- `labelMap.pbtxt`

All of these files have been created within this notebook and are available in the folder `./DLTD/` . In this notebook, only a single `TRAIN.tfrecord` file will be created based on the previously created demo data, to showcase the process. By changing some parameters, also the `EVAL.tfrecord` can be created with the code of this notebook. The original `.tfrecords` used for the actual training are not included here due to their huge file size (several Gigabyte) but can be provided on request.

For the creation of the records, an existing tool was adapted. As this script is only parsing pre-existing data to a new format and does not contain large amounts of new information, the process will only be described very briefly.

> **Remark:** The converter tool is based on this public available implementation: https://gist.github.com/rotemtam/a1f92ab7f3444cf61da305539db4997b (https://gist.github.com/rotemtam/a1f92ab7f3444cf61da305539db4997b) [15.07.2021].

> **Remark:** As this process is needed for both the traffic light and the traffic sign project described in **chapter 11**, this tool is used by both projects simultaniously.

Initially, the script needs some imports:

In [42]:

```
#!interpreter
# -*- coding: utf-8 -*-

"""
Tool to convert the label format from .csv to .tfrecord format
- Confidential -
"""

# ######### Built-in/Generic imports ##########
from __future__ import division
from __future__ import print_function
from __future__ import absolute_import
import os
import io
import pandas as pd
import tensorflow as tf
from PIL import Image
from tqdm import tqdm
from object_detection.utils import dataset_util
from collections import namedtuple, OrderedDict

# ######### Project specific imports ##########

__author__      =     'Tim Dettling'
__copyright__   =     'Copyright 2021, Advanced Traffic Light Detection Deep Learning'
__credits__     =     ['-']
__license__     =     'Confidential'
__version__     =     '1.0.0'
__maintainer__  =     'Tim Dettling'
__email__       =     'tdettling@stud.hs-heilbronn.de'
__status__      =     'Development'
```

The **tfrecord** stores the classes for each annotated object with the number related to the class. This relation is defined in the `./DLTD/labelMap.pbtxt` file which was created previously in this notebook. Initially, a small helper function is set up, which can read in the file and map the class sting to its respective ID.

Secondly, a very small helper function `__split` to split the filename and object is introduced.

Finally, the tfrecord is written by the function `create_tf` .

In [43]:

```
# ------------------------------------------------------------------------------------
# Function to read in the label data from the label map file
# ------------------------------------------------------------------------------------
def read_label_map(label_map_path):

    # Initialize variables
    item_id = None
    item_name = None
    items = {}

    # Loop over the file and extract the IDs and label names
    with open(label_map_path, "r") as file:

        # Read line by line
        for line in file:

            # Replace spaces
            line.replace(" ", "")

            # Search for first line - do nothing - no information
            if line == "item{":
                pass

            # Search for last line - do nothing - no information
            elif line == "}":
                pass

            # Search for second line - contains numeric ID
            elif "id:" in line:
                item_id = int(line.split(":", 1)[1].strip())

            # Search for third line - contains class ID as string
            elif "name" in line:
                item_name = line.split(":", 1)[1].replace("'", "").strip()

            # Save to dict - reset variables
            if item_id is not None and item_name is not None:
                items[item_name] = item_id
                item_id = None
                item_name = None

    # Return a dict containing all the labels with corresponding IDs
    return items

# ------------------------------------------------------------------------------------
# Function to split filename and object, returns group of objects belonging to same file
# ------------------------------------------------------------------------------------
def __split(df, group):

    # Search for filenames in data
    data = namedtuple('data', ['filename', 'object'])

    # Group by filename
    gb = df.groupby(group)

    # Return all labels as group which belong to the same file name
    return [data(filename, gb.get_group(x)) for filename, x in\
            zip(gb.groups.keys(), gb.groups)]
```

```
# ------------------------------------------------------------------------------------------
# Function to create tf record samples
# ------------------------------------------------------------------------------------------
def create_tf(group, path, class_dict):

    # Encoded JPEG of the original image
    with tf.io.gfile.GFile(os.path.join(path, '{}'.format(group.filename)), 'rb') as fid:
        encoded_jpg = fid.read()
    encoded_jpg_io = io.BytesIO(encoded_jpg)

    # Open the image
    image = Image.open(encoded_jpg_io)

    # Read image dimensions
    width, height = image.size

    # Initialize the encoding and bounding boxes
    filename = group.filename.encode('utf8')
    image_format = b'jpg'
    xmins = []
    xmaxs = []
    ymins = []
    ymaxs = []
    classes_text = []
    classes = []

    # Loop over all data of the bounding boxes in one group (filename)
    for index, row in group.object.iterrows():

        # Normalize position in the image [0-1]
        xmin = row['xmin'] / width
        xmax = row['xmax'] / width
        ymin = row['ymin'] / height
        ymax = row['ymax'] / height

        # Append the bounding box coordinates
        xmins.append(xmin)
        xmaxs.append(xmax)
        ymins.append(ymin)
        ymaxs.append(ymax)

        # Append the encoded classes and text
        classes_text.append(str(row['class']).encode('utf8'))
        classes.append(class_dict[str(row['class'])])

    # Write the record
    tf_example = tf.train.Example(features=tf.train.Features(
    feature={
        'image/height': dataset_util.int64_feature(height),
        'image/width': dataset_util.int64_feature(width),
        'image/filename': dataset_util.bytes_feature(filename),
        'image/source_id': dataset_util.bytes_feature(filename),
        'image/encoded': dataset_util.bytes_feature(encoded_jpg),
        'image/format': dataset_util.bytes_feature(image_format),
        'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
        'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
        'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
        'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
        'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
        'image/object/class/label': dataset_util.int64_list_feature(classes), }))
```

```
    # Return the record
    return tf_example
```

The main function reads the previously created `labelMap.pbtxt` and `TRAIN.csv` files and creates the tfrecord based on this input. This process is quite fast for the small example set but takes up to half an hour for the complete DLTD train dataset.

In [61]:

```
# --------------------------------------------------------------------------------------------
# Main function for tf record generation
# --------------------------------------------------------------------------------------------
if __name__ == '__main__':

    # Define the locations of the relevant files - general variables
    pbtxtInput = "./DLTD/labelMap.pbtxt"
    imageDir = ""

    # Split specific files
    outputPath = "./DLTD/TRAIN.tfrecord"
    csvInput = "./DLTD/TRAIN.csv"

    # Read in the dictonary with the class information
    classDict = read_label_map(pbtxtInput)

    # Call the output writer
    writer = tf.compat.v1.python_io.TFRecordWriter(outputPath)

    # Get path of images
    path = os.path.join(imageDir)

    # Read in all examples of the CSV input file
    examples = pd.read_csv(csvInput)

    # Split the examples by filename
    grouped = __split(examples, 'filename')

    # Loop over all examples, generate tf record and montior progress
    for group in tqdm(grouped, desc='groups'):

        # Create TF format for example
        tfExample = create_tf(group, path, classDict)

        # Write example to file
        writer.write(tfExample.SerializeToString())

    # Close the writer
    writer.close()

    # Display the write path
    outputWritePath = os.path.join(os.getcwd(), outputPath)

    # Finished
    print('Successfully created the TFRecords: {}'.format(outputWritePath))

# EOF
```

```
groups: 100%|██████████| 20/20 [02:53<00:00,  8.66s/it]

Successfully created the TFRecords: /media/tim/Data/03_Studium/02_Master/03_
Studium/02_Semester2/06_DeepLearning/project/DL_traffic_sign_traffic_light_d
etection/00_Documentation/01_Traffic_light/./DLTD/TRAIN.tfrecord
```

By this, the process of creating tfrecords is showcased. The `TRAIN.tfrecord` file is now available along with all previously created files in `./DLTD/` . You could try to alter the parameters **csvInput** to `./DLTD/EVAL.csv`

and **outputPath** to `./DLTD/EVAL.tfrecord` in the script above to create the file to create the record needed for evaluation. With this, the major component to start a training is available. The only additionally needed component is the pipeline configuration, which will be introduced in the next chapter.

6.2 Pipeline configurations

Additional to the `TRAIN.tfrecord` and `EVAL.tfrecord`, a so called pipeline file is needed. The `pipeline.config` file is defining the training parameters and is also used to define several predefined augmentations to the images. The file itself is provided as a pretrained model from the Tensorflow object detection API, but is modified for the project on hand.

As the pipeline file is quite large and several different pipeline configurations have been used during the project on hand, only the final configuration is shown in this notebook. However, the variation points of last three training iterations are shownn in the next subchapters. Please note, that these three trainings are not the only iterations done within the project. However, the majority of iterations are mainly focussed on finding the correct class mapping algorithm for the dataset (see **chapter 3.1.3**) and trained on a personal PC with limited computation power. To keep the notebook as short as possible only the latest configurations, trained on more powerful HHN-PCs are mentioned in the next chapters. The configurations are:

Configuration	Comment
Adam_aipc	Trained on HHN AI-PC, data augmentations used.
Adam_aipc_dropout	Trained on HHN AI-PC, data augmentations and dropout used.
Adam_dgx_dropout_batchsize	Trained on HHN dgx station, higher batchsizes possible

First, an overview of the general file structure can be seen below. In the subsequent sections, the relevant parts of the file are described.

```
# Faster R-CNN with Resnet-50 (v1)
# Trained on COCO, initialized from Imagenet classification checkpoint

# This config is TPU compatible.

model {
  faster_rcnn {
    num_classes: 35
    image_resizer {
      fixed_shape_resizer {
        width: 1024
        height: 1024
      }
    }
    feature_extractor {
      type: 'faster_rcnn_resnet101_keras'
      batch_norm_trainable: true
    }
    first_stage_anchor_generator {
      grid_anchor_generator {
```

```
    scales: [0.25, 0.5, 1.0, 2.0]
    aspect_ratios: [0.5, 1.0, 2.0]
    height_stride: 16
    width_stride: 16
  }
}
first_stage_box_predictor_conv_hyperparams {
  op: CONV
  regularizer {
    l2_regularizer {
      weight: 0.0
    }
  }
  initializer {
    truncated_normal_initializer {
      stddev: 0.01
    }
  }
}
first_stage_nms_score_threshold: 0.0
first_stage_nms_iou_threshold: 0.7
first_stage_max_proposals: 300
first_stage_localization_loss_weight: 2.0
first_stage_objectness_loss_weight: 1.0
initial_crop_size: 14
maxpool_kernel_size: 2
maxpool_stride: 2
second_stage_box_predictor {
  mask_rcnn_box_predictor {
    use_dropout: true
    dropout_keep_probability: 0.8
    fc_hyperparams {
      op: FC
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
      initializer {
        variance_scaling_initializer {
          factor: 1.0
          uniform: true
          mode: FAN_AVG
        }
      }
    }
    share_box_across_classes: true
  }
}
second_stage_post_processing {
  batch_non_max_suppression {
    score_threshold: 0.0
```

```
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 300
      }
      score_converter: SOFTMAX
    }
    second_stage_localization_loss_weight: 2.0
    second_stage_classification_loss_weight: 1.0
    use_static_shapes: true
    use_matmul_crop_and_resize: true
    clip_anchors_to_image: true
    use_static_balanced_label_sampler: true
    use_matmul_gather_in_matcher: true
  }
}

train_config: {
  batch_size: 2
  sync_replicas: true
  startup_delay_steps: 0
  replicas_to_aggregate: 8
  num_steps: 1000000
  optimizer {
    adam_optimizer: {
      learning_rate: {
        cosine_decay_learning_rate {
          learning_rate_base: 3.3e-5
          total_steps: 1000000
          warmup_learning_rate: 1.0e-5
          warmup_steps: 2000
        }
      }
    }
    use_moving_average: false
  }
  fine_tune_checkpoint_version: V2
  fine_tune_checkpoint: "/PATH_TO_CHECKPOINT/ckpt-00"
  fine_tune_checkpoint_type: "detection"

  max_number_of_boxes: 100
  unpad_groundtruth_tensors: false
  use_bfloat16: false  # works only on TPUs


  data_augmentation_options {
    random_image_scale {
    min_scale_ratio: 0.9
    max_scale_ratio: 1.1
    }
  }

  data_augmentation_options {
```

```
    random_adjust_hue {
    }
  }

  data_augmentation_options {
    random_adjust_contrast {
    }
  }

  data_augmentation_options {
    random_adjust_saturation {
    }
  }

   data_augmentation_options {
    random_adjust_brightness {
    }
  }

}
train_input_reader: {
  label_map_path: "/PATH_TO_LABELMAP/labelMap.pbtxt"
  tf_record_input_reader {
    input_path: "/PATH_TO_TRAIN/TRAIN.record"
  }
}

eval_config: {
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
  num_visualizations: 1000
  batch_size: 1;
}

eval_input_reader: {
  label_map_path: "/PATH_TO_LABELMAP/labelMap.pbtxt"
  shuffle: false
  num_epochs: 1
  tf_record_input_reader {
    input_path: "/PATH_TO_EVAL/EVAL.tfrecord"
  }
}
```

6.2.1 Network definitions - dropout

Initially, the `pipeline.config` file contains information on the used network. Here, most of the information is already defined by the baseline config delivered with the pretrained network. Therefore, only the major changes to this section will be discribed below.

The most important and mandatory adaption is the parameter `num_classes`. As described in **chapter 3.1.3**, the total amount of mapped target classes is **35** and equal for all training configurations.

One variation point between the configurations is the usage of dropout layers. The amount of dropout layers can be defined by the parameters:

- `use_dropout`
- `dropout_keep_probability`

While `use_dropout` is a simple flag to use the dropout option at all, the parameter `dropout_keep_probability` defines the percentage of neurons unaffected by dropout. To get to a common **dropout percentage of 20% - 25%**, the parameters have been tuned to the following values for the different configurations.

Configuration	use_dropout	dropout_keep_probability
Adam_aipc	false	-
Adam_aipc_dropout	true	0.75
Adam_dgx_dropout_batchsize	true	0.8

The dropout configuration is defined within these lines (for `Adam_aipc_dropout`):

```
mask_rcnn_box_predictor {
        use_dropout: true
        dropout_keep_probability: 0.8
        fc_hyperparams {
          op: FC
          regularizer {
            l2_regularizer {
              weight: 0.0
            }
          }
```

6.2.2 Training configuration - batch size

The batch size is a crucial parameter for maschine learning tasks and can be defined within the training configuration with the parameter `batch_size`. In theory, a batch shall represent the complete variety of the dataset, as the weights are updated based on each batch. As the dataset on hand is very big and the ressources are limited, this is not completely possible. However, for the project on hand, two stationary PCs at the Hochschule Heilbronn have been set up for the training. However, the maximum realizable batch size is still limited to **32**. Below, the different configurations are depicted.

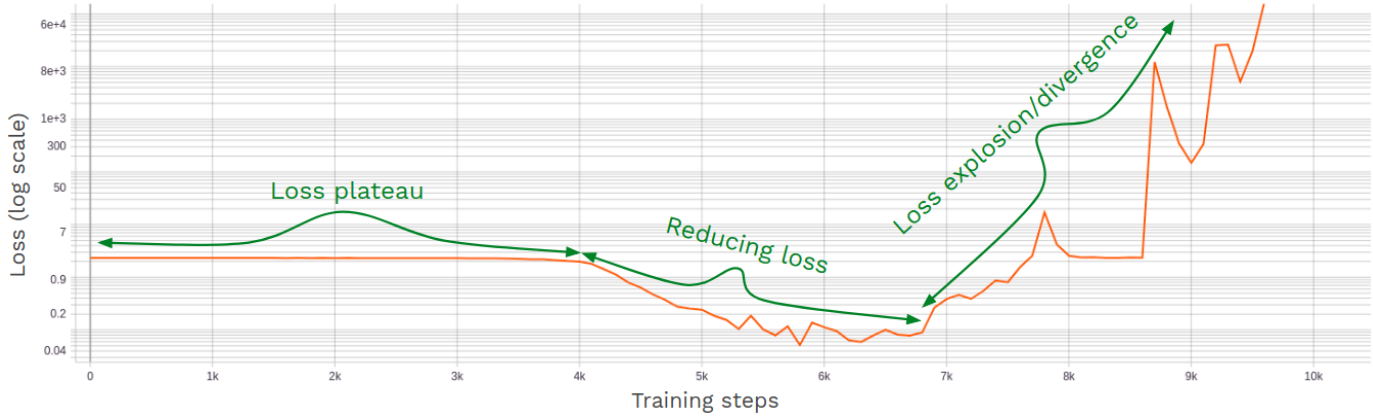
Configuration	batch_size
Adam_aipc	2
Adam_aipc_dropout	2
Adam_dgx_dropout_batchsize	32

6.2.3 Training configuration - optimizer and learning rate

For all final training configurations, an ADAM optimizer was used. Within the pipeline file, there is also the option to tune the learning rate, as a hyperparameter. Here, not only a fixed learning rate can be defined, but also a dynamically adapted learning rate decay can be applied. To find the optimal learning rate, the method of lerning rate finding is applied.

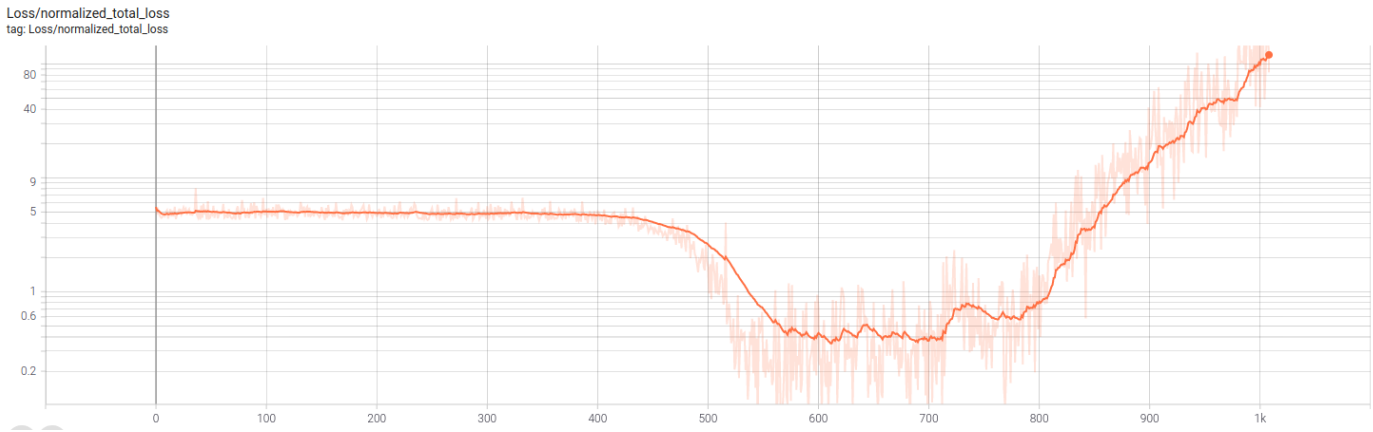
> **Finding the optimal learning rate:** https://medium.com/octavian-ai/how-to-use-the-learning-rate-finder-in-tensorflow-126210de9489 (https://medium.com/octavian-ai/how-to-use-the-learning-rate-finder-in-tensorflow-126210de9489)

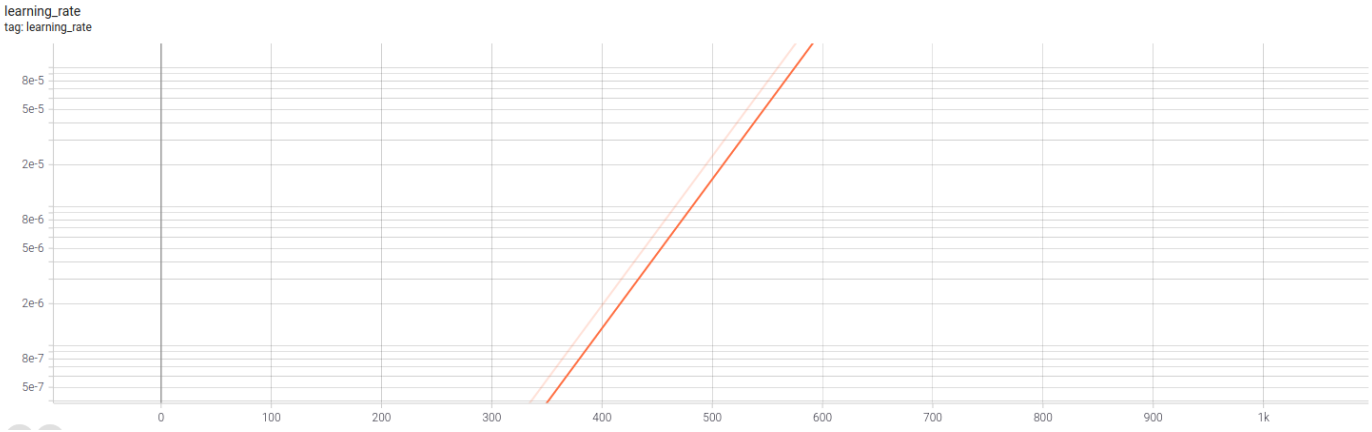
The general method is to start a experiment with a very low learning rate and then exponentially increasing this rate while monitoring the training loss. Initially, the loss is unchanged as the very low learning rate is limiting the weight updates to very small updates. Then, as the learning rate reaches sufficient levels, the loss is reducing ever so steeply, before the learning rate gets too high, the training process diverges and the loss is exponentially increasing again. The optimal learning rate is reached at the turning point of the loss curve. The theoretical curve is depicted here:



Source: https://miro.medium.com/max/2000/1*Mo1cw6dPTMx5hw8v0ekJZA.png (https://miro.medium.com/max/2000/1*Mo1cw6dPTMx5hw8v0ekJZA.png) (05.07.2021)

For the project on hand, the process looks like this. It can easily be seen, that the loss function has its first turning point around step **530**, which corresponds to a learing rate of **3.3e-5**:





Source: own creation

The learning rate configuration is equal for all trained configurations.

```
num_steps: 1000000
optimizer {
    adam_optimizer: {
      learning_rate: {
        cosine_decay_learning_rate {
          learning_rate_base: 3.3e-5
          total_steps: 1000000
          warmup_learning_rate: 1.0e-5
          warmup_steps: 2000
        }
      }
    }
    use_moving_average: false
  }
```

6.2.4 Training configuration - image augmentations

Another important configuration is the data augmentation. The data augmentation is crucial to alter the input information randomly, allowing the network to generalize the information even better. For rel-world camera based detection systems, the input images highly differ depending on the environments illumination, weather and general surroundings. These factors mainly influence the the images colors, contrast adnd brightness. Even though the dataset is recorded in real world, images in the same surrounding recorded from different camera hardware (like in the HHN Passat) would potentially look very different. To mimic different HW configurations, predefined operations on the image colors, contrast and brightness are enabled in the training configuration:

- `random_adjust_hue`
- `random_adjust_contrast`
- `random_adjust_saturation`
- `random_adjust_brightness`

Additionally, geometric data augmentations can be defined. However, these augmentations need to be applied with great care, as a geometric operation can also alter the objects meaning and classification. A horizontal flip of a traffic light with the `pictogram` **arrow_left** would for example alter the `pictogram` to **arrow_right** which would not match to the label any more. Therefore, only image rescaling is applied by using `random_image_scale` . The image is scaled between **0.9 and 1.1 times** the original image size ranomly.

```
data_augmentation_options {
    random_image_scale {
    min_scale_ratio: 0.9
    max_scale_ratio: 1.1
    }
  }

  data_augmentation_options {
    random_adjust_hue {
    }
  }

  data_augmentation_options {
    random_adjust_contrast {
    }
  }

  data_augmentation_options {
    random_adjust_saturation {
    }
  }

   data_augmentation_options {
    random_adjust_brightness {
    }
  }
```

6.2.5 Training configuration - input files

For the training configuration, finally the main input files need to be defined:

- `TRAIN.tfrecord`
- `label_map.pbtxt`

The creation of the file `TRAIN.tfrecord` is described in **chapter 5.1**. The file contains all information, image data and annotations. In the pipeline file, only the link to this file must be provided.

Also the file `label_map.pbtxt` was created already within this notebook. It contains the mapping of the class strings to an integer enumeration. Also for this file, a simple link needs to be provided.

The definition is given within these lines:

```
train_input_reader: {
  label_map_path: "/PATH_TO_LABELMAP/label_map.pbtxt"
  tf_record_input_reader {
    input_path: "/PATH_TO_TRAIN/TRAIN.record"
  }
}
```

6.2.6 Evaluation configuration

The evaluation configuration is very simple and identical for all configurations. Just like for the training, first two files must be defined:

- `EVAL.tfrecord`
- `label_map.pbtxt`

Additionally, the main parameters are:

- `batch_size`
- `num_visualizations`
- `metrics_set`

For the `metrics_set` , the predefined metrics_set `coco_detection_metrics` is used. The coco dataset is a very common dataset to test and benchmark general behaviour of object detection networks. The evaluation metrics contain the for example the loss, recall and precision for the evaluated dataset.

As the evaluation will run parallel to the training on the CPU, it was found that the `batch_size` has only a very slight influence on the speed of the evaluation process. As no weight update is done based on the evaluation, the parameter also has no influence on the quality or convergence of the training process. To make the evaluation run on any given maschine, the `batch_size` was universally set to **1**.

`num_visualizations` is defining the amount of image data to be represented in the tensorboard. The tuning of this parameter is mainly limited by the disk size of the maschine, as the evaluation output contains all `num_visualizations` images per evaluation step. A higher number of visualizations is therefore also slowing down the analysis process significantly. As the evaluation dataset is containing **~6000 images**, the parameter is set to **1000** to get a significant overview of the evaluation data while maintaining reasonable manageability of the large evaluation output files.

The complete structure for the evaluation part is given to:

```
eval_config: {
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
  num_visualizations: 333
  batch_size: 1;
}

eval_input_reader: {
  label_map_path: "/PATH_TO_LABELMAP/label_map.pbtxt"
  shuffle: false
```

```
    num_epochs: 1
    tf_record_input_reader {
      input_path: "/PATH_TO_EVAL/EVAL.tfrecord"
    }
  }
```

7 - Training results

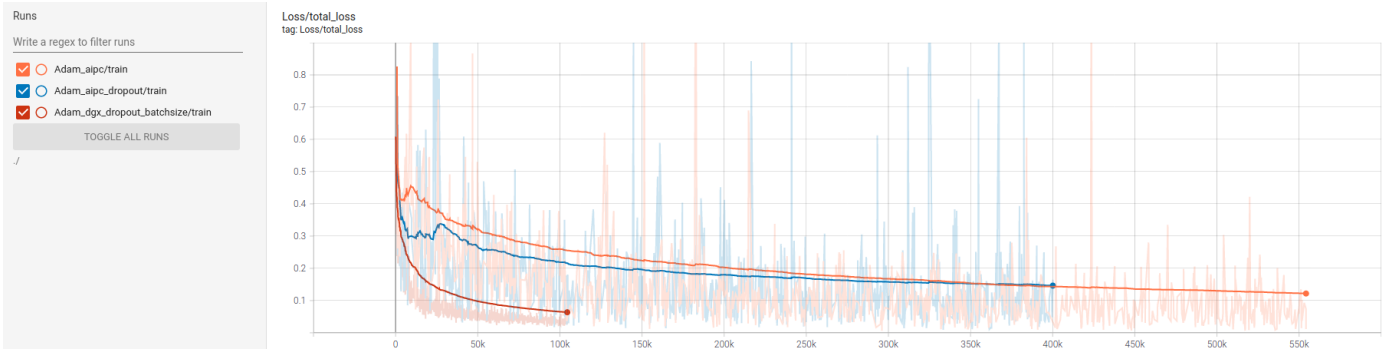
In the following chapter, the training results shall be discussed for the three configurations mentioned previously. However, the notebook is focussed on the results given in **chapter 8**, so this section will only provide a very brief overview of the most relevant metrics for the training process.

> **Remark:** The training process for the procject on hand is very time consuming due to the size of the datasets. A typical training takes up to six days. Therefore, due to restricted project time, not all training processes could be trained to the same amount of training steps.

The convergence of the training processes can be evaluated using the `total_loss` metric of the `train` data. The image below shows the development of the `total_loss` over the three training configurations. There are two main obsevations:

First, using a dropout configuration results in a faster convergance of the training process. The configurations `Adam_aipc` and `Adam_aipc_dropout` , reperesented by the orange and blue line, share the very same configuration except for the additional dropout layer within the `Adam_aipc_dropout` configuration. The chart below shows the faster convergence of the blue curve compared to the orange curve without dropout.

The second observation is that the batch size has a great influence on the convergence speed and the final result of the training. The red curve refers to the configuration `Adam_dgx_dropout_batchsize` , which is trained n the dgx station with **batch_size 32**. While the other trainings converge more or less at the same final `total_loss` values of **0.15**, the magenta curve is converging at a significantly better value of **0.05**. By looking at the gradient of the red curve, the training could even reached slightly better values given more training time.

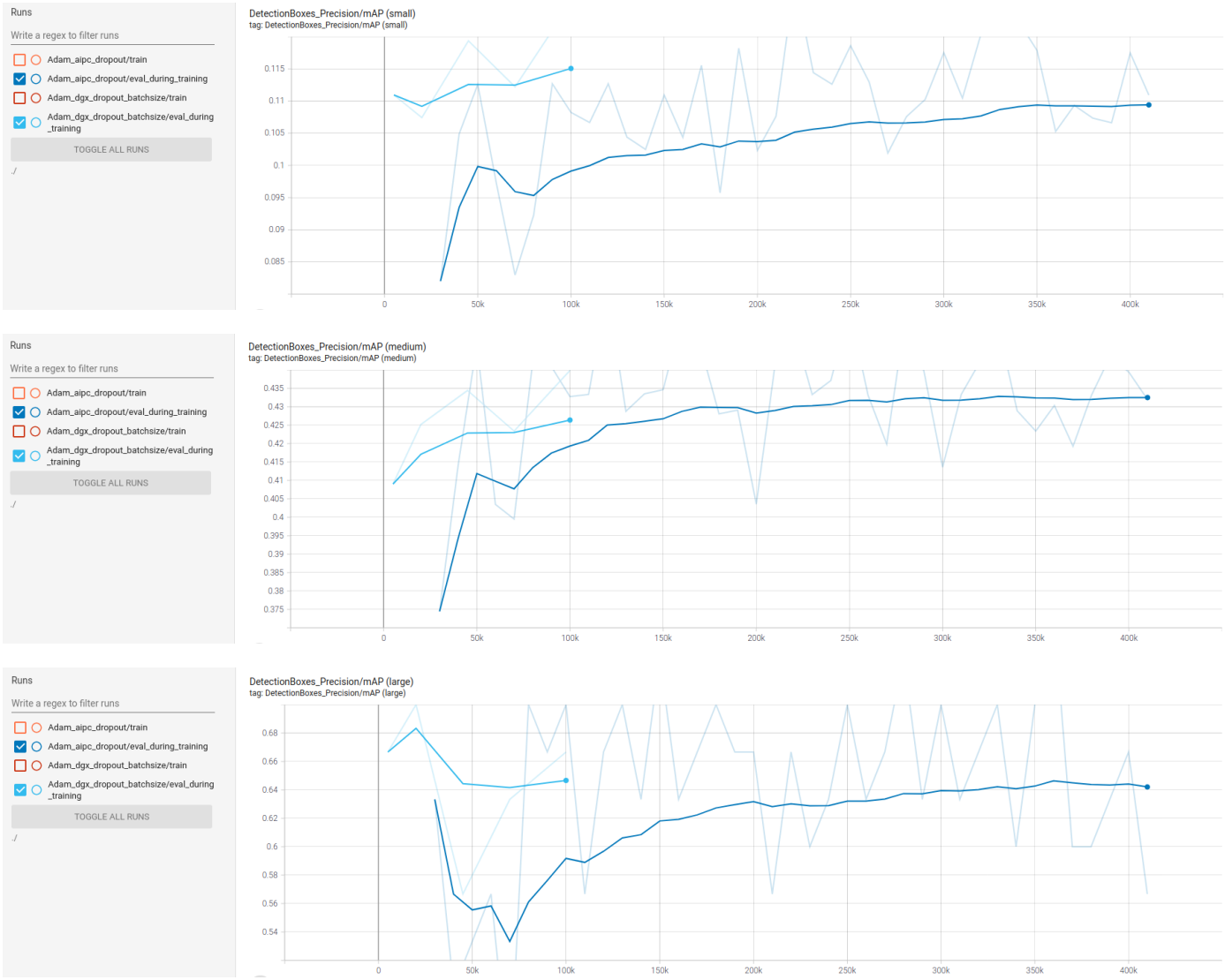


Source: own creation

In order to validate the training process, the `Precision` (generally the mean average Precision **mAP**) on the `eval` dataset is a good metric. As the evaluation records provided by tensorflow contain 1000 images per evaluated steps, the records are of very large file size. Displaying the evaluation results of multiple trainings at once therefore requires a lot of memory which is not available on the private notebook. Therefore in the following section, only the two best configurations, `Adam_dgx_dropout_batchsize` and `Adam_aipc_dropout` will be compared.

The `Precision` values are differenciated for **large, medium and small** area objects. Over both trainings, the performance on both metrics are by far better on **large** objects and reduce for **medium** and **small** objects. However, especially for the small size objects, the `Adam_dgx_dropout_batchsize` training shows significantly better results. Furthermore, as the performance metrics on both trainings increase until the end of the training, hence no overfitting was observed during the training. Overall, the best configuration `Adam_dgx_dropout_batchsize` shows a `Precision` of **0.645** on large objects in the `evaluation` data. The images below show the `Precision` graphs for **large, medium and small** area objects.

A sophisticated discussion of the performance metrics on the `test` dataset is provided in the following section.



Source: own creation

8 - Test results

To test the performance on previously unseen data, the third partition of the dataset, `TEST.tfrecord` is used. The test dataset contains the following amounts of data:

- **6225** images
- **24603** annotations

Due to the size of the dataset, a detailed evaluation of the performance per class is not possible. The evaluation however provides values averaged over all classes, which will be discussed and compared to the evaluation values. Furthermore, good and weak example scenes of the test dataset will be presented and discussed. Finally, in **chapter 9**, the main influences on the remaining limitations are discussed and concluded in detail.

8.1 Performance metrics

First, the performance metrics based on all classes can be accessed. Here, the COCO metrics provide the averaged Precision and Recall. The metrics are additionally broken down into **small, medium and large** area objects. For applications in autonomous vehicles, usually the **medium and large** sized objects are relevant, as bigger objects are closer to the ego vehicle.

The performance metrics show clearly, that the networks performance is highly reliant on the size of the object. Smaller object have both weak Precision and Recall values, while the performance on large objects with a averaged **Precision 0.753** and a averaged **Recall of 0.774** is much better. Also these values should be be viewed in the context of the known influences discussed in **chapter 9**. Overall, the performance on relevant, large objects is pleasing and feasible for real-world autonomous applications and comparable to the evaluation metrics during training.

> **Remark:** Please refer to **chapter 9** for the discussion of the root causes and influences on the performance metric values.

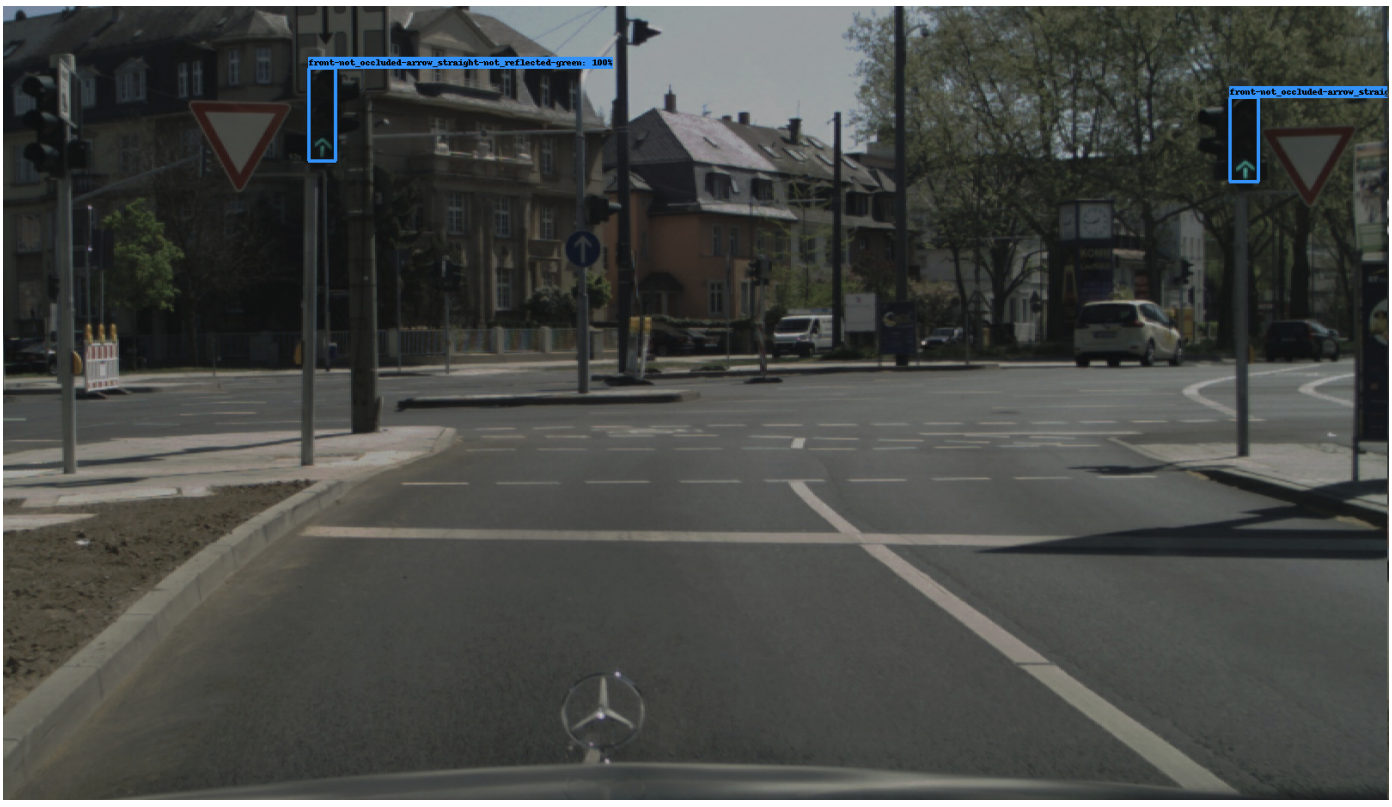
```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.175
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.298
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.183
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.117
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.437
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.753

Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.260
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.373
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.375
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.304
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.607
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.774
```

8.2 Good scenes

Generally, a majority of scenes shows pleasing results. The scenes given below are depicted with the networks predictions in the left section compared to the ground truth labels on the right side. As the predictions in theses scenes mainly match with the ground truth data, only a few zoomed pictures are provided in this section. The class of the detected objects is encoded by the bounding boxes colour, so that the correct classification can be evaluated at the first look even for small pictures.

> **Remark:** Not all scenes with good performance are provided in this section, the aim is more to give an overview of the performance for different classes.

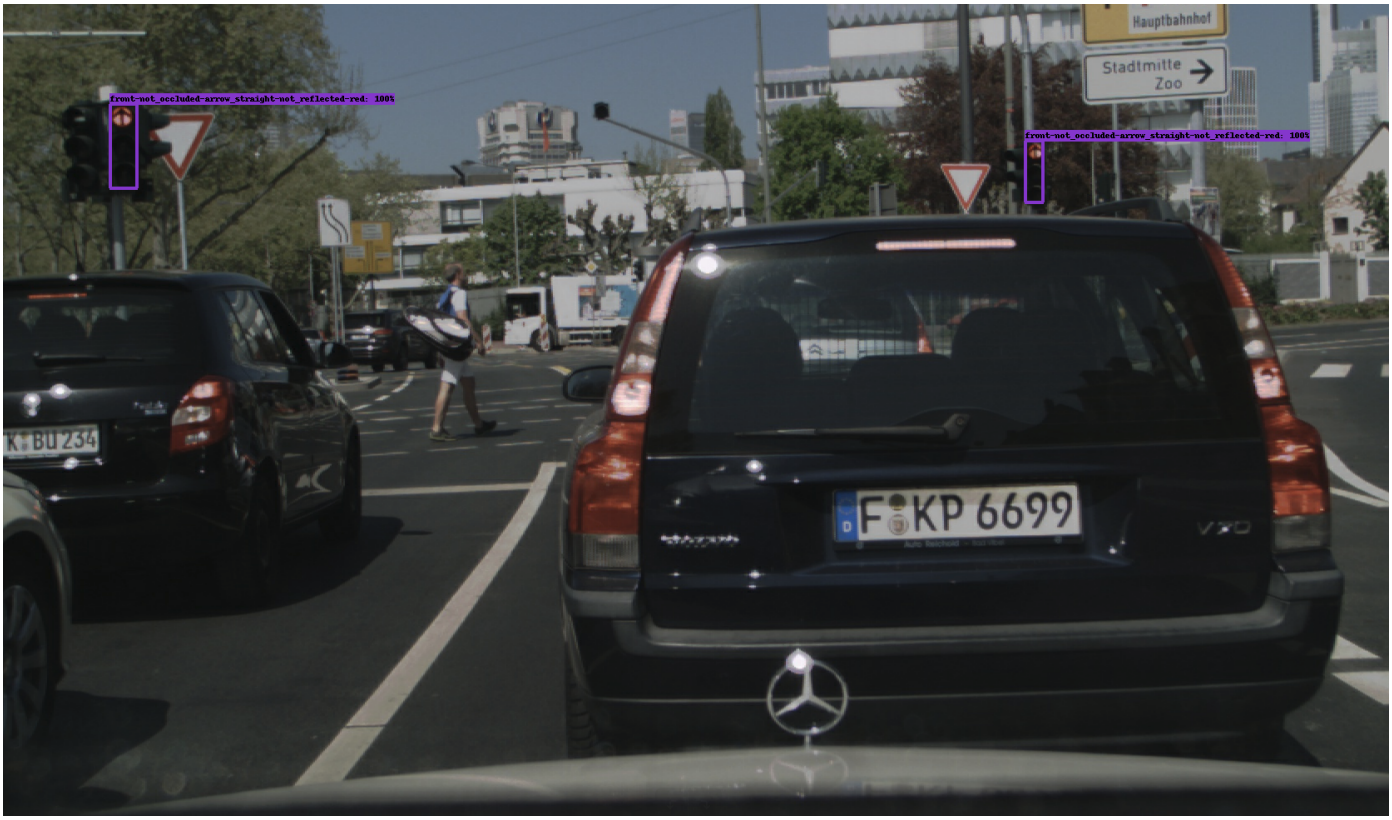
8.2.1 Green vehicle relevant traffic lights

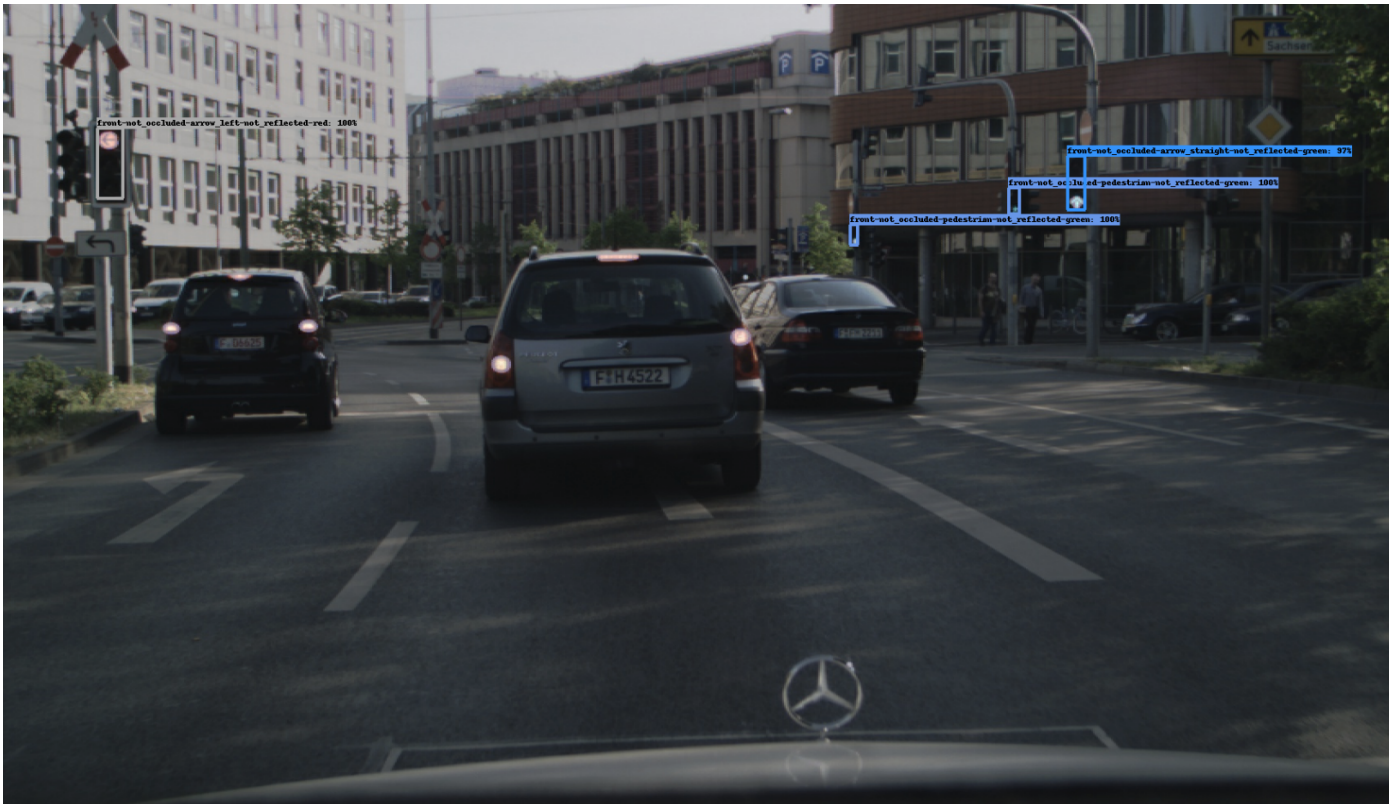


All sources: Source: DLTD dataset, Berlin, own network classifications

The first three images above show green vehicle relevant traffic lights in different variations. The fourth image is a zoomed into the detections of a certain scene. It can be seen, that not only the `state` of the traffic light is correctly classified to `green`, but also the `pictogram` is correctly set to `arrow_straight`, resulting in the classification `front-not_occluded-arrow_straight-not_reflected-green`.

8.2.2 Red vehicle relevant traffic lights

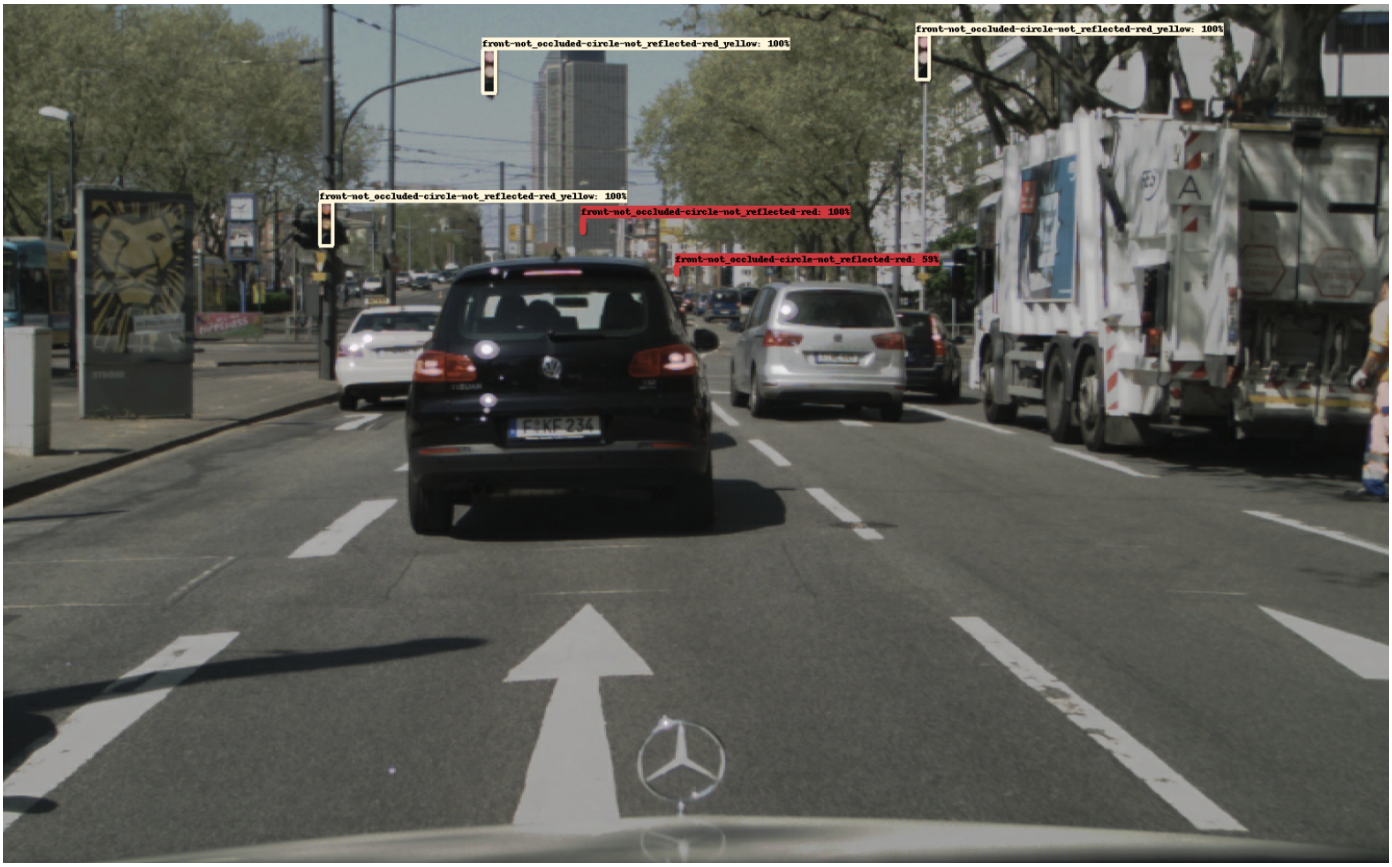




All sources: Source: DLTD dataset, Berlin, own network classifications

The scenes above show the detection performance on red vehicle relevant signs. It is also visible in the two zoomed in pictures, that the network is able to differenciate between different `pictogram` types and classify the objects correctly to `front-not_occluded-arrow_straight-not_reflected-red` and `front-not_occluded-arrow_left-not_reflected-red` .

8.2.3 Yellow or red_yellow vehicle relevant traffic lights



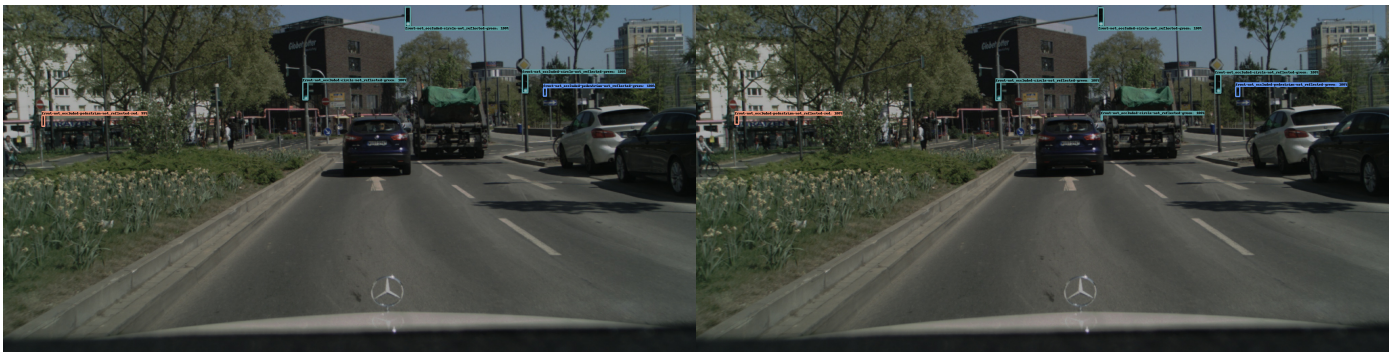
All sources: Source: DLTD dataset, Berlin, own network classifications

The picture above contains a very rarely occuring class, `front-not_occluded-circle-not_reflected-red_yellow` . It can be seen, that even this rarely occuring class is detected and classified correctly for larger objects. The second picture shows a zoomed in view of the detections of the neural network.

8.2.4 Pedestrian and bicycle relevant traffic lights

Finally, also the detections on pedestrian and bicycle traffic lights show pleasing results. As the annotations are usually quite small for these classes, each of the scenes is provided as a side-by-side image and as a zoomed

detection image.



All sources: Source: DLTD dataset, Berlin, own network classifications

8.3 Weak scenes

Apard from the majority of good scenes, frequently appearing errors shall be discussed in the following chapter. The vast majority of weak scenes refer to the classification of the object, not the detection. These misclassifications are often caused by very small objects with not detectable `pictograms` due to the small size. While the following section only shows the weak scenes within the test dataset, the influences on the general behavior is discussed in **chapter 9**.

> **Remark:** Not all scenes with weak performance are provided in this section, the aim is instead to give an overview of the performance for different classes.

8.3.1 False Negatives on small objects

As the metrics show, small objects are generally not as reliably detected as bigger ones, resulting in false negatives on small objects. This problem is discussed in **chapter 9** in detail. For the following scenes, first the side by side image with the networks detections to the left and the ground truth data on the right provided. Secondly, a zoomed in image of the ground truth data is provided.


front-not_occluded-arrow_left-not_reflected-red: 100% 100%
front-not_occluded-circle-not_reflected-green: 100%
front-not_occluded-circle-not_reflected-green: 100%
front-not_occluded-circle-not_reflected-green: 100%
STARCAR
Autovermietung


FB SN 1964
FB SN 1964


front-not_occluded-circle-not_reflected-red: 100%
front-not_occluded-circle-not_reflected-red: 100%
front-not_occluded-pedestrian-not_reflected-red: 100%
FB SN 1964

All sources: Source: DLTD dataset, Berlin, own network classifications

8.3.2 False positives on small objects

Generally, false positives are occuring only very rarely within the test data. The observed false positives however are mainly rear lights of other traffic participants, which are wrongly detected as objects with `state` "red". The following image shows a false positive of the class `front-not_occluded-pedestrian-not_reflected-red` on a busses rear light. As these false positives are usually happening on very small objects, the network classifies the objects normally with the `pictogram` "pedestrian".

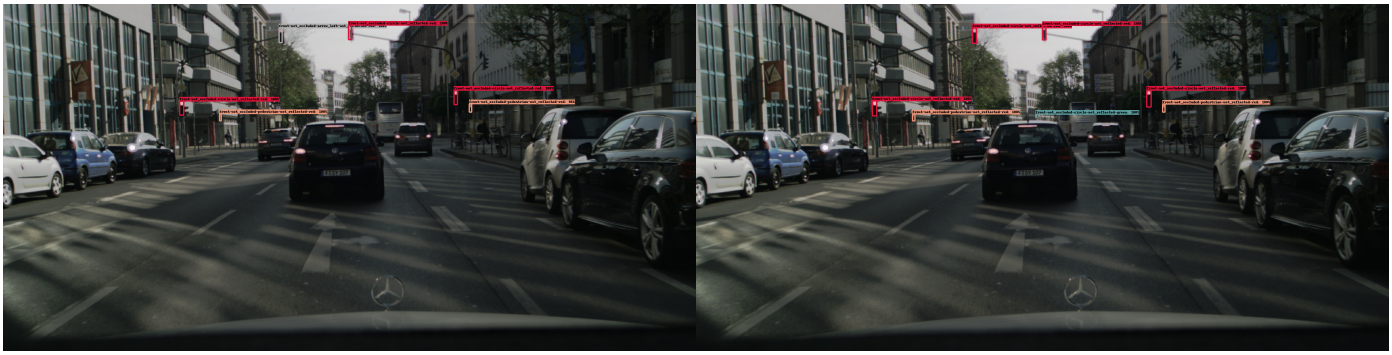
All sources: Source: DLTD dataset, Berlin, own network classifications

Another cause for false positives are reflections in vehicle mirrors. In the following image,the reflection in the mirror of the vehicle to the right contains a green round shape. This leads to a false positive of the class `front-not_occluded-pedestrian-not_reflected-green`. Additionally, the scene has a lot of traffic lights with state `off` which are not detected by the network in this case.

All sources: Source: DLTD dataset, Berlin, own network classifications

8.3.3 Misclassifications

Misclassifications of objects are mainly observed on classes with the same `state` of the traffic light, e.g. "red". In the following image, a red traffic lights `pictogram` is wrongly classified as "arrow_left" resulting in the class `front-not_occluded-arrow_left-not_reflected-red`, while the correct class would be `front-not_occluded-circle-not_reflected-red`. This effect is discussed in detail in **chapter 9**.



All sources: Source: DLTD dataset, Berlin, own network classifications

In the following image, a red traffic lights `pictogram` is wrongly classified as "arrow_left" resulting in the class `front-not_occluded-circle-not_reflected-red`, while the correct class would be `front-not_occluded-arrow_right-not_reflected-red`. Also, a very distant pedestrian traffic light is incorrectly classified as `state` **off**.

All sources: Source: DLTD dataset, Berlin, own network classifications

9 - Influences on remaining limitations and performance metrics discussion

The following section discusses the influences on the remaining limitations seen within the test data. Also, explanations for the **Precision and Recall metrics** in the test images are given.

9.1 Class separation per state and pictogram

Other than problems in the detection of the object, the correct classification of the object is a difficult task. Especially the pictogram of the traffic light is often not visible to the human eye in greater distances. Since the objects are labeled in the context of the previous and following images of the same scene, the `pictogram` is annotated for distant objects even though the pictogram itself is not visible to the human eye. The network can only guess the correct classification based on the pixel position of the object in the image.

However, the class mapping algorithm mentioned in **chapter 3.1** is generating separate classes for different `pictograms` intentionally, as the information on the pictogram of the traffic light is crucial for understanding the situation in a autonomous vehicle application. This approach however differs from the majority of previously publicated studies, which treat the problem of detecting traffic lights as a **three to four class problem** based solely on the `state` of the traffic light (in contrast to **35 seperate classes** in the project on hand).

> **Source:** https://arxiv.org/abs/1805.02523 (https://arxiv.org/abs/1805.02523) (20.07.2021)

While the used class mapping approach brings the mentioned benefits for real-world application, it can lead to misclassifications of objects in the distance. This misclassification is usually happening between two classes indicating the same `state` (e.g. **red**) but different pictograms. This problem for example is very prominent between the following two classes:

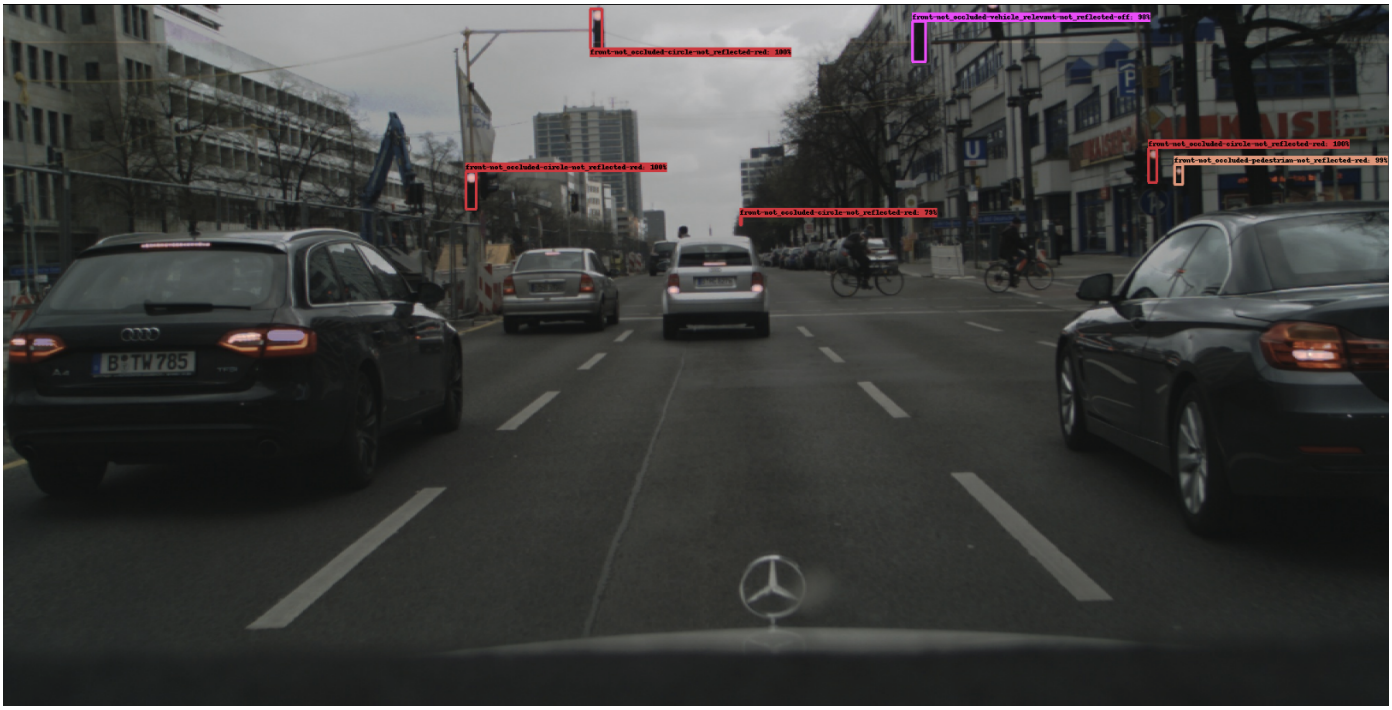
- `front-not_occluded-arrow_left-not_reflected-red`

- `front-not_occluded-circle-not_reflected-red`

In the following picture, a red traffic light with the `pictorgram` **circle** is visible relatively distant on the left of the road. The object should therefore be classified as `front-not_occluded-circle-not_reflected-red` . However, as a pictogram is not really detectable in this distance and red traffic lights at this position (at the left edge of a road and in the left area of the image) typically show an **arrow_left**, the network classifies the object initially as `front-not_occluded-arrow_left-not_reflected-red` .

Source: DLTD dataset, Berlin, own network classifications

As the vehicle is approaching the traffic light and the pictogram is visible, the network classifies the object correctly. The image below shows a frame of the same scene in Berlin. All relevant objects are detected correcly.



Source: DLTD dataset, Berlin, own network classifications

As the main information of a traffic light is the `state` **red**, this is considered to be a "minimal error misclassification" and has only minor impact on the value of the information for the real world application of the network in a autonomous vehicle. Still, while the behaviour of the network is acceptable for a series system, the mentioned behavior is treated as a false positive (FP) and/or false negative (FN) in the performance metrics, resulting in negative influence on Precision and Recall.

9.2 Dataset limitations - annotation problems

The DLTD dataset contains a huge variety of scenes and is by far the most sophisticated dataset for german traffic lights available. Furthermore, the DLTD dataset recently released version 2 of the annotaion data, improving label quality and correcting previously reported errors. However, the label data still has a few problems, which influence the results of the training and test process of the project on hand.

First, there are a lot of very small objects within the dataset, which are not visible to the human eye and contain only a few pixels of information. The image below shows an example of the ground truth data with a large number of very small objects of the class `front-not_occluded-circle-not_reflected-green` which are not visible to the human eye but still labeled. While the system on hand is able to detect all larger traffic lights correctly, the small, distant objects cannot be detected. While this is not a problem for a real life system, as the larger objects are relevant for the vehicle, the not detected traffic lights influence the performance metrics negatively, as each not-detected but labeled object counts as a false negative (FN) and therefore negatively influencing the Recall.

Source: DLTD dataset, Berlin, own ground truth mapped

Another problem are occluded objects or completely non-labeled objects. As objects labeled as "occluded" are usually only partially visible on the image, these objects are sorted out for training, evaluation and test. While this is generally the correct behavior for **highly occluded objects (e.g. only visible to 10%)**, this can cause

problems for very **lightly occluded objects (e.g. visible to 95%)**. Lightly occluded objects are still sorted out and therefore not available as ground truth during eval and test. As the lightly occluded object still looks very similar to other not occluded traffic lights, the network can detect the object. This is negatively influencing the Precision metric, as the detection is wrongly classified as a false positive (FP). The same problem is also caused if a object is not labeled at all.

The following image shows the detections of the network on the left and the ground truth data on the right. The traffic light at the left side is slightly occluded but still detected correctly. As the object is labeled as occluded, the ground truth data does not include this traffic light, hence the detection is wrongly classified as a false positive.

Source: DLTD dataset, Berlin, own ground truth mapped

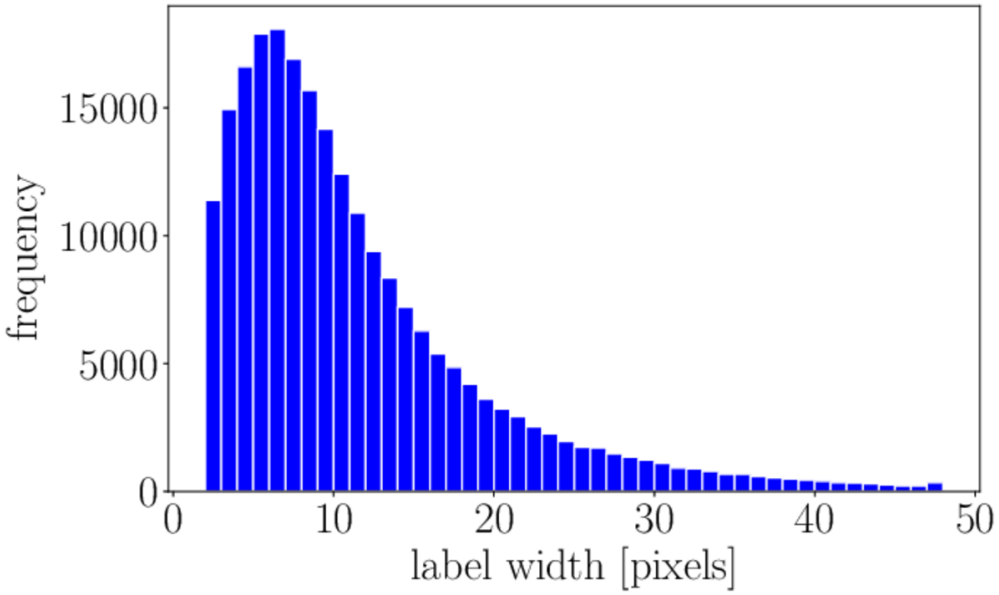
Zooming into the ground truth image, it is visible that the object is in fact slightly occluded by the construction barrier:

Source: DLTD dataset, Berlin, own ground truth mapped

9.3 Dataset imbalance - annotation size

Due to the sophisticated, pixel-accurate label process used to create the DLTD dataset, very small annotations of very distant traffic lights are provided in the dataset. Generally, this is a advantage, as the detection of distant traffic lights can be very beneficial for the real world application in a autonomous vehicle. On the other hand, distant traffic lights are hard to detect and generalize for the network, as the information is only consisting of a few pixels. This theory is backed by the training, validation and test data, as the Precision and Recall are generally the highest for **large** objects in contrast to beeing very small for **small** objects.

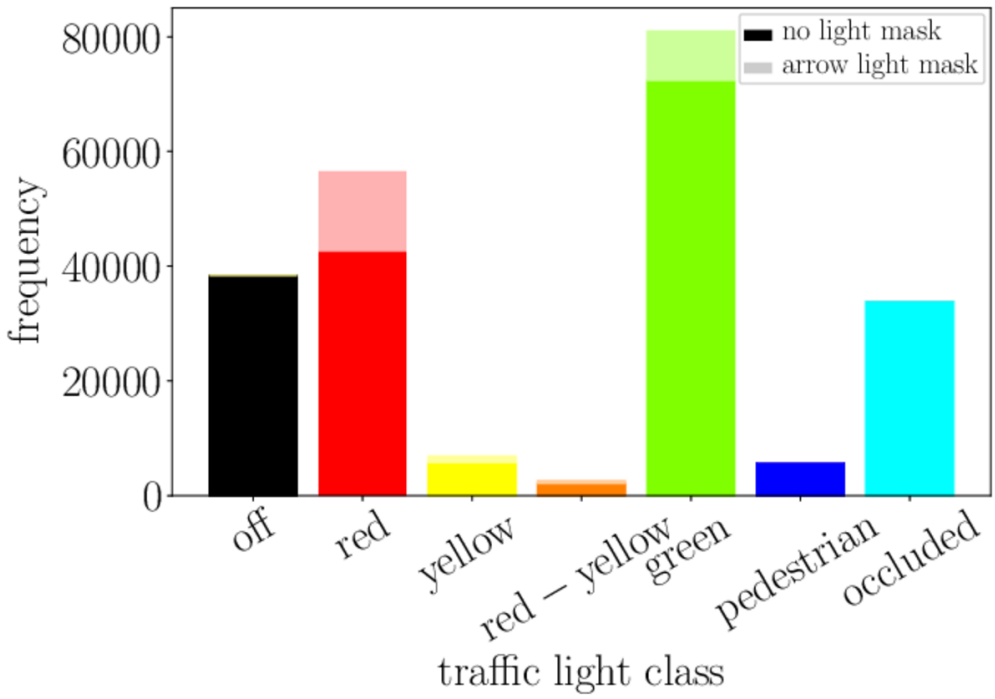
The graphic below shows the distribution of the label width provided by the DLTD dataset. One can clearly see, that small width objects are overrepresented in the dataset in contrast to bigger objects, which amplifies the negative effect of the weaker performance on smaller objects on the overall performance metrics even more.



Source: https://www.uni-ulm.de/fileadmin/_processed_/3/4/csm_label_resolution_daimler-eps-converted-to_8dac7b0b07.png (https://www.uni-ulm.de/fileadmin/_processed_/3/4/csm_label_resolution_daimler-eps-converted-to_8dac7b0b07.png) (08.07.2021)

9.4 Dataset imbalance - traffic light state

As the dataset provides real-life data, the dataset and the respective annotations mirror the real world. As a traffic lights typical `state` is either to be **green** or **red**, these states are naturally overrrepresented in contrast to e.g. traffic lights in state **yellow** or in state **red-yellow**. The lack of training data for some classes will lead to better recognition of e.g. **green** and **red** traffic lights in contrast to other states. This hypothesis is backed up by the statistics provided by the DLTD dataset, as depicted in the image below.



Source: https://www.uni-ulm.de/fileadmin/_processed_/b/9/csm_class_bar_daimler-eps-converted-to_7e064b9a30.png (https://www.uni-ulm.de/fileadmin/_processed_/b/9/csm_class_bar_daimler-eps-converted-to_7e064b9a30.png) (08.07.2021)

10 - Conclusion

With the project on hand, a sophisticated approach on traffic light detection is presented. In contrast to previous works, which typically treat the problem of detecting traffic lights solely as a three or four class problem based on the traffic lights `state` (**red, green, etc.**), the project on hand has a highly exact label concept. For the project on hand, additional to the traffic lights `state` , also the `pictogram` (**circle, arrow_straight, arrow_left, etc.**) is considered and differenciated as seperate classes. Furhermore, the network is capable of detecting not only vehicle-relevant traffic lights, but also pedestrian and bicycle relevant traffic lights. In total, this results in a **35 class differenciating network**. Training, evaluation and test is based on the DLTD dataset containing over **114078 annotated objects within the training partition**.

The test results on a test partition of the DLTD dataset containing **6226 images and 24602 annotated objects** show pleasing results. Especially close to the traffic lights, the detection and classification is exact and reliable. For smaller objects, while the detection of objects is still satisfing the classification performance is significantly reduced. However, misclassifications are mainly based on the `pictogram` of the traffic light, while the `state` is still correctly classified. While this results in relatively low overall Precision and Recall values due to extremely high number of small objects within the dataset, the performance for medium and large objects is satisfiing with much higher Precision and Recall. To further enhance the networks output for real world systems, the traffic lights can be tracked during the approach in a postprocessing chain.

For further improvement, mainly the quality of the labeled data needs to be improved. While the used dataset provides a sophisticated label concept, especially very small labels need to be evaluated and sorted by hand in order to achieve a overall better performance on small objects. For an application in a real life vehicle, additional data from the target camera hardware is needed to reduce the domain gap. A proove of concept for a real-life application of the project on hand in the HHN Passat is given in **chapter 11** of this notebook.

11 - Cross project work - integration of Nico Hessenthalers traffic

sign detection

> **Remark 1:** This chapter is related to cross project work. Contents such as images or code snippets are shared between Nico Hessenthalers report and the report on hand.

> **Remark 2:** This chapter is not presenting the cross project work in detail but much rather discussing the outcome and general approach.

11.1 Approach and setup

The objective of the cross project work is to merge the detection of traffic lights described in this notebook with the detection of traffic signs developed by Nico Hessenthaler in order to realize autonomous driving functions on the vehicle. For this task, the latest checkpoint of the traffic light detection is used as a base checkpoint. Based on this checkpoint, a new training using the manually labeled images from the HHN Passat and the GTSDB dataset containing images of of traffic signs is initiated. The GTSDB dataset is filtered, so that only images containing solely traffic signs are included in the training. In total, the cross project work contains **79** classes of traffic lights and signs.

The combined `labelMap.pbtxt` file contains the previous **35** classes for traffic lights and **44** classes for traffic signs, resulting in a total of **79** classes:

```
item {
    id: 1
    name: 'front-not_occluded-circle-not_reflected-yellow'
}

item {
    id: 2
    name: 'front-not_occluded-arrow_right-not_reflected-red_yellow'
}

item {
    id: 3
    name: 'front-not_occluded-arrow_straight_left-not_reflected-yellow'
}

item {
    id: 4
    name: 'front-not_occluded-arrow_straight_left-not_reflected-green'
}

item {
    id: 5
    name: 'front-not_occluded-arrow_straight-not_reflected-yellow'
}

item {
    id: 6
    name: 'front-not_occluded-bicycle-not_reflected-red_yellow'
}

item {
    id: 7
    name: 'front-not_occluded-arrow_right-not_reflected-red'
}

item {
    id: 8
    name: 'front-not_occluded-pedestrian_bicycle-not_reflected-off'
}

item {
    id: 9
    name: 'front-not_occluded-pedestrian-not_reflected-off'
}

item {
    id: 10
    name: 'front-not_occluded-circle-not_reflected-green'
}

item {
    id: 11
```

```
        name: 'front-not_occluded-bicycle-not_reflected-red'
    }

    item {
        id: 12
        name: 'front-not_occluded-bicycle-not_reflected-green'
    }

    item {
        id: 13
        name: 'front-not_occluded-pedestrian_bicycle-not_reflected-red'
    }

    item {
        id: 14
        name: 'front-not_occluded-pedestrian-not_reflected-green'
    }

    item {
        id: 15
        name: 'front-not_occluded-circle-not_reflected-red_yellow'
    }

    item {
        id: 16
        name: 'front-not_occluded-circle-not_reflected-red'
    }

    item {
        id: 17
        name: 'front-not_occluded-arrow_left-not_reflected-green'
    }

    item {
        id: 18
        name: 'front-not_occluded-arrow_straight-not_reflected-red_yellow'
    }

    item {
        id: 19
        name: 'front-not_occluded-pedestrian_bicycle-not_reflected-green'
    }

    item {
        id: 20
        name: 'front-not_occluded-arrow_left-not_reflected-red'
    }

    item {
        id: 21
        name: 'front-not_occluded-arrow_straight_left-not_reflected-red'
    }
```

```
item {
    id: 22
    name: 'front-not_occluded-pedestrian_bicycle-not_reflected-yellow'
}

item {
    id: 23
    name: 'front-not_occluded-pedestrian-not_reflected-yellow'
}

item {
    id: 24
    name: 'front-not_occluded-pedestrian-not_reflected-red'
}

item {
    id: 25
    name: 'front-not_occluded-bicycle-not_reflected-off'
}

item {
    id: 26
    name: 'front-not_occluded-pedestrian_bicycle-not_reflected-red_yellow'
}

item {
    id: 27
    name: 'front-not_occluded-arrow_straight-not_reflected-red'
}

item {
    id: 28
    name: 'front-not_occluded-bicycle-not_reflected-yellow'
}

item {
    id: 29
    name: 'front-not_occluded-arrow_left-not_reflected-yellow'
}

item {
    id: 30
    name: 'front-not_occluded-arrow_straight-not_reflected-green'
}

item {
    id: 31
    name: 'front-not_occluded-arrow_right-not_reflected-green'
}

item {
```

```
    id: 32
    name: 'front-not_occluded-arrow_right-not_reflected-yellow'
}

item {
    id: 33
    name: 'front-not_occluded-arrow_left-not_reflected-red_yellow'
}

item {
    id: 34
    name: 'front-not_occluded-vehicle_relevant-not_reflected-off'
}

item {
    id: 35
    name: 'front-not_occluded-pedestrian-not_reflected-red_yellow'
}

item {
    id: 36
    name: 'animals'
}

item {
    id: 37
    name: 'bend'
}

item {
    id: 38
    name: 'bend_left'
}

item {
    id: 39
    name: 'bend_right'
}

item {
    id: 40
    name: 'construction'
}

item {
    id: 41
    name: 'cycles_crossing'
}

item {
    id: 42
    name: 'danger'
```

```
}

item {
    id: 43
    name: 'give_way'
}

item {
    id: 44
    name: 'go_left'
}

item {
    id: 45
    name: 'go_left_or_straight'
}


item {
    id: 46
    name: 'go_right'
}

item {
    id: 47
    name: 'go_right_or_straight'
}

item {
    id: 48
    name: 'go_straight'
}

item {
    id: 49
    name: 'keep_left'
}

item {
    id: 50
    name: 'keep_right'
}

item {
    id: 51
    name: 'no_entry'
}

item {
    id: 52
    name: 'no_overtaking'
}
```

```
item {
    id: 53
    name: 'no_overtaking_trucks'
}

item {
    id: 54
    name: 'no_traffic_both_ways'
}

item {
    id: 55
    name: 'no_trucks'
}

item {
    id: 56
    name: 'pedestrian_crossing'
}

item {
    id: 57
    name: 'priority_at_next_intersection'
}

item {
    id: 58
    name: 'priority_road'
}

item {
    id: 59
    name: 'restriction_ends'
}

item {
    id: 60
    name: 'restriction_ends_speed_limit'
}

item {
    id: 61
    name: 'restriction_ends_overtaking'
}

item {
    id: 62
    name: 'restriction_ends_overtaking_trucks'
}

item {
```

```
    id: 63
    name: 'road_narrows'
}

item {
    id: 64
    name: 'roundabout'
}

item {
    id: 65
    name: 'school_crossing'
}

item {
    id: 66
    name: 'slippery_road'
}

item {
    id: 67
    name: 'snow'
}

item {
    id: 68
    name: 'speed_limit_100'
}

item {
    id: 69
    name: 'speed_limit_120'
}

item {
    id: 70
    name: 'speed_limit_20'
}

item {
    id: 71
    name: 'speed_limit_30'
}

item {
    id: 72
    name: 'speed_limit_40'
}

item {
    id: 73
    name: 'speed_limit_50'
```

```
}

item {
    id: 74
    name: 'speed_limit_60'
}

item {
    id: 75
    name: 'speed_limit_70'
}

item {
    id: 76
    name: 'speed_limit_80'
}

item {
    id: 77
    name: 'stop'
}

item {
    id: 78
    name: 'traffic_signal'
}

item {
    id: 79
    name: 'uneven_road'
}
```

Initially, the data recorded on target Hardware in the HHN Passat need to be labeled manually. The data is recorded on a route decribed in **chapter 2.2**, while an image is taken each second. In total, **3595** images are then analyzed and labeled manually for relevant traffic signs and traffic lights. In total, **910** images with **3153** labeled objects have been created for the cross project using the tooling presented in chapter **chapter 3.2**. The complete training and validation data is defined as follows:

- GTSDB training data part 1: **510** images with **697** labels (traffic signs only)
- GTSDB training data part 2: **136** images with **258** labels (traffic signs only, former validation dataset)
- Own labeled training data: **707** images with **2447** labels
- Own labeled validation data: **203** images with **706** labels
- **Total: 1556** images with **4108** labels

Obviously, the dataset is quite small compared to the DLTD dataset used to train the initial network for the traffic light detection. Due to the high amount of classes, the amount of labeled data is normally not enough to reach acceptable performance. However, as a pretrained checkpoint for traffic light detection is used as basis, the amount of data can be sufficient for a proof of concept using transfer learning. However, to achieve robustness, a lot more labeled data is needed. Due to lack of time and tight schedule labeling of more data is not possible.

> **Remark:** The presented resuls are in "proof of concept" state. More data is needed for sophisticated performance.

On this data, the same process as presented within this notebook is applied. The data is converted to the `tfrecord` format and together with a `pipeline.config` file, a training can be started. The pipeline config is very similar to the ones presented in **chapter 6.2**. Dropout and data augmentations are used together with the ADAM optimizer. However, as the training is applied on a personal PC, only a batch size of 1 can be applied.

11.2 Results

Below, images of the networks predictions on the left and the ground truth data on the right are shown. The performance of the predictions is working very well in many scenes. Traffic lights and traffic signs are detected within the same frame simultaniously, enabeling advanced autonomous driving functions. The performance on traffic lights is comparable to the performance in the solo trained network. The first picture shows a typical driving situation when approaching a red traffic light with the additional traffic sign `go_straight` . In the second picture, also pedestrian traffic lights are detected and classified correctly. The last picture shows a correct classification of a traffic light with the `pictogram` "arrow_straight".

All sources: Own creation by Tim Dettling & Nico Hessenthaler (21.07.2021)

In contrast to the good scenes, the limitation from the solo training described in **chapter 9** are still remaining and observable in the test data of the cross project. The following pictures show misclassifications of traffic lights by their `pictogram` from a higher distance and false negatives on very small objects. Still, the network also predicts the traffic lights `state` correctly for all objects and also detects traffic signs next to the traffic lights.

All sources: Own creation by Tim Dettling & Nico Hessenthaler (21.07.2021)

11.3 Conclusion

In conclusion, the combined training shows promising results for the simultanious detection of traffic lights and traffic signs and by this, the proof of concept is achieved. However, it becomes obvious that the cross projects combined training suffers from the lack of training data of the target hardware. Combined with the very high number of classes, this results in unstable performance.

To overcome these problems in the future, firstly more data can be labeled from the Passat. While this is great effort and a cumbersome process, a sufficient amount of label data would greatly improve the training results. Additionally, measures to overcome the domain gap between publically available datasets and the Passats 120° fisheye camera can be applied, e.g. rectifing the images before applying them to the networks training. Furthermore, the number of used classes is quite high. Reducing the classes by e.g. merging similar classes or excluding non-relevant classes, the number could drastically reduce. Finally, other training configurations can be investigated.

For real life application, the network will be included in the ROS-based system on the HHN Passat and tested in real world and real time.