BABEŞ-BOLYAI UNIVERISTY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

DIPLOMA THESIS

Automated traffic light detection using convolutional neural networks

Supervisor:
Conf. Dr. Craciun Florin

Author:
Dobrea Andrei

2020

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI
INFORMATICĂ

LUCRARE DE LICENŢĂ

Sistem automat pentru detecţia semafoarelor folosind reţele neuronale convolutive

Conducător ştiinţific:
Conf. Dr. Craciun Florin

Absolvent:
Dobrea Andrei

2020

Abstract While prior work on object detection used separate detectors and classifiers, the YOLO system proposes a new approach. It frames object detection as a regression problem, allowing a single neural network to predict both bounding boxes and class probabilities from full images in one evaluation. This leads to a detection pipeline that is faster and can be optimized end-to-end directly on detection performance.

This paper presents **my own implementation of the YOLO system** from the ground up. In the beginning it closely followed the description in the publication, but was later modified to better fit the problem of traffic light detection. The big YOLO architecture is too resource intensive and lacks in prediction speed, while also having problems with instability at the beginning of the training.

To solve these problems, **my contributions** were: designing and implementing a new lighter, faster network architecture and improving the current loss function design. Also, I have created a special color selection system for when there is an uncertainty in traffic light classification.

By the use of the detection and classification capabilities of both YOLO and the custom color selector, the system is capable of predicting traffic light location and state with an incredibly good precision and in real time.

The Retinanet-101-500[1] network achieves 53.1 mAP at 11 frames per second. The SSD321[2] network achieves 45.4 mAP at 16 frames per second. My system is able to achieve **57.2 mAP**, while running at **20 frames** per second, therefore outclassing some of the best state-of-the-art designs both in terms of speed and accuracy.

Contents

1 Introduction

As car manufacturing gets cheaper by day, more and more people have access to powerful cars. The automotive industry has exploded in the past few years and studies have shown that only in the USA there are more than 800 cars per 1000 people [3]. With the recent developments in camera and computer technology, the world is slowly but surely approaching the feat of truly autonomous driving. Besides other topics such as pedestrian and sign detection, autonomous cars must be capable of perceiving traffic lights and recognizing their current states in order to share the streets with human drivers.

Most of the time, humans can easily identify the relevant traffic lights, but with the illusion of safety modern cars give, people tend to pay less attention to driving, leading to an increasing number of lives lost each year. A study [4] done by the AAA Foundation for Traffic Safety shows that more than 28% of crash deaths that occur at signalized intersections are the result of a driver running through a red light. Each day, in USA only, this leads to two human deaths.

Moreover, although not as severe, car crashes come with another complication, the economic one. The National Highway Traffic Safety Administration in USA discovered that car crashes make up for yearly losses of $76 billion in property damage, $57.6 billion in productivity losses for employment and household, $23.3 billion in medical care and on-going rehabilitation costs [5]. Therefore automated traffic light detection is of utmost importance.

One of the big problems is generalization. Traffic lights come in many different quantities, positions, shapes, sizes, and layouts, each of them being

different from region to region. To deal with this issue, a common solution for autonomous cars is to integrate recognition with prior maps. However, an additional solution is required for the detection and recognition.

With a deep learning based approach the global differences of traffic light position and shape are easily overcome. One must not write specific algorithms for each region of the world, but simply collect examples of different types of traffic lights in the area the car will be driving. Another problem that is solved by using machine learning is that of distance estimation. Object detectors predict also a high accuracy confidence for each detection which, in turn, allows a good estimation of distances. The better this estimation is, the closer can the car match the other data points. For example, this kind of confidence-based distance estimation can tell which side of the intersection a traffic light is.

As a consequence, deep learning techniques have showed great performance and power of generalization. Motivated by their advantages, recent works leveraged some state-of-the-art deep detectors to locate and further recognize traffic lights from 2D camera images. Some of them use the approach of extracting the detected traffic light first, then running a second classifier on it. That approach provides flexibility, however, it usually comes with the disadvantage of added pipeline complexity and computation. This leads large development costs and a slower algorithm.

The project described in this paper aims to provide a detection and classification system that is easier to optimize while also being faster. The system will then be used to detect the traffic lights from a wide-angle video feed and recognize their color.

To achieve the increase in speed and optimization time, the project is based on the YOLO detection system. This new approach to object detection frames it as a regression problem to spatially separated bounding boxes and associated class probabilities. A single convolutional neural network predicts bounding boxes and class probabilities directly from full images in one evaluation, thus providing a fast pipeline that can be optimized end-to-end directly on detection performance. While certainly having multiple benefits compared to other detection systems, YOLO also has some drawbacks. The architecture is very deep, causing it to be resource intensive and also affecting prediction time. Furthermore, the whole network is unstable at the beginning of the training, requiring a small learning rate that is increased after the first epochs, only to decrease in the end.

The project started as an **my implementation from ground up** of the YOLO system, then the major problems were rectified. I designed and **implemented a new, lighter, faster architecture (3.1)** and **created a custom loss function (3.1)** in order to improve the stability of the network in at beginning of the training. Also, to increase the reliability of the system when it did not come to a unanimous result regarding the traffic light color, **I implemented a specially designed color selection system (3.1)**.

The results are satisfying. In the end, I have implemented from ground up a fast and robust system, capable of predicting traffic light location and state. It has a precision comparable to other state-of-the-art object detectors, but improves on their performance by being real time. It achieves a mAP(mean Average Precision) metric of 57.2 while being able to run at 20 frames per second on only a laptop GPU.

2 Theoretical background

2.1 Deep learning - Discover the power of CNN

The main use case of the application is to detect traffic lights from the video feed from a car and recognize their color. We can achieve this using artificial intelligence, more precisely Convolutional Neural Network (CNN).

CNN are neural networks that work with images as input and they are widely used for image recognition, image classifications, object detection, etc. The output can be in several forms:

- for image recognition the output is a class (e.g. Dog, Cat,etc.) telling us if the image contains an object of that class.
- for object detection and localization also specifies the bounding box of the object of a certain class
- for semantic segmentation the output is a pixel-wise map telling us of which class does every pixel in the image belongs to

We will focus on object detection, localization and classification because our target is to detect traffic lights on the road and then classify the traffic lights based on color.

Human neural networks Human brains have about 100 billion neurons, each neuron may be connected to as many as 10 000 other neurons, resulting in 1000 trillion synaptic connections that can support electro-chemical signalling.[6] From the early childhood we learn to distinguish different kinds

of features from what we see and also develop the ability to recognize objects, even though they are not necessarily identical because in fact we learn how to recognize basic features and patterns with them. We develop a set of weights that according to the input image activate some of the neural links that gives us the final output: what is each object we see and where it is located. The learning process of a human being consists in labeling the patterns we recognize with words that describe them using assistance from older people who already went through this learning process.

Artificial neural networks Similarly, the artificial neural networks are build on the same principles, but instead of having biological components they have numerical values and functions. The electro-chemical signal is the activation function, the synaptic connections are called weights and they are a set of numerical values that represent the connection between the neurons, the neurons itself represent nodes in the network.

Image representation Usually an image has three channels: Red, Green and Blue (RGB), each pixel having a value from 0 to 255 for each of those three colors. So an image with width w and height h can be defined as a multidimensional matrix with size w x h x 3.

Filter A filter is a small square matrix or a kernel, usually odd sized, 3x3 or 5x5, that has some numerical values which are used to pass over the input image and extract specific features according to the type of the kernel (e.g. gaussian filter for blurring).

Convolution A convolution is an mathematical operation between an input image and a filter. The filter is passed over the entire image and at each pixel we apply the dot product between the filter and the section over which the filter passes.

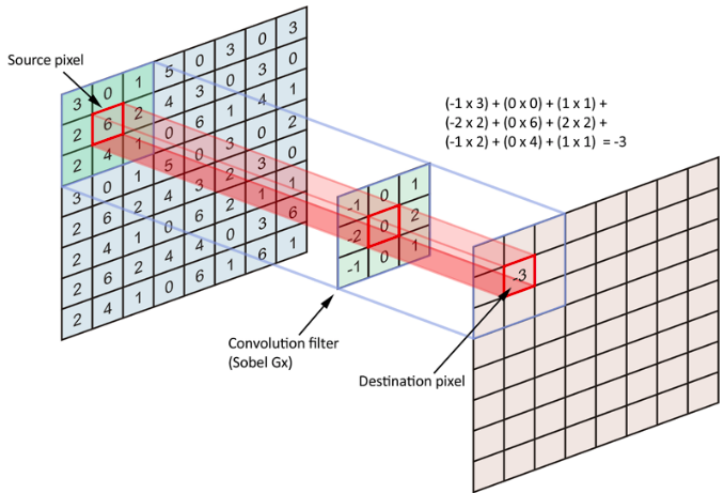


Figure 1: Convolution filter
Source: [7]

Stride The step which is used to move the filter over the image (in number of pixels) is called stride. If the stride is 1 then we move the kernel column by column and line by line. If the stride is 2 the we skip one column and one line at each step.

Padding In order to preserve the image dimensions regardless of the filter size we may add zero padding to the image.

Pooling Having many layers and large images the number of parameters and computation is huge. So we can reduce the size of the images and

keep the essential information using spacial pooling (also called subsampling or downsampling). There are several types of spacial pooling:

- Max Pooling : take the max value from the kernel
- Sum Pooling : sum up the values from the kernel
- Average Pooling : average the values from the kernel

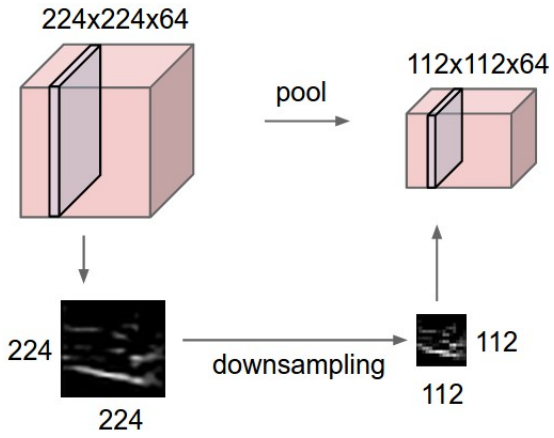


Figure 2: Convolution filter

Source:[8]

Upsampling In some cases, after downsampling reduces the size of an image, a bigger image size would be required for further processing. This problem is solved by the use of upsampling. It increases the size of an image by generating new pixels to fill the void. This can be done by the use of advanced mathematical procedures or, more common, by simply duplicating existing pixels.

Batch Normalization [9] When a CNN's input distribution changes (covariate shift) it may lead to the behavior of machine learning algorithms changing. This is especially harmful if the train/test set images are different. The basic idea behind batch normalization is to limit covariate shift by transforming the inputs in each layer to be zero mean and unit variance, effectively normalizing the activations. Furthermore, this enables the use of higher learning rates, greatly accelerating the learning process, while also having a slight overfit prevention effect.

Convolutional Neural Network CNN is a series of convolution layers, pooling layers, fully connected layers and at the end apply softmax function to approximate the probability of an object to belong to a class.

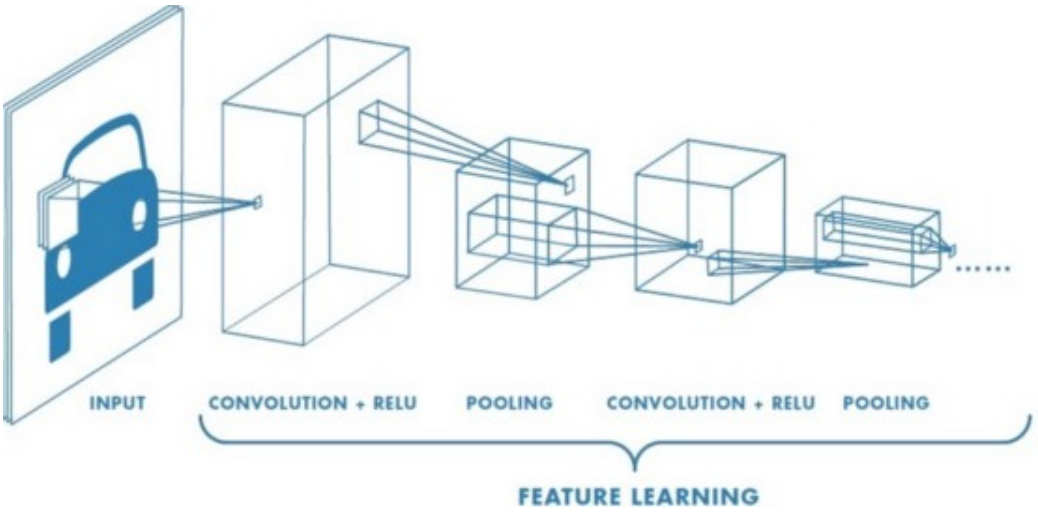


Figure 3: CNN Layers
Source:[10]

2.2 Unsupervised learning - K-means clustering

K-Means clustering is an unsupervised learning algorithm that finds a fixed number K of clusters in a set of data. A cluster is a group of data points that are grouped together due to similarities in their features.

Centroids When using a K-Means algorithm, a cluster is defined by a centroid, which is a point at the center of a cluster. Every point in a data set is part of the cluster whose centroid is most closely located.

Algorithm K-Means starts by randomly defining K centroids. From there, it works in iterative steps to perform two tasks. It assigns each data point to the closest corresponding centroid, using the standard Euclidean distance, then, for each centroid, calculates the mean of the values of all the points belonging to it. The mean value becomes the new value of the centroid.This process is repeated over and over until there is no change in the centroid values, meaning that they have been accurately grouped.

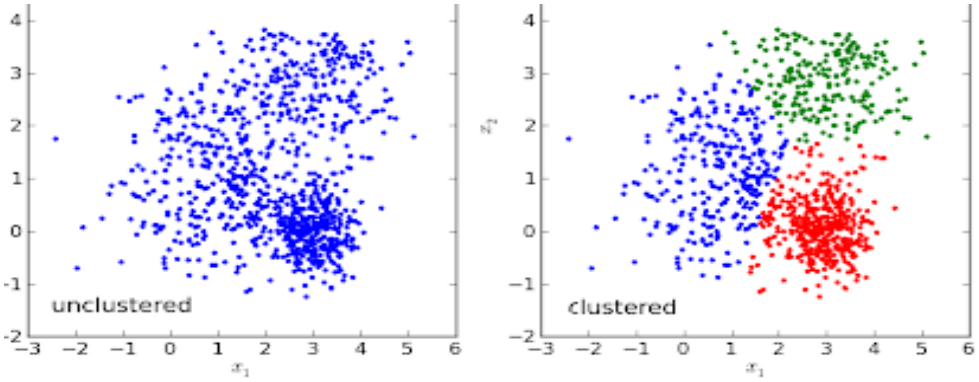


Figure 4: K-means clustering
Source:[11]

2.3 YOLOv3 algorithm

Residual blocks [12] In nature, for optimisation purposes, some neuronal paths can be simply skipped by the use of another connection. Residual blocks mimic this feature, allowing the network to bypass entire layers by use of an extra identity connection. This makes the network more flexible and easier to train: it allows it to be simple, while also giving it the potential to increase its depth if the problem requires it.

Anchors To predict the bounding boxes for the detected objects, YOLO uses an anchor-based system. Each bounding box width and height is predicted in regard to an assigned anchor. By selecting a bigger or smaller anchor the network can finely tune the detection of different-sized objects. Before the training, k-means clustering is run on the train data to find the appropriate anchors, which are then grouped by size.

Network design Yolo v3 is a fully convolutional network. It uses batch normalization between each layer and a number of residual connections. The detection is done by applying 1 x 1 detection kernels on feature maps of three different sizes at three different places in the network.

The input image is passed through an number of convolutional, batch normalization and pooling layers. Then a number of 1x1 kernels are applied to get the prediction for the first scale. The resulting image is too small to be later processed, so an upsampling of 2x is applied. This procedure is repeated two more times, totaling to three prediction scales. The first scale detects large objects only, the second medium only, and the third, small objects only.

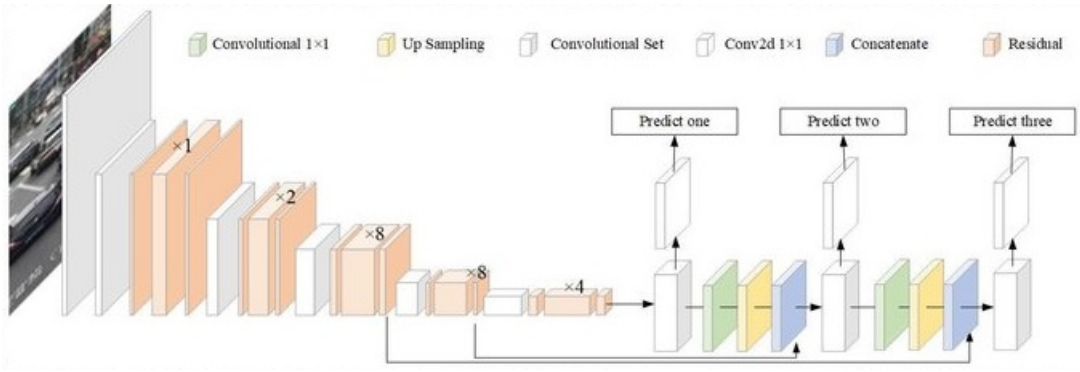


Figure 5: Yolo v3 architecture
Source: [13]

How it works [14] For each scale, the image is split into 3 grids, each one finer than the last. Because the predictions are made as percents of the grid cell size, the coarse grid is better at predicting bigger objects, the middle one for medium objects and the fine one for small objects.

For each grid cell in each of the 3 grid sizes YOLO predicts:

- coordinates of the bounding box center
- bounding box width and height
- prediction confidence
- a conditional class probability for each of the C classes

Therefore, the output of each scale in the network is a $grid\_width * grid\_height * (B * 5 + C)$ tensor.

Conditional class probability reflects if the detected object belongs to one of the given classes (one probability per class for each cell). YOLO's output has a shape of (S, S, B x 5 + C) . So the main idea of YOLO is to implement

a CNN that reduces the image to a (S, S, B x 5 + C) tensor and then apply linear regression using fully connected layers to obtain the desired tensor.

Bounding Box prediction A box has five elements: x, y, width, height and box confidence score. The confidence represents the likeliness of the box to contain an object and how accurate the bounding box fits the object. The bounding box is represented in regard to:

- the cell it is placed in (cx, cy)
- the offset from the upper left corner of the image (bx, by)
- the actual width and height (bw, bh)

The box width and height are calculated as a percentage of the width and height of the anchor assigned to the box (pw, ph). The network predicts 4 coordinates for each bounding box (tx, ty, tw, th).

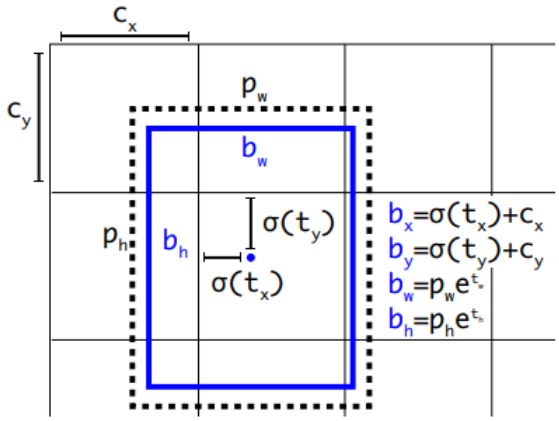


Figure 6: Bounding box encoding
Source: [13]

The formulas for calculating the dimensions in pixels (bx, by, bw, bh) from the ones predicted by the network (tx, ty, tw, th) by use of the anchor's dimensions (pw, ph) can be seen in the figure (6).

Loss During training YOLO uses sum of squared error loss for the bounding box coordinates and for the confidence. For the class predictions it uses binary cross-entropy.

Non-max suppression To have the final predictions the network filters out all bounding boxes with low box confidence scores (e.g. less than 0.25) and applies non-maximal suppression for the other ones to eliminate duplicates. Non-max suppression takes into consideration the predicted bounding boxes for each class. For each group of boxes having an IOU (intersection over union) bigger than a given threshold, it keeps only the one with the biggest confidence, deleting the others.

2.4 Data augmentation

Because of its ability to fit complex systems and its tendency to overfit smaller ones, deep learning requires a large dataset. In order to improve the already existing datasets and increase their size, data augmentations are used. This is accomplished by means of image processing. A general image processing operator is a function that takes one or more input images and produces an output image.

Image transforms can be seen as:

- Point operators (pixel transforms)
- Neighborhood (area-based) operators

Brightness and contrast adjustments This implies a pixel transform in order to adjust the image.In this kind of image processing transform, each output pixel's value depends on only the corresponding input pixel value plus, potentially, some globally collected information or parameters.

Two commonly used point processes are multiplication and addition with a constant: $g(x) = \alpha x + \beta$.

The parameters α and β are often called the gain and bias parameters. Sometimes these parameters are said to control contrast and brightness respectively.

A more conveniently we can write the expression as $g(i, j) = \alpha * f(i, j) * \beta$ where i and j indicates that the pixel is located in the i-th row and j-th column.

Figure 7: Brightness and contrast adjusted image
Source: [15]

As seen in the figure (7) the overall brightness has been improved but the clouds are greatly saturated due to the numerical saturation of the method.

Gamma correction It implies a pixel transform in order to correct the brightness of an image by using a non linear transformation between the input values and the mapped output values:

$$O = \left(\frac{I}{255}\right)^{\gamma} * 255$$

As this relation is non linear, the effect will not be the same for all the pixels and will depend to their original value.

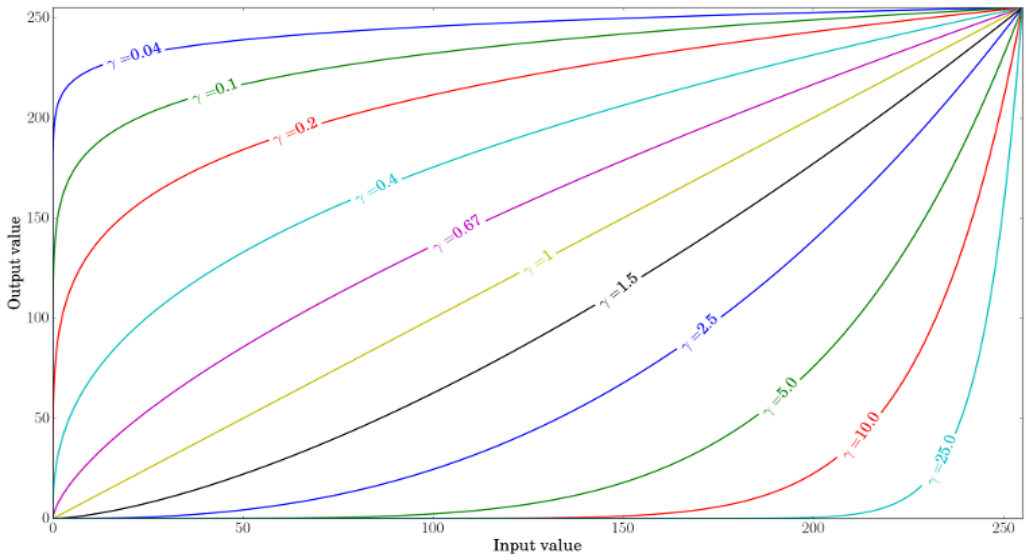
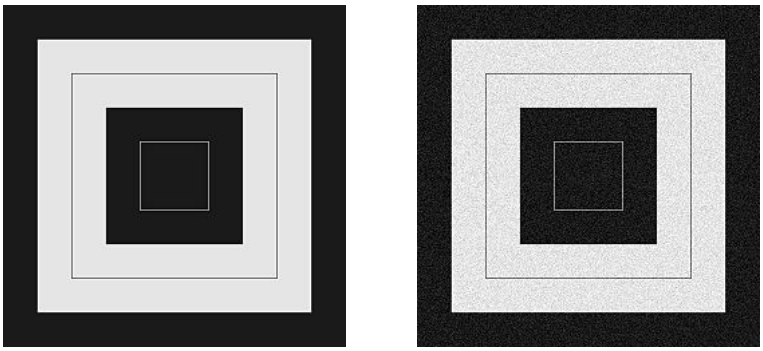


Figure 8: Gamma correction graph
Source: [15]

As seen in the figure (9), the gamma correction should tend to add less saturation effect as the mapping is non linear and there is no numerical saturation possible as in the previous method.

Figure 9: Gamma corrected image
Source: [15]

Gaussian noise It is a statistical noise having a probability density function equal to normal distribution, also known as Gaussian Distribution. From the gaussian distribution, a random image is created and then added to the original image to generate this noise.



(a) No noise (b) Gaussian noise

Figure 10: Application of gaussian noise

The magnitude of Gaussian Noise is directly proportional to the Standard Deviation. It is also called electronic noise because it arises in amplifiers and image sensors, being caused by poor illumination and high temperature or simply interference.

2.5 Used technologies

To help with implementation time and to avoid boiler-plate code writing, a number of libraries are used.

NumPy[16] which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed.

Tensorflow[17] is an open source library which facilitates the computation with tensors (multidimensional data arrays) using data flow graphs. It is used for both research and production in machine learning algorithms such as neural networks. The library is flexible and easy to use in implementation of complex deep learning networks that can be deployed on one or many CPUs and GPUs or on mobile devices without having to change the implementation.

Keras[18] is a open source library for designing neural networks in python on top of Tensorflow or Theano. It is great for experiments because it offers a fast way to implement a neural network that is user-friendly, modular and extensible.

OpenCV[19] is a cross-platform library which facilitates the development of real-time computer vision applications. It mainly focuses on image processing, video capture and analysis including features such as object detection.

Matplotlib[20] is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter and Qt.

SciPy[21] is a free and open-source Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

Scikit-learn[22] is a free software machine learning library for the Python programming language.[3] It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

3 System design and implementation

As more and more car manufacturers are including on-board cameras in their vehicles, the solution to this problem would be an automatic traffic light detection system. This system should capable of detecting the traffic lights in an intersection and their color. In case the driver fails to slow down at a red light, the system is able to give a warning or take control of the car and safely stop it before the cross-walk, thus reducing the danger for both the driver and the pedestrians.

3.1 System design

The system needs several components:

- deep CNN that recognizes traffic lights in the input video
- color selection system for solving out uncertain color predictions
- system for testing the pipeline on a laptop
- system for detecting when the driver is about to pass a red light

The entire system was implemented from ground up, so in order to create a CNN better suited to the problem, to make the code easier to understand and to better organize it, I made several design decisions which are presented next.

Bounding box representation In order to mitigate the problem of conversion between different image shapes, the center of the image, its width and height are saved as percentages of the image size. Therefore, regardless of further scaling, the coordinates can be computed by simple multiplications.

Network architecture One of the big problems of the YOLO v3 architecture is its depth. Because of it, even with some of the best graphics cards on the market, the network takes very long to train and occupies a substantial amount of memory. This depth is mostly useful in large scale detectors such as the ones trained for image recognition competitions. Compared to such competitions, traffic light detection is a relatively small problem and the used dataset is modest, so a very deep feature detector or classifier creates more drawbacks than the advantages it poses.

Therefore I decided to **design and implement a new architecture**:

Feature extractor As opposed to YOLO, the image is not squared because resizing it would mean that some important features might get too distorted for the network to successfully recognize them. Therefore, it uses a rectangular image of 832 by 320 pixels as input. The first half of the new architecture is inspired from the feature extractor design from YOLO v3, but it has considerably fewer layers. A big number of residual connections are used to allow the network to perform just as well, if not better, than a system with only a few number of layers. This also simplifies the learning process, allowing the model to be trained faster.

Classifier The second half of the classifier takes the TinyYolo [23] classifier and improves on it by increasing the number of filters and by adding three prediction sizes, rather than two. This allows a finer tuning of the dimensions of predicted bounding boxes.

Loss function As YOLO v3 is a fairly new network design and because it has a complex loss function, the Keras framework has not been able to implement this loss function yet. Therefore, this has to be done separately which is good because it also enables me to **improve the loss function**. I observed that at the beginning of the training, as the network predictions were incredibly poor, the loss function was so high that the system became unstable and had a strong tendency to diverge. Therefore, I have come up with the idea that at first, it would be good to gently guide the network towards better predictions. This is accomplished by training the network to first approximate an area where the bounding boxes should be placed rather than learn the exact, perfect position from the beginning. To implement this, the model is made to ignore the loss of the bounding boxes that have an IOU with the ground truth bigger than a certain threshold. This lowers the loss in general, while also allowing the system to learn and speeding up the convergence process.

Color selector In a number of cases, the system might predict two classes with similar confidence, so it cannot decide on what to choose. To solve this, **I designed a special color selection system**. It crops the bounding box predicted by the network, then runs a K-means clustering algorithm on the RGB values of the pixel colors. Two centroids are chosen and initialized

to the RGB values corresponding to the black and white colors. After the clustering, the centroids will represent the color values of the black box of the traffic light and the lighter colored light. The values corresponding to the light are chosen and the error between the RGB encoding and the ones of the two previously predicted classes is computed. The class with the smallest error will be chosen.

Data generation Deep learning requires a large dataset compared to other machine learning approaches, so storing, loading and processing this data poses a big problem. To solve it, the preprocessed bounding boxes will be stored as sets of coordinates in object files. Also, the images required will be downscaled to the network requirements and loaded into memory only when needed. The encoding of the data will be done during the training, batch by batch, each one being cleared from memory after usage. This drastically improves the memory efficiency of the application, allowing the use of less performant technology and achieving better accuracy by being able to train with bigger batch sizes.

Data augmentation Since the data is generated during the training time, it is a good decision to also do the data augmentation there. The data will be augmented by use of linear techniques such as vertical flips, translations, crops, zoom in and zoom out. Also lighting effects are used: exposure and contrast adjusting, addition of gaussian (salt-pepper) noise. Each of the parameters for the augmentations will be randomly chosen in order to ensure a better diversity.

Confidence threshold During the inference, a threshold for selecting which predictions are discarded is needed. To better choose this parameter, a useful method is the use of a ROC curve. This is also good for comparing different models, the ones with a greater area under the ROC curve being the better ones.

Class diagram The program makes use of many of the Keras classes and methods, but for better storage of data during preprocessing, some custom data holders are implemented. Also, with the used dataset, some parsing classes and algorithms are provided. However, these are mapped to some simpler classes for easier label storage and processing.

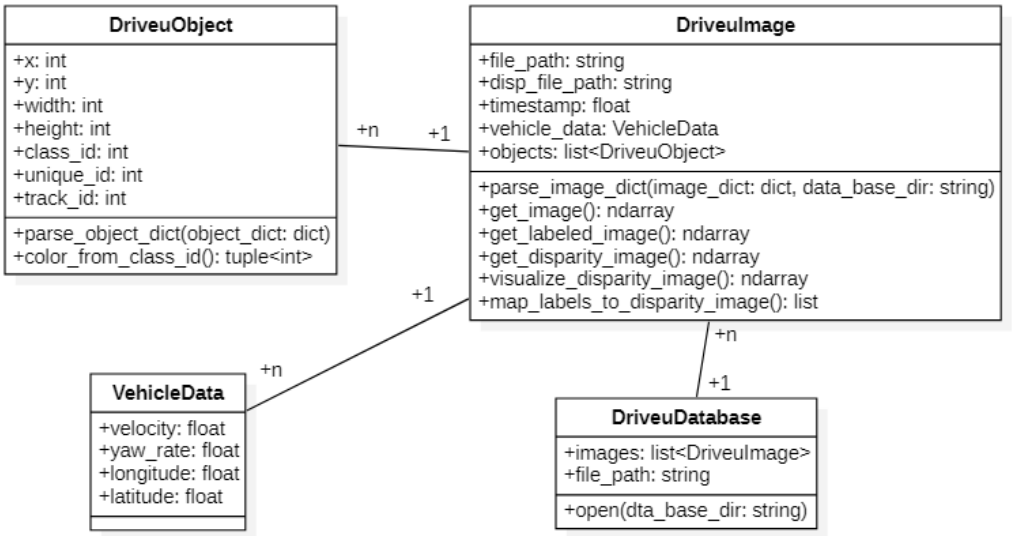
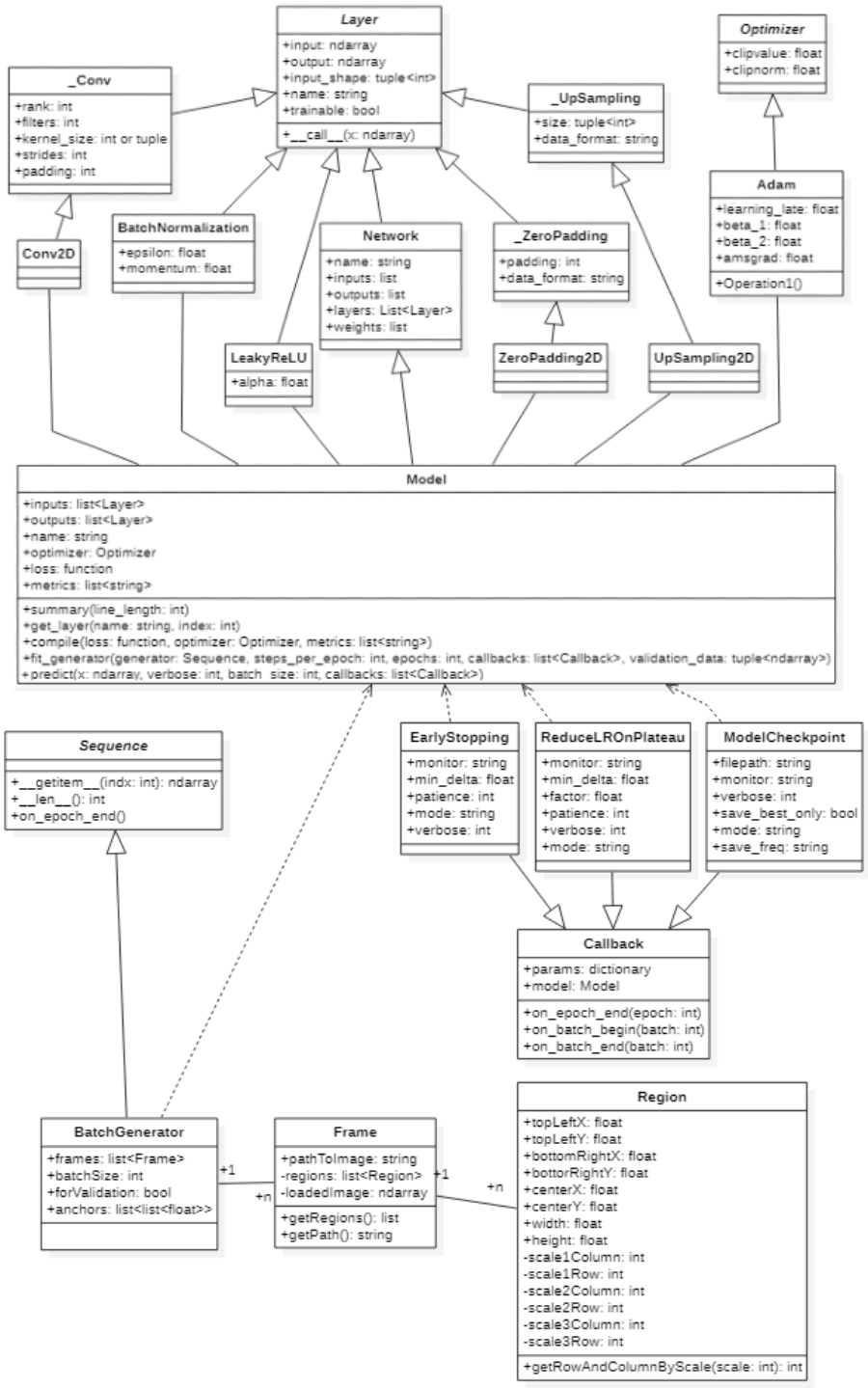


Figure 11: Driveu dataset [24] parsing classes


Layer
+input: ndarray
+output: ndarray
+input_shape: tuple<int>
+name: string
+trainable: bool
+__call__(x: ndarray)
Optimizer
+clipvalue: float
+clipnorm: float
_Conv
+rank: int
+filters: int
+kernel_size: int or tuple
+strides: int
+padding: int
_UpSampling
+size: tuple<int>
+data_format: string
Adam
+learning_late: float
+beta_1: float
+beta_2: float
+amsgrad: float
+Operation1()
Conv2D
BatchNormalization
+epsilon: float
+momentum: float
Network
+name: string
+inputs: list
+outputs: list
+layers: List<Layer>
+weights: list
_ZeroPadding
+padding: int
+data_format: string
LeakyReLU
+alpha: float
ZeroPadding2D
UpSampling2D
Model
+inputs: list<Layer>
+outputs: list<Layer>
+name: string
+optimizer: Optimizer
+loss: function
+metrics: list<string>
+summary(line_length: int)
+get_layer(name: string, index: int)
+compile(loss: function, optimizer: Optimizer, metrics: list<string>)
+fit_generator(generator: Sequence, steps_per_epoch: int, epochs: int, callbacks: list<Callback>, validation_data: tuple<ndarray>)
+predict(x: ndarray, verbose: int, batch_size: int, callbacks: list<Callback>)
Sequence
+__getitem__(indx: int): ndarray
+__len__(): int
+on_epoch_end()
EarlyStopping
+monitor: string
+min_delta: float
+patience: int
+mode: string
+verbose: int
ReduceLROnPlateau
+monitor: string
+min_delta: float
+factor: float
+patience: int
+verbose: int
+mode: string
ModelCheckpoint
+filepath: string
+monitor: string
+verbose: int
+save_best_only: bool
+mode: string
+save_freq: string
Callback
+params: dictionary
+model: Model
+on_epoch_end(epoch: int)
+on_batch_begin(batch: int)
+on_batch_end(batch: int)
BatchGenerator
+frames: list<Frame>
+batchSize: int
+forValidation: bool
+anchors: list<list<float>>
+1
+n
Frame
+pathToImage: string
-regions: list<Region>
-loadedImage: ndarray
+getRegions(): list
+getPath(): string
+1
+n
Region
+topLeftX: float
+topLeftY: float
+bottomRightX: float
+bottorRightY: float
+centerX: float
+centerY: float
+width: float
+height: float
-scale1Column: int
-scale1Row: int
-scale2Column: int
-scale2Row: int
-scale3Column: int
-scale3Row: int
+getRowAndColumnByScale(scale: int): int

3.2 Implementation details

The project is mainly divided in three parts: data preprocessing, training and inference, each of them with different subcomponents.

Data preprocessing In the used dataset, the images are of higher resolution than the ones needed for the network, in Bayer pattern and on 12 bits. The network operates with 8 bit RGB images, so the data must be firstly converted. This issue is easily solved with the use of OpenCV conversion functions and simple multiplications and divisions.

Further, the images have to be resized to the network specifications. This causes some of the old bounding boxes to have width and height of only a couple pixels, so they will not contain enough information for the network to properly learn. Therefore, the boxes are filtered to a minimum width and height and their dimensions are stored as percents of the width and height used by the network input. With the disadvantage of not being human-readable, the data is stored in an object file for achieving a better loading time.

The YOLO based CNN used requires a set of 9 anchors, 3 for each prediction size. These are generated by running K-means clustering with 9 centroids on the width and height of each bounding box previously selected from the dataset. After finding the best anchors, they are sorted according to their IOU with the image dimensions and then they are assigned to each prediction size of the network.

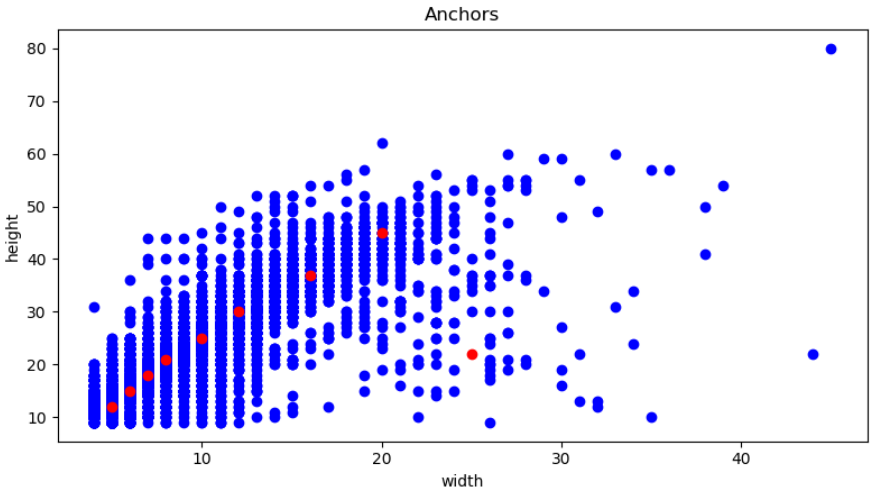


Figure 12: Anchors after running K-means

Training As the data encoding and augmentation is done during the training phase, it will be described as such.

Initially a batch of images is selected from the whole dataset in such a manner that in every epoch all the images are used, but they each step contains different images. Then the image is loaded into memory, transformed and the annotations are modified to ensure that they are correct. It is randomly decided if the image will be augmented or not and also the augmentation parameters are chosen randomly in order to provide the best variation possible. Because of this, in every step of every epoch, the network is presented with new images, always different. This greatly improves learning capabilities.

After the data augmentation is applied to the batch images, their corresponding bounding box coordinates are processed in order to bring them in the format suitable for the network training. For each box, the best fitting

anchor is calculated by doing IOU with all the anchors. Then the row and column of the cell responsible for predicting the box are calculated. Finally, the center coordinates, image width and height are encoded and the whole matrice is saved and then passed on to the training.

The loss function has to be implemented separately. It uses sum of squared error for the bounding box coordinates and binary cross entropy for the confidence and classes. In addition to this, there is a system implemented such that if a prediction has an IOU with the ground truth better than 0.7, its error is ignored. Although at first glance it seems that the final result will not be as precise, this idea significantly improves the training time and quality while also making the model more stable.

During training the dataset is split into three parts: the training set(0.7 of full set), the validation set(0.8 of full set) and test set(0.1) of full set. The data is provided to the Keras framework for training using the aforementioned batch generators with one exception: for the validation set, no augmentation is needed.

For making the training process more efficient, a set of callbacks is used. The first of them saves the model each time it has seen an improvement in the validation loss. This ensures that the model is saved just before the network starts to overfit the train set, thus ensuring the best validation result. The second callback reduces the learning rate by 25% if there was no loss improvement in the last 3 epochs. The final callback stops the training if for 10 epochs there was no improvement in the loss.

After the training, the train and validation loss history is saved for future reference and comparison.

Inference In order to predict bounding boxes, the system requires a confidence threshold to select which bounding boxes will be kept and which will be discarded. To better find this threshold, the ROC curve of the model is calculated and then displayed. To calculate it, the user selects which data set to use, then the program makes predictions on the images of that dataset. Using the ground truth confidence and the previously predicted confidence, the false positive rate, true positive rate and corresponding thresholds are calculated. These are then plotted as the ROC curve.

During the prediction functionality, the program loads the required images based on user given parameters and makes predictions on them using the trained model. Then, the predictions have to be decoded from the network-specific format.

The decoder transforms the bounding box center coordinates and its width and height from the format required by the network back to percents of image width and height. Then, it discards the boxes with low confidence, it selects the class of the remaining ones and depending on the size of image to be displayed, it transforms the bounding box coordinates to pixels.

Further, non-max suppression is run on the predicted boxes. The threshold used can be set by the user, but the default is 0.5. After suppressing the non-maximal boxes, the remaining ones are displayed over the image.

Testing the system The correctness of the data preprocessing and of the post-processing can be easily tested by running the decoder part on the preprocessed ground truths. If the results are the same, that guarantees a correct implementation.

4 Experimental validation

4.1 Grayscale dataset experiments

The first dataset used consisted of a few thousand images that were grayscale. At the beginning it was useful for a coarse tuning of the system implementation. Next, there will be presented the development stages on this dataset, together with the improvements and the results.

Debug In the beginning, after the basic structure of the YOLO v3 is implemented as in the paper, the system is tested to ensure a bug-free implementation. To check if it is able to learn, an overfit of 5 images is tried with a batch size of 1. Because of the small batch size, batch normalization is not used as it is irrelevant.

Full dataset train After the overfit succeeded, indicating that the network was implemented correctly, a training on the full dataset is tried. The anchors are imported from Darknet-53, the framework that implements YOLO v3 in the scientific publication. From the start, an observation is made: batch normalization greatly reduces the number of epochs required to converge. After the training, the results are decent, but it is clear that the model has a problem in predicting small objects.

Anchors New anchors are generated by running K-means clustering on the sizes of the annotations. Then the model is trained using the new anchors and the results for small images improve.

Custom loss The next problem to be solved was the instability of the model during the first epochs of training. Although a relatively small learning rate was used, the loss had a tendency to diverge quickly. In the YOLO system, the authors propose starting with a lower learning rate of 10^{-5} which after a small number of epochs will be increased at 10^{-3}, then to be lowered consecutively during training. However, this method increases the training time, while not always being a good divergence deterrent. Therefore, I decided that a better approach was to make the model ignore the loss of the bounding boxes that have an IOU with the ground truth bigger than 0.7. This lowered the loss in general, while also allowing the system to learn and speeding up the convergence process.

Callbacks In order to achieve a better flexibility and be able to finely tune model training, a set of Keras callbacks is used. They take control of model saving, decreasing the learning rate and stopping the training.

Transfer learning Next, I tried to use transfer learning. The network was loaded with the weights from the training done on COCO by the YOLO v3 team. Then, the classifier part of the network was frozen so it could not learn and the feature extractor was trained with a small learning rate. Further, the feature extractor was frozen and the classifier part was trained for a long number of epochs and with a normal learning rate. Although the COCO dataset contained "traffic light" as a class between the other 249, this was not enough for achieving good result through transfer learning, so the idea was discarded.

Multiple predictions on one image In an attempt to solve the problem of detecting small bounding boxes, I tried training the network on big and medium labels and then split the image for inference in 4 parts. The parts would be stacked and run through the system at once, thus not affecting prediction time. While in idea this seemed a good strategy, in practice, the model failed to detect the boxes. Also, if a box was split down the middle by the partitioning, it would always fail to predict it. This idea was also discarded.

Data augmentation As the model showed strong signs of overfitting, I implemented data augmentation. While it considerably solved the problem, it was not enough to completely eradicate it.

New architecture After thoughtful consideration, I have come to the conclusion that the cause of overfitting was the depth of the network. Therefore, a shallower architecture was required. This would also help with training and prediction time. As a result, I designed and implemented an architecture that is 25% of the initial size, while also being fully capable of fitting the dataset. This led to a big increase in inference time and quality.

ROC curve Next the ROC curve generation was done. It was used to enhance the capabilities of the inference part. This coupled with the new architecture and the augmentation of the data gave some of the best results on this first, grayscale dataset.

Using the grayscale dataset, the best possible mAP achieved was 14.3 and during training, the validation loss decreased to only 620.

4.2 Color dataset experiments

After perfecting it on the grayscale dataset as much as I could, the system still had some problems. If trained on small bounding boxes, as there was not enough data to properly detect the traffic lights, the system would have a big false positive rate. Furthermore, because it could not distinguish colors, it would consider car tail lights as traffic lights. This is understandable in the context of grayscale images, as the black fender of the car and the bright light resemble a traffic light, but this behavior cannot be accepted in a system meant to guide a car on the road.

Figure 13: False positives

Because of big false positive rate and also because training metrics indicated it, I have come to the conclusion that the current dataset was too small and with too little information for the network to understand and successfully learn how to predict traffic lights.

Therefore, after some research I switched on a new bigger, colored dataset [24]. The information provided by the three color channels (RGB) of the dataset coupled with the fact that it was 3 times larger than the previous one led to a substantial increase in inference quality.

Data generation The size of the dataset came with its own problems. Now it could not be loaded all at once in the memory without affecting the training batch sizes. Therefore, I implemented a method to feed the system images batch by batch in order to free up the memory for use with higher batch sizes. The system could now reliably predict traffic lights, although only at a close distance.

Small boxes To enable the system to detect the traffic lights sooner, it was then successfully trained on smaller bounding boxes.

Hyperparameter optimization Next the hyper parameters of the network were tuned during multiple trainings. In the end, they were as following:

- Initial learning rate = 0.001
- Maximum number of epochs = 40
- Batch size = 4
- Confidence threshold for object detection = 20
- Non-max suppression threshold = 0.1
- Color selector threshold = 20
- Gamma correction threshold = 0.55

After this optimization, the model constantly recorded a mAP over 35, while now the validation loss was decreasing to 70.

Lighting correction The model was able to successfully detect most traffic lights, but it still had problems classifying them. As I found out, this was due to the underexposed nature of the images in the dataset. Therefore, I applied gamma correction to the train and prediction images. This led to a definite reduction in classification mistakes but did not completely solved the problem. The model achieved a mAP of 43.6 with improvements to be made.

Color selection system and threshold For the times when the system is not certain which color to choose, the color selection system is employed. Further experimental training led to a good choice for the threshold after which the color selection algorithm is engaged. The best performing value was 20. This means that if the network predicts two color classes and their difference in confidence is smaller that 20/100, the color selection system will be used to decide which class is used.

Dataset increase As the model still presented a slight overfit tendency, the training set was increased from 4000 to 7500 examples. This worked well, but the model was still overfitting to the end. Instead of adding even more data and thus increasing the training time substantially, data augmentations were used. The exposure and contrast adjustments proved especially useful. After this, the model presented a good prediction quality without suffering from overfitting as could be seen in the plotted training and validation losses. The mAP became 48.5 and the validation loss was decreasing to 30.

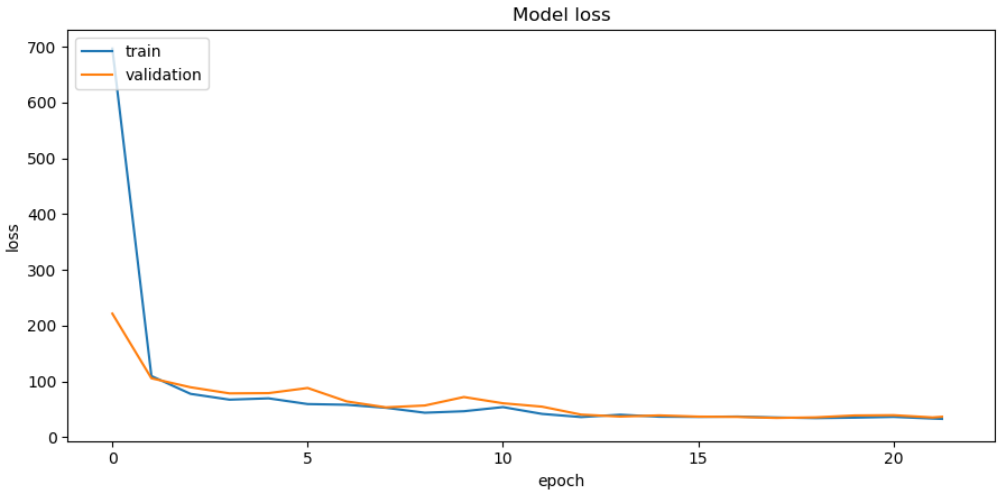


Figure 14: Good training and validation losses

Dataset balancing The mAP for the red and green classes were well over 60, but the yellow class was dragging the mAP down. This was because the train dataset contained a small amount of yellow traffic lights. This was remedied by adding a couple thousand frames with only yellow traffic lights. By doing so, the model reached its peak performance of 57.2 mAP.

Sequence prediction As the model training phase was completed, the demo was prepared. A script was written to load images, make predictions and display them in real time.

Post-processing vectorization As the system's speed could still be improved, the post-processing part was vectorized. This raised the inference speed of the whole system to 20 frames per second.

5 User Manual

5.1 System requirements

For training the network, the system requires a PC/laptop with a graphic card that has at leas 4 Gb of memory. For testing it does not require as much memory, but with more, time performance is improved.

In order to be able to install and run the training application and the already-trained network, the system needs to have an Anaconda environment that contains the following: Python 3.6, Tensorflow-gpu, Keras, Numpy, Pillow, Opencv and Matplotlib.

5.2 Installation

- Create Anaconda environment and install Python 3.6, Tensorflow-gpu, Keras, Numpy, Pillow, Opencv and Matplotlib.
- Install PyCharm and configure it to use the newly created environment

5.3 Run the system

Data preprocessing For preprocessing the data, a user needs to download and unzip the Driveu dataset [24]. Next, the user has to run the *preprocess_labels.py* script found in the *preprocessing* module.

This scrip requires the following command-line arguments:

- *–min_width* - Minimum width of the bounding boxes
- *–min_height* - Minimum height of the bounding boxes

- *–label_file* - Location of label file to be used
- *–image_file* - Location of the folder where all the image folders are contained
- *–output_file* - Location of the folder in which to store the processed annotations

Training For training the system, a user has to open the *train.py* script found in the *training* module, adjust the contained hyperparameters, then run the script. The network will start training, periodically displaying its progress.

The system takes care automatically of learning rate decrease, saving the trained model and early stopping the training, in case of convergence. After the training, the script displays the training/validation loss graph as an evolution over time.

Predicting To start the prediction process, a user has to first plot the ROC curve in order to select a good threshold for the confidence under which the predicted bounding boxes are ignored. This can be achieved by running the *model_visualization.py* script found in the *visualization* module.

After selecting a threshold to be used as the confidence threshold, the user has to open the *predict.py* script found in the *predicting* module. Here, he has to set which part of the dataset he wants to run a prediction on, give the previously selected threshold and run the script.

As the Tensorflow backend is not initialized yet, the system will take about 7 seconds to initialize, after which the prediction will be done, the re-

sults post-processed and displayed. The user can move to the next predicted image by pressing any key.

Running detection on a video To run a detection on a video, the user has to first save each frame in a folder. Then the *sequence_predict.py* script found in the *sequence_display* module has to be run.

The required command-line parameters are as follows:

- *–image_folder* - The folder from where to take the images for prediction
- *–delay* - Delay between images in miliseconds. At least 1 ms should be used.

Computing metrics To compute different metrics of the model, the user has to run the *compute_metrics.py* script found in the *metrics* module.

The required command-line parameters are as follows:

- *–frame_file* - The location of frame file to be used for extracting the ground truths
- *–image_file* - The location of the folder where all the image folders are contained. This is used for finding the detection results
- *–output_file* - The location where to store the metrics

After running this script, the system will make predictions based on the given images, save the detection results, save the ground truth labels, then intersect the two of them. Then, the metrics are computed. The mean average precision is printed and displayed, while the other metrics ca be found in the given output file.

6 Conclusions and future work

In the end, this system represents the basis of a framework for general purpose object detection. It is fast and robust, while also having a performance that rivals modern state-of-the-art systems. Therefore it is suitable for use in different localization and classification applications, in a multitude of environments.

As for traffic light detection, this project is an ADAS system for a smart car which improves life quality overall by adding autonomous features and increasing safety. In the future, it can be easily expanded with some other features such sign and pedestrian detection as the groundwork is already laid down. In terms of functionality, the system could be improved with more cameras around the car that can give video input and be processed at once by the system as to confer an overall 306 degree image of the surroundings.

The system also needs a night vision camera and a new dataset of corresponding images. The system should be aware of the sunrise and sunset hours and change the gamma correction accordingly. Also, it should be trained to be a more reliable detector during the night as well. Furthermore, the training dataset should be enriched with different weather conditions (rain, snow, etc.) so that the system can tackle any environment and condition.

Following these improvements, as well as using an extensive dataset for training, the system if fully capable of achieving production quality.

References

[1] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal loss for dense object detection," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, August 2017.

[2] W. Liu, D. Anguelov2, E. Dumitru, C. Szegedy, S. Reed, C.-Y. Fu, and A. Berg, "Ssd: Single shot multibox detector," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, December 2016.

[3] April 2019. [Online]. Available: https://www.racfoundation.org/motoring-faqs/mobility#a5

[4] August 2019. [Online]. Available: https://www.npr.org/2019/08/29/755441473/deaths-from-red-light-running-at-a-10-year-high-aaa-study-finds

[5] [Online]. Available: https://www.cooneyconway.com/blog/economic-and-societal-impact-zero-car-accidents-year

[6] "Neurons synapses," April 2019. [Online]. Available: http://www.human-memory.net/brain_neurons.html,April2019

[7] "Convolutional filter image link," May 2019. [Online]. Available: https://cdn-images-1.medium.com/max/1600/1*EuSjHyyDRPAQUdKCKLTgIQ.png

[8] "Pooling layer filter image link," May 2019. [Online]. Available: http://cs231n.github.io/assets/cnn/pool.jpeg

[9] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[10] "Convolutional neuronal network image link," May 2019. [Online]. Available: https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148

[11] [Online]. Available: http://pypr.sourceforge.net/kmeans.html

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The Third International Scientific Conference on Computer Vision and Pattern Recognition*, June 2015.

[13] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," in *International Conference on Machine Learning, Big Data management, Cloud and Computing*, July 2019.

[14] "Real-time object detection with yolo, yolov2 and now yolov3," March 2019. [Online]. Available: https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088

[15] [Online]. Available: https://docs.opencv.org/3.4/d3/dc1/tutorial_basic_linear_transform.html

[16] "Numpy." [Online]. Available: https://numpy.org/

[17] "Tensorflow," April 2019. [Online]. Available: https://www.tensorflow.org/

[18] "Keras," 2019. [Online]. Available: https://keras.io/

[19] "Opencv," 2019. [Online]. Available: https://opencv.org/

[20] [Online]. Available: https://matplotlib.org/

[21] [Online]. Available: https://www.scipy.org/

[22] [Online]. Available: https://scikit-learn.org/stable/

[23] W. He, Z. Huang, Z. W. C. Li, and B. Guo, "Tf-yolo: An improved incremental network for real-time object detection," August 2019.

[24] [Online]. Available: https://www.uni-ulm.de/en/in/driveu/projects/driveu-traffic-light-dataset/