Road markup detection algorithm for Duckiebot

Ilia Nechaev

14.02.2024

A project report submitted in partial fulfilment for the degree of

BSc (Hons) Computing

School of Sciences

University of Central Lancashire—Cyprus (UCLan Cyprus)

Abstract

This report provides a description of the road marking recognition project.

The Duckietown project uses a light-sensitive road marking recognition algorithm as a baseline. It is fast and consistent in constant light conditions. However when the lightning changes (different weather, time or even just one broken lamp in the room) this algorithm should be adjusted by changing its config file. It is a pretty suboptimal solution, which makes developers of the Duckietown project rebuild their solutions each time the light conditions are changed. Assuming that the Duckietown project imitates real autonomous driving cars and the idea of the smart city, rebuilding the solution for autopilot because of the change of the daytime doesn't look like a good imitation of the real system. So it was decided to create a new algorithm for detecting road markups, based on a deep learning (DL) algorithm. This approach makes the algorithm light-independent. The report describes in detail the mathematical foundations of the algorithms used in the project, as well as a description of the overall design and implementation. In addition, collecting and marking up the dataset for DL is described in this report. As a result, the algorithm was created and the dataset was published. This algorithm was tested in Joint Advanced Student School (JASS) 2024 by JetBrains and was pretty successful.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.
I certify that this document reports original work by me during my University project.

Signature: Ilia Nechaev

Date: 29.02.2024

Acknowledgements

Special thanks to Mr. Konstantin Chaika for helping and guidance in Duckietown world.

Also I'd like to thank JetBrains for an opportunity to participate in JASS 2023, where I discovered Duckietown and decided to learn more about computer vision.

It's also important to say thanks to UCLan Cyprus for providing me with the Duckiebot and Dr. Panayiotis Andreou for helping me with the paperwork.

Special thanks to my roommates: Ivan Iambarshev, Vasilii Telnov, Daria Chystiakova and Vadim Volkov for help in collecting the dataset and emotional support.

Also I'd like to say thanks to my teachers: Dr. Sergey Nikolenko, Dr. Vlad Shakhuro, Mr. Andrey Stotski, Mr. Anton Alexeev, Dr. Alexander Avdushenko for giving an amazing start in machine learning and computer vision.

And of course a big thanks to my parents: Mrs. Natalia Nechaeva and Mr. Andrey Nechaev for support in any meaning of this word.

Table of Contents

List of tables

List of figures

1 Introduction

1.1 Background

Thanks to the development of technology, self-driving cars and robots have become our daily routine, they have already changed and continue to change the paradigms of transport and automation in various industries such as delivery, taxi, trucking and more. The combination of the development of machine learning, robotics, mechatronics and input systems (cameras, lidars, radars, etc.) has become an important step towards improving self-driving cars, robots and other systems, providing incredible opportunities and prospects in other areas.

Because of innovations in computer vision especially with using machine learning and deep learning as well as different types of sensors cars with autopilot and bots got a breakthrough in development. Tesla with its autopilot for electric cars, Amazon with its research and implementation of autonomous delivery bots and other companies are great examples of developing the industry. These companies use different technologies like deep learning and different sensors (like lidars, cameras, gyroscopes, etc) for developing their algorithms.

The detection and labeling of environmental objects is an important factor for the operation of the perception system, which ensures the correct operation of autonomous cars and bots. To ensure safe and effective autonomous operation, detection and labeling processes must be configured accurately and reliably since these systems aim to simulate human perception and the ability to make their own decisions in unforeseen situations.

The perception and interpretation of complex environmental stimuli in real-time is made possible for autonomous vehicles and bots mainly through the use of detection and labeling algorithms. Using sensor data from cameras, lidar, radar and other detection tools, these algorithms simplify the search and identification of objects, obstacles and road infrastructure around the car. Also, accurate detection and labeling capabilities increase the vehicle's environmental awareness and allow proactive decisions to be made in dynamic driving conditions. Examples of how these algorithms work are the ability to distinguish between road signs, road markups, as well as pedestrians and cyclists.

For the full functioning of a vehicle in autonomous driving scenarios, accurate identification and labeling of possible hazards is crucial to reduce risks and ensure the prevention of various types of accidents. Active collision avoidance techniques such as emergency braking or maneuverable steering are made possible by fast and well-established detection and marking systems that recognize and classify objects and obstacles in the path of an autonomous vehicle. Reliable detection algorithms also have the ability to predict unforeseen situations to some extent, such as unexpected pedestrian crossings or strange vehicle behavior, and respond to them with the slightest delay, increasing overall road safety and thus reducing the risk of accidents.

In addition to recognizing objects, labeling algorithms help with semantic understanding and contextual interpretation of the environment. These methods allow a vehicle to distinguish between different types of objects (for example, vehicles, pedestrians and bicycles) and draw conclusions about their possible behavior by providing semantic labels and attributes to identified elements and entities. Making rational navigation decisions, such as giving way to pedestrians at pedestrian crossings or reconciling complex traffic scenarios with numerous parallel interacting road users, depends on this semantic understanding.

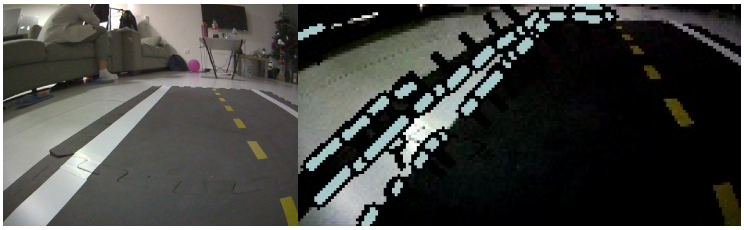
Reliability, as well as the ability to adapt to rapidly changing circumstances, are also necessary for detection and marking systems to cope with their inherent unpredictability and variety of situations in the real world. Autonomous vehicles and robots face a number of obstacles and circumstances that affect the accuracy of detection and labeling algorithms, such as changing lighting conditions, interference or noise in sensors. Therefore, in order to improve the performance of these algorithms and generalization capabilities in various work situations, continuous optimization and refinement using machine learning and methodologies based on the data obtained is crucial.

Summing up, we can say that the perception systems in autonomous cars and bots mostly rely on detection and labeling systems as fundamental building blocks that allow them to perceive, see, understand and overcome difficult situations on their own. In order to ensure safer and more efficient autonomous operation, and in addition to increase awareness of the autonomous transport environment, as well as reduce risks and facilitate smooth interaction with it, detection and labeling algorithms must be sufficiently accurate, reliable and flexible.

1.2 Context

In March April of 2023 I was a participant in Joint Advanced Student School (JASS) by JetBrains. There we developed a smart city prototype using the Duckietown project. I have noticed that Duckiebots robots that imitate self-driving cars in a smart city behave differently with different lightning. For example, we always kept curtains closed, but if the wind flow opened them the bot started driving unpredictably.

After a small investigation, I've discovered, that the current algorithm that is run on Duckiebots is based on segmenting pixels by colors. This approach is good when light conditions are constant, but each time they change (new room with different lightning or changing time in a room with opened curtains) the bot needs recalibration. Also important to mention that each Duckiebot has LED lights that can be almost any color which can lead to wrong detection.



(a) Raw image of the road with the first light condition (b) Mask of the road with the first light condition

(c) Raw image of the road with the second light condition (d) Mask of the road with the second light condition

Figure 1: Old algorithm segmentation with different light conditions

1.3 Applicability of Findings to the Commercial World

So after this observation, the idea of developing a light-insensitive road markup detection algorithm came to my mind. I wanted to create an algorithm based on the deep learning (DL) approach, so it would be easy to scale (in terms of developing a new neural network), light-insensitive, but at the same time light in computational terms.

1.4 Bot specification

The idea of using the DL approach came to me as soon as I've discovered that Duckiebots can be powered by Jetson Nano by Nvidia.

AI Performance	472 GFLOPs
GPU	128-core NVIDIA Maxwell™ GPU
CPU	Quad-Core ARM® Cortex®-A57 MPCore
Memory	4GB 64-bit LPDDR4 Memory
Storage	Micro SD

Table 1: Jetson nano specs

4GB of RAM and 128 CUDA cores are more than enough for a lightweight convolutional neural network (CNN)

2 Background reading

2.1 Introduction

Creating the algorithm for lane detection involves two major steps: collecting and marking data and creating the algorithm itself.

Lack of finance makes me find a way to mark up the dataset in an autonomous way. To do this I will try non-ML and some unsupervised learning methods of marking. The second step is creating the algorithm itself. After the CNN revolution that happened in CV more than 10 years ago, when AlexNet was presented [Krizhevsky et al., 2012], many different approaches and architectures were created, most of which can be used almost on any hardware for almost any task. In this review, my goal is to choose a few of them, test and find one that suits my task best.

2.2 Data preprocessing

2.2.1 Introduction

All ML projects start with collecting a dataset. So I followed the same procedure.

The dataset was collected using Duckiebot. A small server was created using GoLang to receive images from the bot's camera. After that a few different approaches were used to mark it up. All approaches had some disadvantages that will be listed below:

2.2.2 Non-ML approach

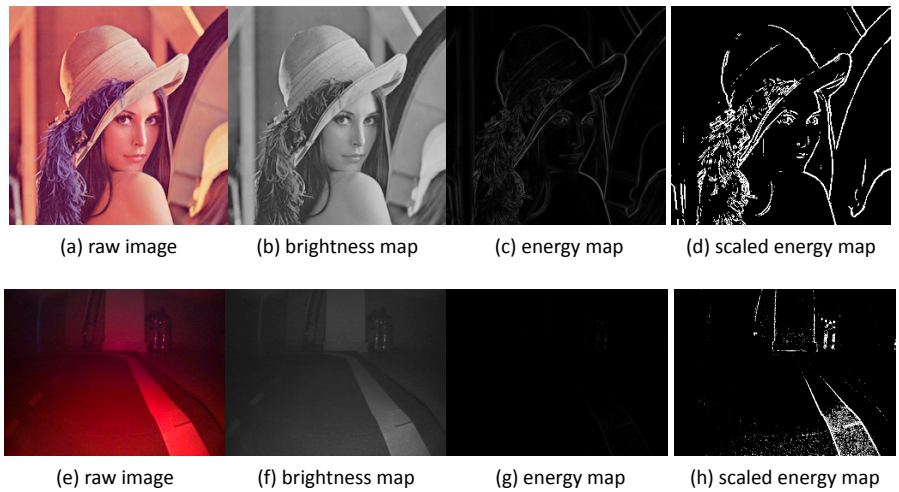


Figure 2: Results of each step for high contrast and low contrast image

The first idea was to use a non-ML approach based on "energy". [Avidan and Shamir, 2007]

The main ideas of this approach are:

1. Convert RGB pixels to brightness
2. Use discrete derivatives of brightness to compute energy

1. **Brightness**
   To convert RGB image to brightness map it was decided to use Y component of YCbCr color space, because this component shows the brightness. To convert RGB image to YCbCr we need to use simple operation [Podpora et al., 2014]:

$$\begin{pmatrix} Y' \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix}$$

2. **Energy** For energy, it was decided to use the formula from seam carving paper [Avidan and Shamir, 2007]:

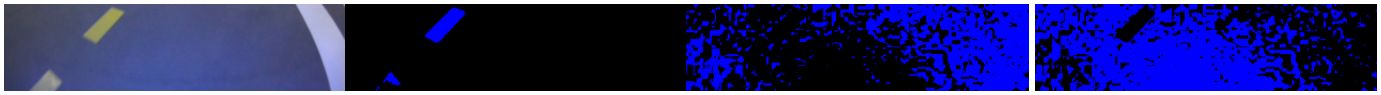
$$e_1(I) = |\frac{\partial}{\partial x} I| + |\frac{\partial}{\partial y} I|$$

   Where

$$|\frac{\partial}{\partial x} I_{i,j}| = \frac{I_{i+1,j} - I_{i-1,j}}{2}$$

   The intention was to use this algorithm to find a contour of road markup. Unfortunately, this approach performs well on high-contrast images, but the project's problem is that images in the dataset can and should be low contrast. So this method was abandoned.

2.2.3 KNN approach



(a) Raw image (b) Middle lane mask (c) Side lane mask (d) Road mask

Figure 3: Masks provided by KNN algorithm

The second thought was about KNN [Guo et al., 2003]. It's powerful, but the computational heavy clustering algorithm can be used in unsupervised learning. I tried to use it in two ways:

1. Use only color for clustering
2. Use colors and position of pixels

Both approaches had their advantages and disadvantages, but in the end this method was rejected, because it couldn't provide good markup even on bright images.

2.2.4 EM algorithm

Last hope was the EM [Ng et al., 2004] [Dellaert, 2003] algorithm. The idea of the EM algorithm is to use general knowledge about our world to predict something. In my case, I assumed that the color distribution of each image is a mixing of a few Gaussians.

Assumptions:

1. Colour distribution is formed by three Gaussians with parameters: $\{\bar{\mu}_1, \Sigma_1\}, \{\bar{\mu}_2, \Sigma_2\}, \{\bar{\mu}_3, \Sigma_3\}$
2. Choosing gaussian is made with probabilities: $\{\pi_1, \pi_2, \pi_3\}$

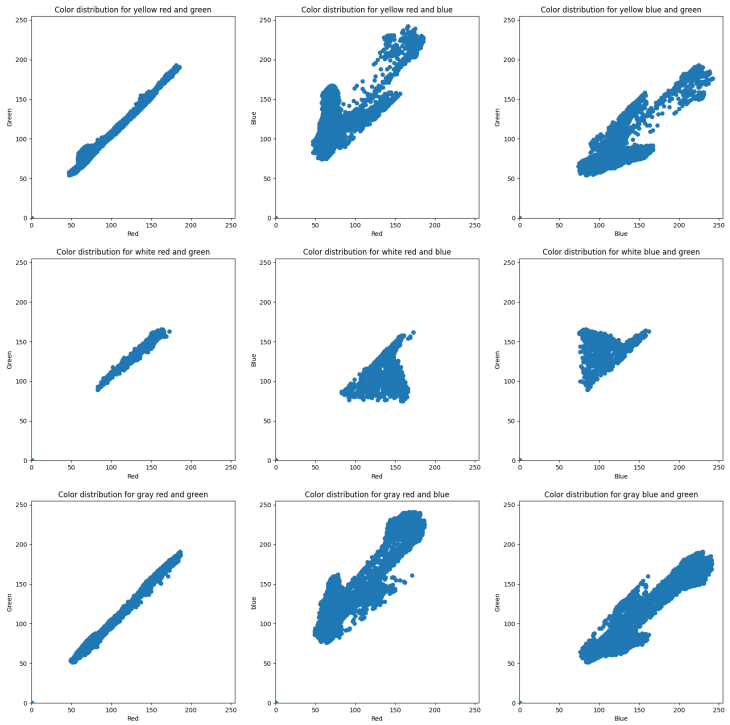


Figure 4: Color distribution on images

PDF of Normal distribution is

$$\mathcal{N}(\bar{x}|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

The idea of the EM algorithm is simple:

1. E-step: fix $\bar{\theta}$ parameters of model, find $\mathbb{E}[z]$ math expectation of hidden parameters:

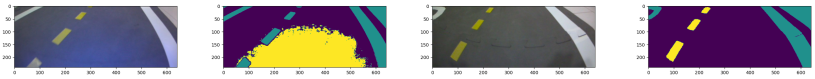
$$\mathbb{E}[z_{n,k}] = \frac{\pi_k \cdot \mathcal{N}(\bar{x}_n|\bar{\mu}_k, \Sigma_k)}{\sum\limits_{l=1}^{K} \pi_l \cdot \mathcal{N}(\bar{x}_n|\bar{\mu}_l, \Sigma_l)}$$

2. M-step: fix $\mathbb{E}$ and maximize likelihood: $\mathbb{E}[\log p(x,z|\bar{\theta})] \underset{\bar{\theta}}{\longrightarrow}$ max:

$$\mathbb{E}[\log p(x,z|\bar{\theta})] = \sum_k \Big(\sum_n \mathbb{E}[z_{n,k}]\Big) \log \pi_k + \sum_k \sum_n \mathbb{E}[z_{n,k}] \cdot \log p(\bar{x}_n|\bar{\mu}_k, \Sigma_k)$$

Also we can notice, that $\sum\limits_n \mathbb{E}[z_{n,k}] = \mathbb{E}[|C_k|]$, where $|C_k|$ is the amount of samples in cluster C_k

The results of the EM algorithm were also poor, but a bit better than KNN's ones:



(a) Raw image bad sample (b) Mask bad sample (c) Raw image good sample (d) Mask good sample

Figure 5: Masks provided by EM algorithm

2.2.5 Result approach

The final approach is combining energy with EM, then using its outputs as inputs to logistic regression [Peng et al., 2002] with data augmentations [Perez and Wang, 2017] [Nikolenko, 2019].

It was decided that this approach should be more useful, because:

1. Data augmentations can help with the inefficient of previous approaches, because applying color filters to 'clean' images can emulate the light of LEDs of the Duckiebot without loss of accuracy of EM algorithm predictions
2. The noise of the labels can be compensated by the size of the dataset [Sun et al., 2017]

2.3 Algorithm implementation

2.3.1 Introduction

To implement the algorithm some DL approach will be used. There are a few ways to do it:

- Fully connected NN to classify the color of pixel [Scabini and Bruno, 2021]
- MobileNet [Howard et al., 2017]
- ShuffleNet [Zhang et al., 2017]
- EfficientNet [Tan and Le, 2020]
- Squeeze-and-Excitation Networks [Hu et al., 2019]

2.3.2 Fully connected NN

This approach will be used as the baseline in terms of accuracy and computational speed

This architecture is extremely easy to implement and it is relatively fast. [Scabini and Bruno, 2021] The key idea of fully connected NN is MLP (multi-layer perceptron):

If we have input tensor $X \in \mathbb{R}^{n\times 1}$, we can apply linear transformation to it: $W \cdot X$, where $W \in \mathbb{R}^{k\times n}$, after that we need to add some nonlinear activation (ReLU, sigmoid, Tanh, etc) and apply the same block a few more times As the last activation, we can use softmax for classification or just leave it as it is for regression

But this type of NN isn't so popular in computer vision, because it can't handle random-sized images without preprocessing and it struggles with detecting objects in different parts of the image. But in the case of this task, it will not be a problem, because all images will be the same shape and objects will

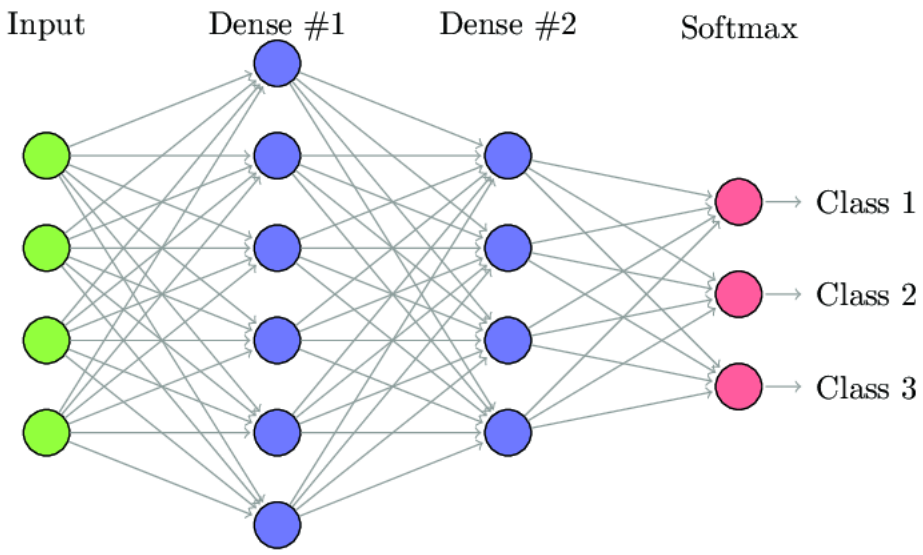


Figure 6: Architecture of fully connected NN with 3 layers

be in approximately the same spots. However using a hole image as input tensor will be tough because the image size is 320×480 pixels, so the input tensor will be of size 153600, even if a relatively small hidden layer is used (10000 neurons), then one forward pass will be about 23,5 GFLOPs, which is much bigger than CNNs consume. So this architecture can be used to determine the color of each pixel only independently from others, this will affect accuracy because such an approach can't determine objects, only colors, which is good as a baseline, but can't be used in production.

Determining pixel color will require an input tensor of size 3 (or 5 if positional embeddings are used), output tensor will be of size 3 (to show probabilities of belonging pixel to yellow, white or gray color) and hidden layer can be any size because as a result flops will be: $153600 \cdot (3 \cdot n + n \cdot 3) = 921600 \cdot n$ FLOPs if n is less than 20, then such NN can't use more than 18 MFLOPs which is good in comparison with CNN

2.3.3 MobileNet

MobileNet Architecture family seems to be a more suitable approach as it is relatively fast and has pretty good accuracy on the ImageNet dataset.

Core idea of MobileNet [Howard et al., 2017] is splitting standard convolution [O'Shea and Nash, 2015] on two steps:

- Using convolution with one channel for each input channel (depthwise convolution)
- Use the result of the previous step as input for pointwise convolution (convolution with $1 \times 1 \times M$ kernel, where M is amount of input channels) this step lets us extract features

This approach is fast, because in usual convolution with kernel $D_K \times D_k \times M \times N$ (where D_K is kernel size, M amount of input channels and K amount of output channels) we need $D_K \cdot D_k \cdot M \cdot N \cdot D_F \cdot D_F$ operations per image (we can assume that all images are squared, it is the usual approach in CV). On the other hand in MobileNet architecture convolution needs only $D_F \cdot D_F \cdot M\big(D_K \cdot D_K + N\big)$ operations, so we have improved in

$$\frac{1}{\frac{1}{N} + \frac{1}{D_K^2}}$$

Assuming that the usual kernel has size 3×3 it is a significant boost.

Also, this approach reduces memory usage, because each convolutional layer needs only $M \cdot D_K \cdot D_K + N \cdot M$ parameters, while classical convolutional layer with the same input hyperparameters needs $M \cdot D_K \cdot D_K \cdot N$

2.3.4 ShuffleNet

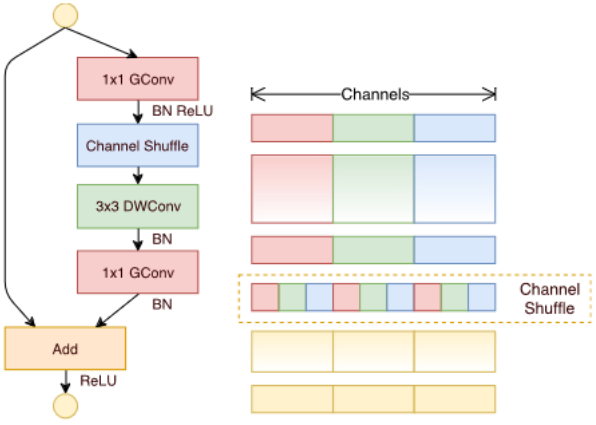
This model develops ideas of MobileNet (even if it is not fully represented in paper, however, it's obvious).

Authors of ShuffleNet propose to use group pointwise convolution (GPC) instead of classical pointwise convolution [Zhang et al., 2017].

The idea of GPC is relatively simple: instead of performing this operation: $I * K$ (where $*$ is convolution, I is tensor with shape $H \times W \times C \times M$ and K pointwise convolution kernel with shape $1 \times 1 \times C \times M$, C amount of input channels, M amount of output channels), which takes $W \cdot H \cdot C \cdot M$ MFLOPs (floating point multiplication-adds), we can use this operation:

$$\underset{i=0}{\overset{g-1}{CONCAT}}(I_{[i \cdot g:(i+1) \cdot g]} * K_i)$$

where $I_{[a:b]}$ tnesor which consists of a'th till b'th channels of tensor I. Such an approach requires in g times fewer MFLOPs, but we have to face another problem: output channels, which were produced by one group of input channels 'do not know' anything about other input channels. To fix this authors of ShuffleNet propose a shuffle layer, which can be used in the bottleneck block (Figure 7).



(a) ShuffleNet Bottleneck block

(b) Shuffle layer

Figure 7: ShuffleNet architecture

If we count MFLOPs of bottleneck block in ShuffleNet and MobileNet (we can use its ideas to construct bottleneck block), we will get such results:

1. ShuffleNet: $W \cdot H \cdot (\frac{2 \cdot C \cdot M}{g} + 9M)$
2. MobileNet: $W \cdot H \cdot (2 \cdot C \cdot M + 9M)$

As you can see we have a small gain, but we lose some accuracy, even in the article authors provide us with data, that shows that ShuffleNet isn't always better than MobileNet, so I will try to implement both of them to find the best one.

2.3.5 EfficientNet

EfficientNet also seems to be a good model to try [Tan and Le, 2020].

It uses MobileNetV2 [Sandler et al., 2019] bottleneck as main building blocks and this type of architecture is easy to scale. Moreover base model EfficientNet-B0 has fewer MFLOPs in comparison with MobileNet (390 MFLOPs vs 569 MFLOPs), but has a bit more parameters. However, the Duckiebot has 4GB RAM so it can handle up to 1 billion parameters, which is much bigger than 5 million of EfficientNet and 4,2 million of MobileNet

2.3.6 Squeeze-and-Excitation Networks

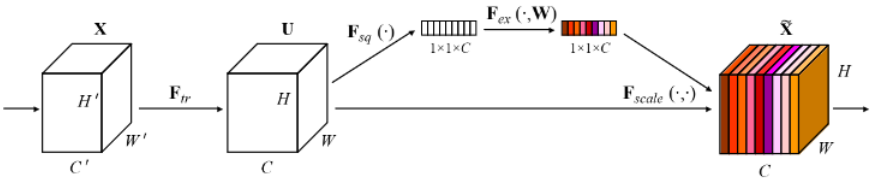


Figure 8: Squeeze-and-Excitation block architecture

This type of CNN is a transitional link between classical CNN architecture and transformer architecture in CV [Dosovitskiy et al., 2021]. Authors of the paper [Hu et al., 2019] propose a Squeeze-and-Excitation block that performs as attention mechanism that can be embedded in CNN architecture.

The main idea is to flatten each channel of the input feature map in a such way, that each channel would be represented by only one value. It can be done by applying global average pooling to each channel:

$$F_{sq}(u_c) = z_c = \frac{1}{H \cdot W} \cdot \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i,j)$$

where u_c input channel, W and H are spatial dimensions, z_c result statistic

After that, it's needed to implement nonlinear interaction between channels, to do that it's a good idea to use 2 fully connected layers, one with ReLU activation and the other with simple sigmoid:

$$F_{ex}(z, W) = s = \sigma(W_2 \cdot (\delta(W_1 \cdot z)))$$

where z embeddings from previous step, $W_1 \in \mathbb{R}^{\frac{C}{r} \times C}$, $W_2 \in \mathbb{R}^{C \times \frac{C}{r}}$, σ ReLU activation.

r is a scale factor: fully connected layers are computationally heavy, so decreasing the hidden layer is needed for optimization purposes.

Finally, s should be scaled up to save spatial dimensions of input, to achieve this the input of the block can be multiplied on s channel-wise:

$$F_{scale}(u_c) = \widetilde{x}_c = s_c \cdot u_c$$

The full architecture is shown in Figure 8.

This approach can be implemented in any architectures described above, it's computationally lightweight and gives a good accuracy boost (1.7% for MobileNet and 1.4% for EfficientNet).

2.3.7 General ideas

Also important to mention, that all the described above approaches can't be implemented directly due to a lack of computational resources, so only its ideas will be taken. Such layers like batch normalisation [Ioffe and Szegedy, 2015] or dropout [Srivastava et al., 2014] also will be used, because they prevent overfitting and let models converge faster, which is important, due to the low size of the dataset.

3 Project planning

3.1 Introduction

The project was divided into three main steps:

1. Preparing the dataset.
2. Choosing and implementing deep learning model.
3. Deploying model on the Duckiebot.

Each step was a small project with its research, implementation and evaluation.

3.2 Methodology

3.2.1 Intorduction

I haven't intentionally used any specific methodology or Agile techniques except MoSCoW, because, in my opinion, it is nothing but a waste of time. However, it happened that I had some kind of scrum sprints. Each step described above was made as a scrum sprint of two weeks of 14 hours of work per day.

Priority	Description	Result
Must	Design, develop and deploy algorithm using reinforcement learning technique for marking up dataset	Dataset collected, algorithm published
Must	Design and develop a light-insensitive and light-weight algorithm for detecting road marking	Algorithm developed and published
Should	Deploy light insensitive and light-weight algorithm for detecting road marking on Duckeibot	Algorithm deployed, full built is published
Could	Crossroad crossing algorithm based on computer vision techniques	Algorithm hasn't been developed, but it can be after finishing all the exams. In this case, the algorithm would be published as a scientific paper
Won't be	Light-weight general-purposes line detection algorithm, which can be used for any type of road markup	Algorithm hasn't been developed and won't be developed by me during spring-summer 2024

Table 2: MoSCoW table with the results

3.2.2 Preparing dataset

The preparation of the dataset was divided into five substeps fully described in the design and implementation chapters:

1. Writing some utilities for getting pictures from the Duckiebot.
2. Collecting dataset.

3. Research on self-supervised learning techniques for marking up the dataset.
4. Creating a self-supervised learning model and pipeline for marking up the dataset.
5. Validation of the resulted markup by a human.
6. Publishing the dataset.

3.2.3 Choosing and implementing deep learning model

This step was just a classical example of the creation of a DL model. So it was divided into 2 substeps:

1. Reading articles.
2. Model implementation.

3.2.4 Deploying model on the Duckiebot

The last step of the project was also divided into 3 substeps:

1. Running DL model on the Duckiebot.
2. Adjust the outputs of the model in a such way that they fit the inputs of the lane-following algorithm of the base docker image of the Duckiebot.
3. Evaluate the result of the deployment and the whole project.

3.3 Requirements

3.3.1 Preparing dataset

For this part of the project, there were 2 requirements:

1. Collect about 9000 photos for the dataset. In the flat, where the polygon was mounted, there are 2 sets of lamps, so 3000 photos per set were collected, as well as 2000 photos for the combination of the sets and 1000 photos with daylight only.
2. Create a self-supervised learning algorithm for marking up the dataset with an accuracy of at least 70%.

The first requirement is needed to make resulted algorithm light-insensitive. With a combination of data augmentation (to be more precise color and brightness shifting) the dataset collected in a such way can make the resulting DL model pretty accurate, because it will not rely only on the colors and brightness of pixels in its predictions (this will be shown in the implementation chapter).

The second requirement was partly my interest and partly project requirements. For me, it was interesting to learn how to segment a picture on colors in different light conditions. For the project, it was important to have a pretty large dataset to train the DL model on.

Human validation

As a result, of implementing this part of the project one more requirement was found. It is human validation of the dataset.

Human validation is a critical step in creating a road marking dataset (even for Duckietown, which cannot harm a person), the marking of which was created automatically. A human can ensure the accuracy and reliability of the data obtained, which is extremely important in the autopilot contest. Unlike machine learning algorithms without a teacher, people are able to recognize inaccuracies in machine markup: incorrect classification, noise and other artifacts. Human verification of the markup, although it reduces the size of the dataset, improves the average quality of the markup, reducing the risk of incorrect training of the neural network, and thereby improving autopilot in real conditions.

Human verification reduces the overall noise of the dataset. This provides an opportunity to get better training data for deep learning algorithms. With careful verification, a person is able to eliminate false labeling and correct falsely labeled pixel maps, thereby ensuring semantic consistency throughout the dataset. This noise reduction has a beneficial effect on the training of neural networks, especially on relatively small datasets for computer vision.

Even though human verification entails additional financial expenses and consumes a lot of time, its necessity is priceless, especially for datasets on which machine learning algorithms are trained, the results of which can harm a person. Human verification allows you to overcome the limitations associated with automatic markup, such as false segmentation or noise, which can lead to incorrect training, and then the operation of the neural network. In the case of using neural networks for autopilot operation, incorrect operation of the algorithm can lead to human casualties.

Last but not least. The project used human verification of automatic markup, rather than fully human markup. This approach allows, on the one hand, to ensure the accuracy of the dataset, and on the other hand, it does not require resources such as completely human markup. Although this approach slightly reduces the size of the dataset, it is a reasonable compromise combining both human precision and the scalability that automatic markup provides. This approach allows, using limited resources, to create large and relatively accurate datasets, which is very important in such computer vision algorithms.

3.3.2 Choosing and implementing deep learning model

For this step, requirements were hardly evaluated, because some characteristics of the model (e.g. time per froward pass) are different in a training environment (Gooogle Colab) and working environment (the Duckiebot). So the requirements of this part of the project can be implemented and evaluated in different steps.

1. Model should be pretty small. I assumed the volume of all the parameters of the model can't be more than 2GB (even thou the Duckiebot has 4GB of RAM I've decided to leave some space for other parts of the OS).
2. Model should be fast. The forward pass of the model shouldn't take more than 20 milliseconds. This requirement is important to make enough FPS rates. I assumed that having 20 FPS is more than enough for the Duckiebot. So if the model takes 20 milliseconds and data transferring takes 30 milliseconds this will be 50 milliseconds per image which is 20 FPS. In the Implementation chapter it will be described in more detail.

3.3.3 Deploying model on the Duckiebot

In this part of the project, there were no requirements in the beginning, but when during the implementation, it was figured out that this task wasn't as easy as expected.

1. Deploy the DL model to the Duckiebot and adjust the outputs of the model in a such way that they fit the inputs of the lane-following algorithm of the base docker image of the Duckiebot. This was the only requirement at the beginning of the project.
2. Migrate to TensorRT SDK. This requirement appeared as soon as I ran the model on the Duckiebot because without TensorRT SDK the forward pass of the model and the data transferring were much

slower than I expected (in more detail it is described in the Implementation chapter).

3.4 Potential Solutions

3.4.1 Introduction

There can be a lot of variants of implementation of this project. I will divide this section into two subsections:

1. Languages and libraries to discuss choices of the languages and libraries used in the project.
2. Algorithms to discuss choices of the algorithms used in the project.

3.4.2 Languages and libraries

1. Language for the server to collect photos from the Duckiebot. Servers can be written in almost any language: Python, Java, NodeJS, etc. But I've decided to use GoLang as I'm familiar with writing servers in this language. Because the server wasn't the main part of the project, but just a utility I didn't need it to be extensible or super efficient. The only important thing about the server was quick development and GoLang can provide instruments for this in addition my knowledge of this language made it the only option for this project.
2. Main language for the project. It's already standard for the industry to use Python as a language for the ML/DL projects [Raschka et al., 2020], so for me, it wasn't a question of which language to use. However important to mention that sometimes ML/DL projects can be written in RLang or MATLAB, but I'm not familiar with either of them, also MATLAB as far as I know isn't free to use, moreover, Duckietown uses Python as the main language. Also, it was possible to consider using C++ and writing the DL model in this language, but this would increase development time because I've never worked with Torch on C++ and have no experience in developing ROS nodes for the Duckiebot in this language.
3. Libraries for the project. Talking about libraries it's important to mention that there are almost no analogs for the Scikit-learn library which was used for the EM algorithm. But for the DL model, there are two big analogs of the PyTorch library used in this project. These analogs are TensorFlow and JAX, which are also libraries for creating neural networks. However, I learned deep learning using PyTorch so I know this library much better than TensorFlow and JAX, moreover, the Duckietown project has no base docker images with JAX or TensorFlow so these libraries weren't even considered.

3.4.3 Algorithms

Model	Parameters
My model	10,5K
DeepLabV3	11M
FCN	35,3M
LRASPP	3,2M

Table 3: Fully implemented NN in comparison with my architecture

1. Using EM algorithm for marking up the Dataset. EM algorithm wasn't the only option for the marking up algorithm as it was described in the Background reading chapter. Also in addition to that chapter, I considered using the DBScan algorithm in combination with the 'energy approach'. But the EM algorithm showed its consistency and it was decided to use it.

2. DL architecture also has a lot of options as was described in the Background reading chapter. But the resulting algorithm was pretty consistent, light-weighted in computational terms and had a relatively small amount of trainable parameters, so it was decided to stay on the created architecture. I thought about using other backbones like ShuffleNet [Zhang et al., 2017], EfficientNet [Tan and Le, 2020] or Visual Transformers [Dosovitskiy et al., 2021] or even fully implemented segmentation neural networks like DeepLabV3 [Chen et al., 2017], FCN [Long et al., 2015] or LRASPP [Howard et al., 2019], but all these variants are either too big or too computationally heavy and do not give much better results.

3.5 Tools and Techniques

During this project, a lot of different tools were used:

1. IDEs of JetBrains were used for developing:
   (a) PyCharm for Python developing.
   (b) GoLand for GoLang development.
2. Jupiter notebooks were used for developing ML algorithms (both the EM algorithm and the DL model). For the model training, Google Colaboratory was used, because I don't have a computer with GPU to train my model on.
3. Also usual in development things were used in my project:
   (a) git for VCS and GitHub for publishing results.
   (b) Docker for deployment on the Duckiebot.
4. Visual Studio Code was used for writing this report because it is written in LaTeX.

3.6 Legal, Social, and Ethical Issues

3.6.1 Intorduction

As it was written in the technical plan (appendix 1) it is a research project with no legal, social or ethical issues. However, after the full implementation of the project, some things need clarification.

3.6.2 The dataset

The Dataset was collected and published under MIT license [Snyk, 2024]. The choice of license is so because I want the dataset to be used by other users, but because of autonomous marking up (even if the markup has been verified by a human), there can be some mismarked pictures, which can cause problems In training some algorithms.

3.6.3 The algorithm

The algorithm is also published under MIT license [Snyk, 2024]. The reason is simple. I want the algorithm to be open-sourced and available for free commercial, educational or any other usage. The algorithm was developed for Duckiebot, but it can (but is not recommended to) be used as a baseline for real autonomous driving cars. However, this usage of the algorithm wasn't initially planned that's why I don't want to be responsible for any consequences of misusing my algorithm.

3.6.4 The Duckiebot

The Duckiebot can drive autonomously. So it can cause some legal issues if it bumps into something or someone. Even though the bot is small it can break something. That is the reason why it should be run only under human control to prevent such accidents.

Also, the bot was provided for me by the University, so I'm responsible for its condition and possible damage, that's why running the bot without my control and permission was and remains strongly prohibited.

3.7 Summary

This section was written after the project was finished so it shows not only planning but also the resulting technologies and ideas that were used. As you can notice different approaches were considered and rejected, to fully satisfy the desired requirements. As it was said Agile wasn't intentionally used, but anyway the project was finished on time and it met all the requirements. A lot of different tools and programming languages were used in this project. As it was said it is a research project that doesn't involve any people (besides me, the developer) and doesn't use unlicensed software, so it can't have any legal, ethical or social issues.

4 Design

4.1 Introduction

This chapter covers project architecture and process flow as well as the math foundations of algorithms used in this project. The second topic is partly covered in the Background reading chapter, but here there is all the information necessary for understanding.

4.2 Project architecture and process flow

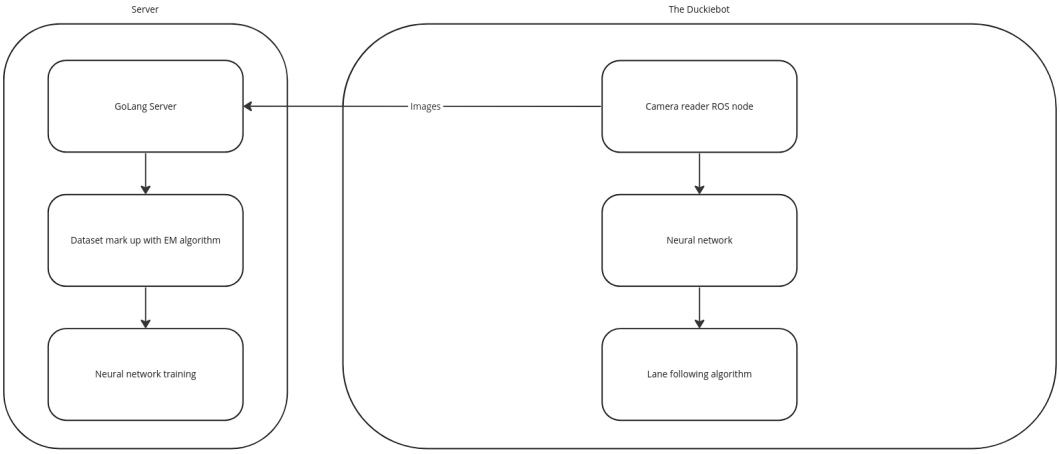


Figure 9: Project architecture

4.2.1 Introduction

As was written in the previous chapter the project was divided into three mini-projects:

1. Creating the dataset.
2. Creating road markup segmentation algorithm.
3. Deployment of the algorithm on the Duckiebot.

Each mini-project has its own architecture and process flow, however, they were developed in similar environments. The overall environment is shown in Figure 9. Pictures were collected from the Duckiebot's camera, then they were sent to the GoLang server. After that, they were processed in Google Colab and then passed to the neural network training pipeline. During the deployment part images from the same camera on the Duckiebot were passed to the neural network which had been already deployed on the Duckiebot. And outputs of the neural network were passed to the lane-following algorithm.

4.2.2 Creating the dataset

Collecting the dataset involved three steps:

1. Developing the infrastructure. The infrastructure includes the ROS node to send data from Duckiebot and the GoLang server to receive and save data. The image was received from the camera

and after that, it was sent to the server. Because sending takes some time and this process couldn't be run asynchronously, not all images were sent, but every 20.

2. Collecting the dataset. The collecting of the Dataset was performed manually. The Duckiebot was driven around the road for a few hours. During this process, the lights were changed to collect different data.

3. Marking up the dataset. The marking up of the Dataset was performed in the Jupiter notebook. Images were marked up in batches of size no more than 500 samples. Moreover, pictures of the different light conditions were marked up in different iterations of the pipeline, to prevent noise. After the marking up the dataset was evaluated by humans to remove samples with bad segmentation and after that uploaded to GitHub.

4.2.3 Creating road markup segmentation algorithm

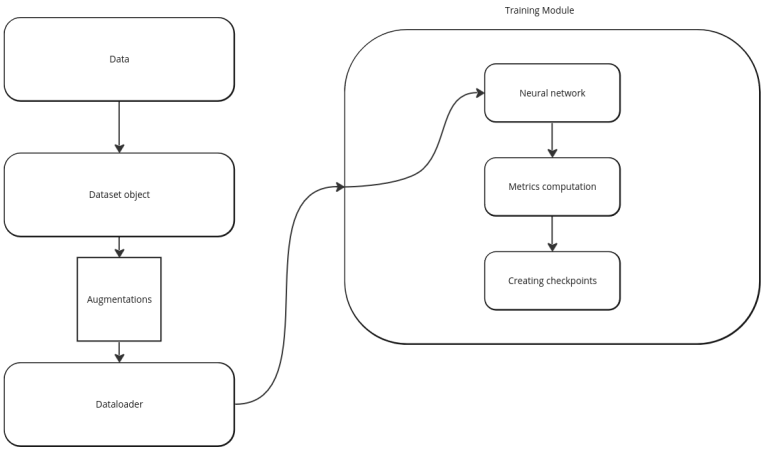


Figure 10: Training pipeline

The algorithm was developed in the Jupiter notebook in Google Colab. The pipeline is shown in figure 10.

The dataset was downloaded from GitHub (it more quicker than uploading it to Colab from a laptop). After that, the data of the dataset was loaded into the dataset class. The idea of this class is to keep information about the dataset, but not the data itself, because the dataset is pretty huge and keeping all data loaded in the RAM is pretty expensive. So the Dataset class provides lazy access to the dataset data. If some sample is required it will be loaded, otherwise, only the path to the image will be saved. Also, the Dataset class is responsible for the augmentation of the data. When a sample is required if this sample is part of the training part of the dataset, then the object of the Dataset class will return it with some augmentation, otherwise (if the sample is part of the validation or test part) it will be returned without augmentations. The Dataset class is used in the Dataloader class. This class is responsible for creating batches of the data and shuffling the test part of the dataset. In this project, it is the only functional of the Dataloader class that was used. Whenever the neural network requires a new batch of data, it asks the Dataloader class for it, which uses the Dataset class to load a few images and provide them to the neural network.

Batches received from the object of the Dataloader class are used in the training and validation of neural network. The TrainingModule class is responsible for this. This class is developed to make training of the neural network easy. This class has a huge amount of functionality, however, in the project only a few things were used. The object of the TrainingModule class is responsible for:

1. Training the neural network for a few epochs, so the neural network will use in its training each sample of the training part of the dataset a few times.

2. Calculating metrics during training and validation. It's extremely important because early stopping is used in the training pipeline.
3. Creating checkpoints of the model. If the target metric of the model doesn't improve after a few epochs the training is finished and a few best iterations are saved.

The overall process flow can be described like this:

1. All images saved in Google Colab. All paths are saved in RAM.
2. When an image is required for training or validation it is loaded into RAM. If the image is part of the training part of the dataset augmentations are applied to it.
3. Required images are combined into a batch.
4. The batch is fed to the neural network.
5. The metrics of the batch are computed.
6. Weights of the neural network are adjusted based on metrics.
7. When all images of the training part of the set are used (it's called epoch), the images of the validation part are fed to the neural network.
8. The metrics of the validation part are computed.
9. If the metrics of validation part don't improve after a few epochs training stops.
10. The neural network is validated on the test part of the dataset.

4.2.4 Deployment

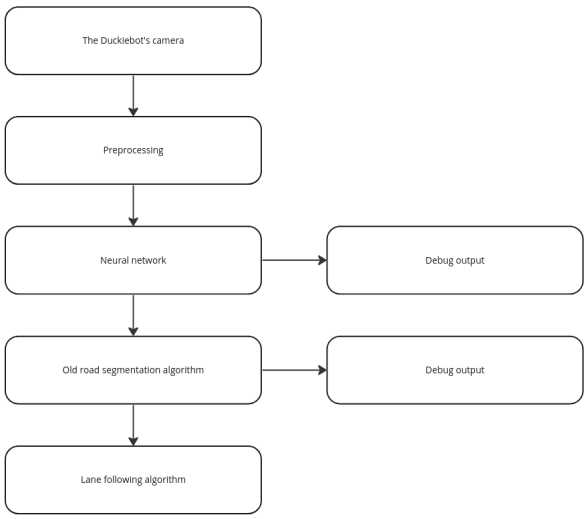


Figure 11: Deployment

The resulting architecture that is used on the Duckeibot is shown in figure 11. The image received from the camera of the Duckiebot is preprocessed. The upper part of the image is removed, because usually it doesn't contain road. After that image is converted to a torch-GPU-tensor. Then this tensor is fed to the neural network. The output of the neural network is fed into the old lane segmentation algorithm, to convert the output of the neural network to the input of the lane-following algorithm. The result is fed to the lane-following algorithm, which controls the Duckiebot's movements based on the road markup.

It was decided to use the old lane segmentation algorithm to increase the speed of the development and reduce the risk of creating bugs. However, the old algorithm isn't fully used. Only the part that is responsible for converting the pixel map to a special input format of the lane-following algorithm is used. The part of creating pixel map is removed, because this functionality is taken by the neural network, which was the initial goal of the project.

Also important to mention that the output of the neural network (pixel map) and old lane segmentation (segments of road markup) are sent to a special debug ROS node, which can visualize in real-time the images produced by these algorithms. To create these pictures some time is required, so if the debug node isn't used (the window with debug information isn't opened) these images won't be sent, which saves some time.

4.3 EM algorithm

4.3.1 Simple case

Assume there are some amount of points in d-dimensional space, they are samples. The task is to segment them. Assume that the picture above is generated by a mixture of three Gaussians.

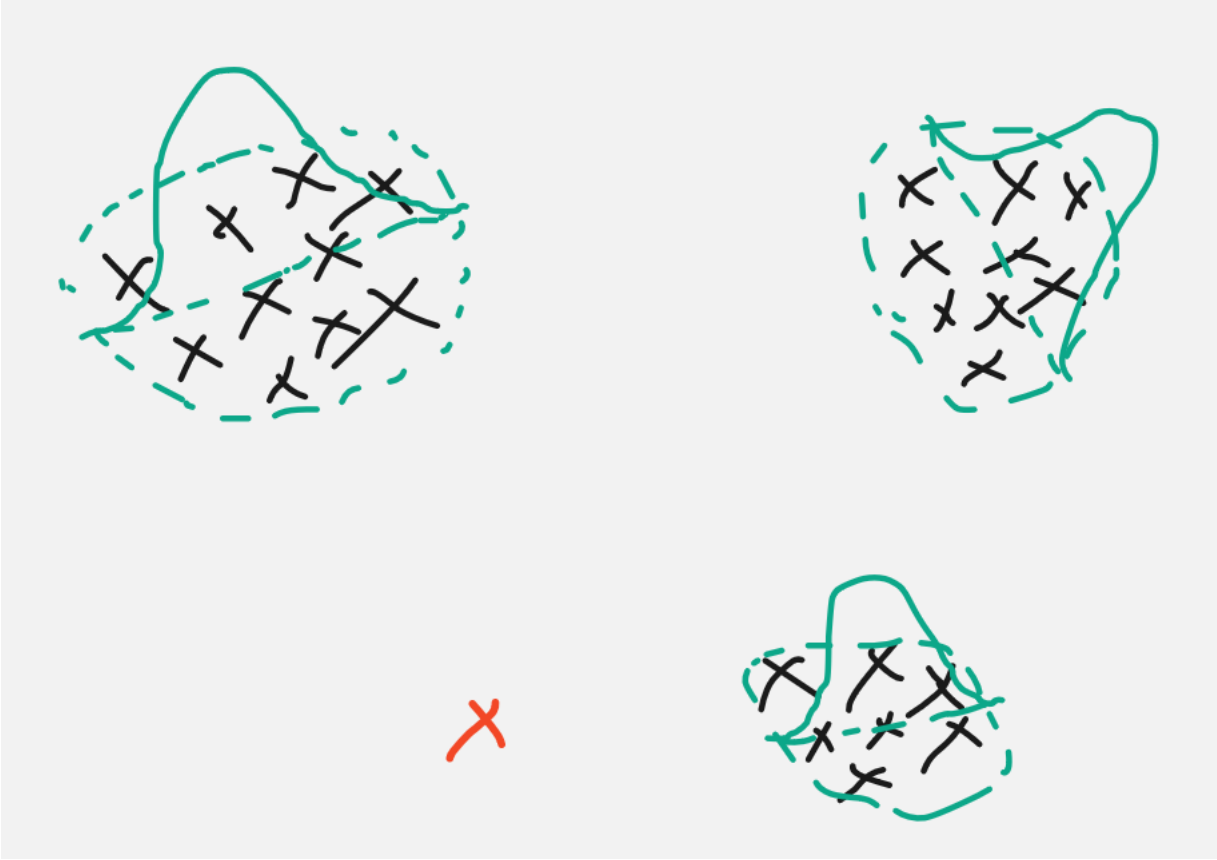


Figure 12: Data visualisation

$$p(\overline{x}) = \pi_1 \cdot p_1(\overline{x}) + \pi_2 \cdot p_2(\overline{x}) + \pi_3 \cdot p_3(\overline{x})$$

where

$$\sum_{k=1}^{K} \pi_k = 1 \text{ where } K = 3$$

In other words, with probability π_i Gaussian is chosen and after that, the sample is generated based on this Gaussian $\overline{x} \sim p_k(\overline{x})$. But how to classify a new point (the red one)? The likelihood function is

required:

$$p(D|\overline{\theta}) = \prod_{n=1}^{N} p(\overline{x}_n|\overline{\theta}) \text{ here D means whole dataset, in other words } D = \{\overline{x}_n\}_{n=1}^{N}$$

Where

$$p(\overline{x}|\overline{\theta}) = \sum \pi_{k\cdot k}\,(\overline{x}) = \sum_k \pi_k \cdot \mathcal{N}(\overline{x}|\overline{\mu}_k, \Sigma_k)$$

Where $\overline{\mu}_i$ and Σ_i are parameters of PDF of normal distribution and $\overline{\theta} = (\pi_1, \dots \pi_K, \overline{\mu}_1, \Sigma_1, \dots, \overline{\mu}_K, \Sigma_K)$

So after that:

$$p(D|\overline{\theta}) = \prod_{n=1}^{N} p(\overline{x}_n|\overline{\theta}) = \prod_{n=1}^{N} (\pi_1 \cdot p_1(\overline{x}_n|\overline{\theta}_1) + \pi_2 \cdot p_2(\overline{x}_n|\overline{\theta}_2) + \dots + \pi_K \cdot p_K(\overline{x}_n|\overline{\theta}_K)) \underset{\overline{\theta}}{\to} \max$$

Where $\overline{\theta}_i = (\overline{\mu}_i, \Sigma_i)$

The problem is that it is a multiplication of a big amount of sums which is difficult to optimize. The idea is to look at the distribution, likelihood and generation process and think about what other information about this process is required to make the likelihood function easy to optimize.

It turns out that adding to the $\overline{x}$ label of its class makes everything much easier. In math terms:

$$Z = \{\overline{z}_n\}_{n=1}^{N}, \text{ where each } \overline{z}_n \text{ is one-hot encoded: } \overline{z}_n = (0, \dots, 1, \dots 0)$$

and only 1 of this vector is on the kth position, which means that $x_n \in C_k$, where C_k means cluster number k. So if Z is known, then

$$p(D, Z|\overline{\theta}) = \prod_{n=1}^{N} p(\overline{x}_n, \overline{z}_n|\overline{\theta}) = \prod_{n=1}^{N} p(\overline{z}_n|\overline{\pi}) \cdot p(\overline{x}_n|\overline{z}_n, \overline{\theta}) = \prod_{n=1}^{N}\prod_{k=1}^{K} (\pi_k \cdot \mathcal{N}(\overline{x}_n|\overline{\mu}_k, \Sigma_k))^{z_{nk}}$$

Now it's good practice to take the logarithm of likelihood function because it transforms the product into a sum which is a more convenient operation to work with

$$\log(p(D, Z|\overline{\theta})) = \sum_{n=1}^{N}\sum_{k=1}^{K} z_{nk}\cdot(\log(\pi_k)+\log(\mathcal{N}(\overline{x}_n|\overline{\mu}_k, \Sigma_k))) = \sum_{k=1}^{K}\log(\pi_k)\cdot\sum_{n=1}^{N} z_{nk} + \sum_{k=1}^{K}\Big(\sum_{n=1}^{N} z_{nk}\cdot\log(\mathcal{N}(\overline{x}_n|\overline{\mu}_k, \Sigma_k)$$

Now important to notice that the first and the second summands depend on different parameters. The first summand depends on π_k, while the second depends on parameters of PDF of normal distribution $\overline{\mu}_k, \Sigma_k$. So it's possible to optimize these summands separately.

And now the main idea. There is a complicated likelihood function with latent variables that are not known, but if they were everything would be easy. So the idea is to split the process into two steps:

1. E-step (fix $\overline{\theta}$, find $\mathbb{E}[Z]$):

$$\mathbb{E}[z_{n,k}] = p(C_k|\overline{x}_n, \overline{\theta}) = \frac{p(C_k|\overline{\theta}) \cdot p(\overline{x}_n|C_k, \overline{\theta})}{\sum\limits_{l=1}^{K} p(C_l|\overline{\theta}) \cdot p(\overline{x}_n|C_l, \overline{\theta})} = \frac{\pi_k \cdot \mathcal{N}(\bar{x}_n|\bar{\mu}_k, \Sigma_k)}{\sum\limits_{l=1}^{K} \pi_l \cdot \mathcal{N}(\bar{x}_n|\bar{\mu}_l, \Sigma_l)}$$

2. M-step (fix $\mathbb{E}[Z]$, maximize $\mathbb{E}[\log(p(D, Z|\overline{\theta}))]$). In most cases, instead of Z, we can use soft estimates $\mathbb{Z}$:

$$\mathbb{E}[\log p(D, Z|\bar{\theta})] = \mathbb{E}\Big[\sum_n\sum_k z_{nk} \cdot \big(\log(p(\overline{x}_n|\overline{\mu}_k, \Sigma_k)) + \log(\pi_k)\big)\Big]$$

$$= \sum_n\sum_k \mathbb{E}[z_{nk}]\cdot(\log(\pi_k)+\log\big(p(\overline{x}_n|\overline{\mu}_k, \Sigma_k))) = \sum_k\Big(\sum_n \mathbb{E}[z_{n,k}]\Big)\log\pi_k + \sum_k\sum_n \mathbb{E}[z_{n,k}]\cdot\log p(\bar{x}_n|\bar{\mu}_k, \Sigma_k)$$

4.3.2 General case

There are:

1. X the dataset
2. Z latent variables we don't know them
3. θ parameters we also don't know them

The goal is to maximize $p(X|\theta)$, which is a complicated task, but optimizing $p(X, Z|\theta)$ is easier. So let

$$Q(\theta, \theta^{(n)}) := \mathbb{E}_{p(Z|X,\theta^{(n)})}\big[\log p(X, Z|\theta)\big] = \int \log p(X, Z|\theta) \cdot p(Z|X, \theta^{(n)})dz$$

And on each step:

$$\theta^{(n+1)} = \underset{\theta}{argmax} Q(\theta, \theta^{(n)})$$

These steps are executed until θ stops changing or the likelihood function stops changing (theoretically it should happen simultaneously). Also, the likelihood function should increase after each step. So in the end:

$$\log(p(X|\theta)) > \log(p(X|\theta^{(n)}))$$

$$\log(p(X|\theta)) - \log(p(X|\theta^{(n)})) = \log\Big(\int p(X, Z|\theta)dz\Big) - \log(p(X|\theta^{(n)}))$$

$$= \log\Big(\int p(Z|X, \theta^{(n)} \cdot \frac{p(X, Z|\theta)}{p(Z|X, \theta^{(n)})})dz\Big) - \log(p(X|\theta^{(n)}))$$

$$= \log\Big(\mathbb{E}_{p(Z|X,\theta^{(n)})}[\frac{p(X, Z|\theta)}{p(Z|X, \theta^{(n)})}]\Big) - \log(p(X|\theta^{(n)}))$$

$$\underset{\text{by the Jensen's inequality}}{\geq} \mathbb{E}_{p(Z|X,\theta^{(n)})}\Big[\log\big(\frac{p(X, Z|\theta)}{p(Z|X, \theta^{(n)})}\big)\Big] - \log(p(X|\theta^{(n)}))$$

$$= \mathbb{E}_{p(Z|X,\theta^{(n)})}\Big[\log\big(\frac{p(X, Z|\theta)}{p(Z|X, \theta^{(n)}) \cdot p(X|\theta^{(n)})}\big)\Big]$$

Now important to notice, that $p(Z|X, \theta^{(n)}) \cdot p(X|\theta^{(n)}) = p(X, Z|\theta^{(n)})$

And therefore:

$$\log(p(X|\theta)) \geq \log(p(X|\theta^{(n)})) + \mathbb{E}_{p(Z|X,\theta^{(n)})}\Big[\log\big(\frac{p(X, Z|\theta)}{p(Z|X, \theta^{(n)}) \cdot p(X|\theta^{(n)})}\big)\Big] =: \mathcal{L}(\theta, \theta^{(n)})$$

As shown in the picture in each point ($\theta^{(n)}$, $\theta^{(n+1)}$, $\theta^{(n+2)}$, etc) the lower score of $\log(p(X|\theta))$ is known. This score is $\mathcal{L}(\theta, \theta^{(n)})$

Also, it's important to mention that $\log(p(X|\theta^{(n)})) = \mathcal{L}(\theta^{(n)}, \theta^{(n)})$

So the idea is to optimize $\mathcal{L}(\theta, \theta^{(n)})$. The function converges to a local maximum which is also good.

But in the beginning, it was said that $Q(\theta, \theta^{(n)})$ needs to be optimized, it turns out that in terms of optimization Q and $\mathcal{L}$ are the same functions.

$$\mathcal{L}(\theta, \theta^{(n)}) = \underset{const(\theta)}{\log(p(X|\theta^{(n)}))} + \underset{\text{it is } Q(\theta,\theta^{(n)})}{\mathbb{E}_{p(Z|X,\theta^{(n)})}\Big[\log\big(p(X, Z|\theta)\big)\Big]} - \underset{const(\theta)}{\mathbb{E}_{p(Z|X,\theta^{(n)})}\Big[\log\big(p(X, Z|\theta^{(n)})\big)\Big]}$$

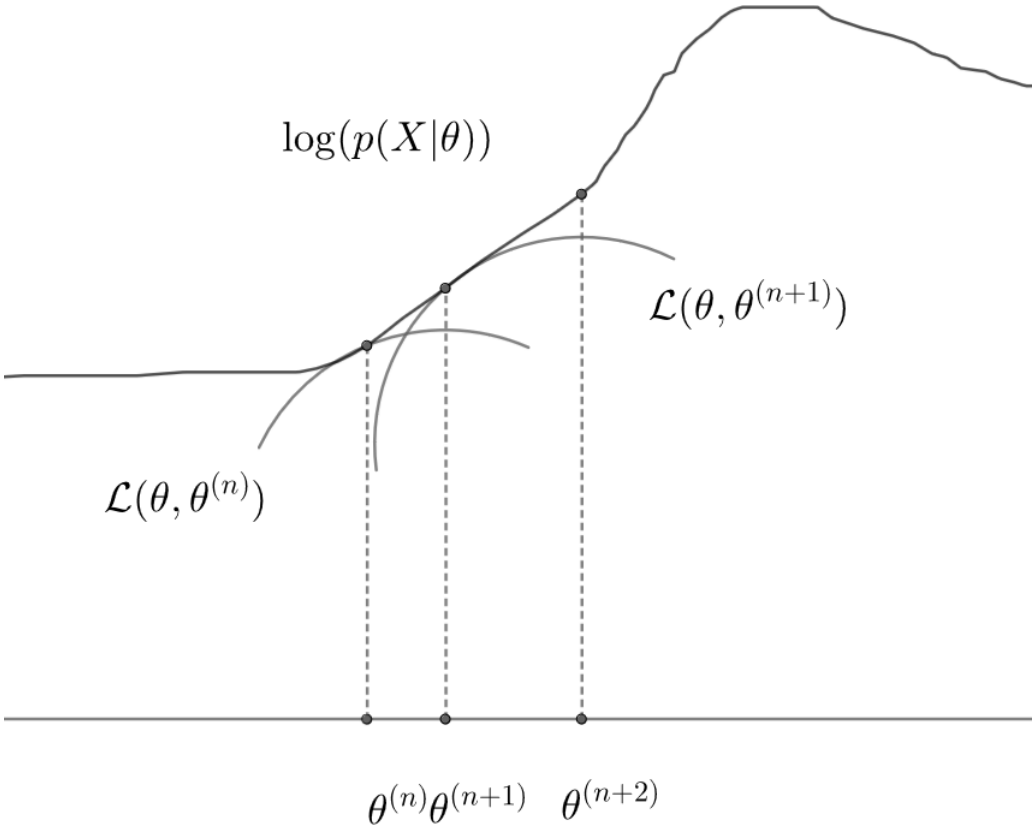


Figure 13: Algorithm visualisation

4.4 Deep learning model

4.4.1 Architecture

The architecture of the model is shown in Figure 14. The idea of MobileNet [Howard et al., 2017] for splitting the convolutional layer into two parts was used to increase the speed of the neural network. As the loss function, the cross-entropy function was used.

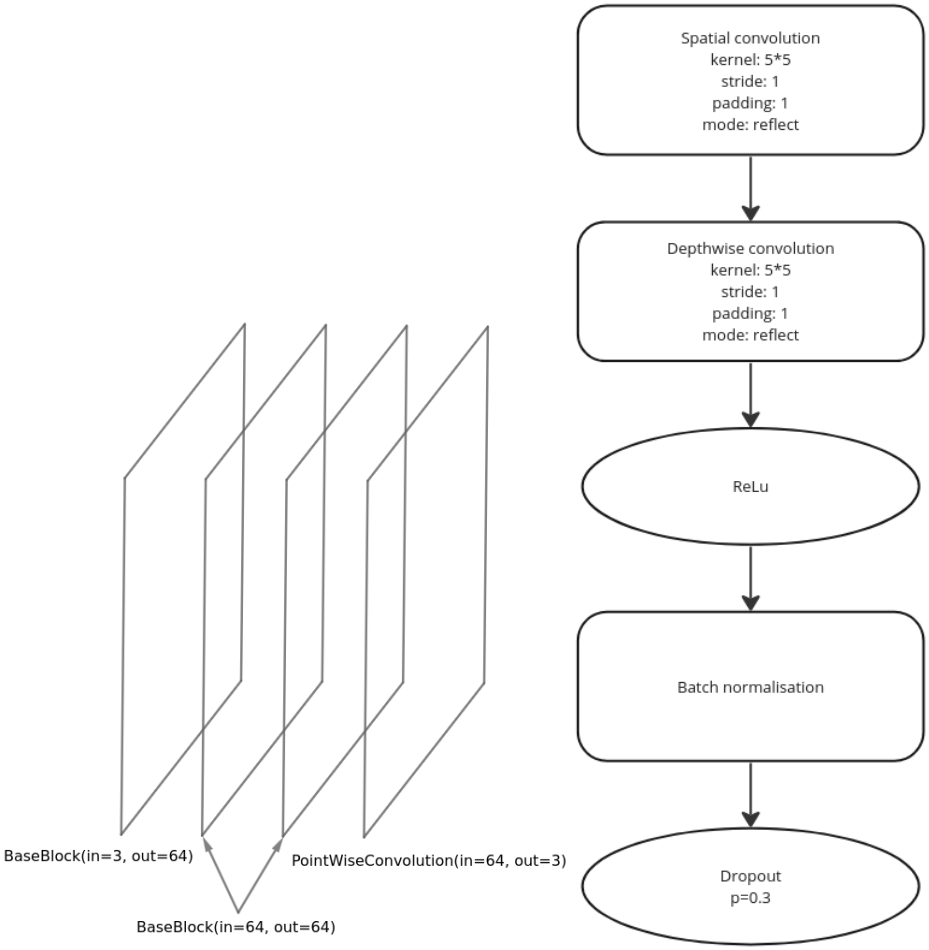
$$\mathcal{L}(x, y) = \{l_1, \dots, l_n\}; l_n = -w_{y_n} \log \frac{e^{x_{n,y_n}}}{\sum_{c=1}^{C} e^{x_{n,c}}}$$

where $x \in \mathbb{R}^{n \times C}$ are outputs of the model and $y \in \mathbb{N}^n$ are ground truth class labels for each sample

For the road markup segmentation task, this architecture was enough. Scores of the model are shown in the Implementation chapter. Each layer is described in more detail below.

4.4.2 General neural networks ideas

To talk about convolutional neural networks it's important to discuss the idea of neural networks in general.



(a) Full model architecture (b) Basic block architecture

Figure 14: The DL model architecture

Each neural network can be represented as a computational graph. The first node of the graph is the input and the last is the output of the network. The output of the neural network (in formulas $\hat{y}$ is used) passes to the loss function. Almost always the task of training is to minimize the loss function. To do this backpropagation is used. Let $X \in \mathbb{R}^{n \times d}$ is input of the neural network, $\hat{y} \in \mathbb{R}^{n \times k}$ is output, $\{\hat{y}_i \in \mathbb{R}^{n \times k_i}\}_{i=1}^{H}$ is the output of the ith layer (or node in terms of the graph) of the neural network and $y \in \mathbb{R}^{n \times m}$ is a vector of ground truth answers (real numbers, class labels, etc). Also each layer has weights $\{w_i \in \mathbb{R}^{k_i}\}_{i=1}^{H}$ (bias is included in weights for convenience) Here n is used as the size of a dataset used for training and H is the number of layers of a neural network (or nodes in terms of a graph)

In the described terms backpropagation can be described like this:

- $\dfrac{\partial L}{\partial \hat{y}} =: \dfrac{\partial L}{\partial y_H}$ is the partial derivative of the loss function by the output of the neural network
- $\forall i \in \{1 \dots H-1\} \dfrac{\partial L}{\partial y_i} = \dfrac{\partial L}{\partial y_{i+1}} \cdot \dfrac{\partial y_{i+1}}{\partial y_i}$

The training is an iterative process, so $w_i^{(j)}$ mean weights of layer i on the jth iteration of the process. The process can be described like this:

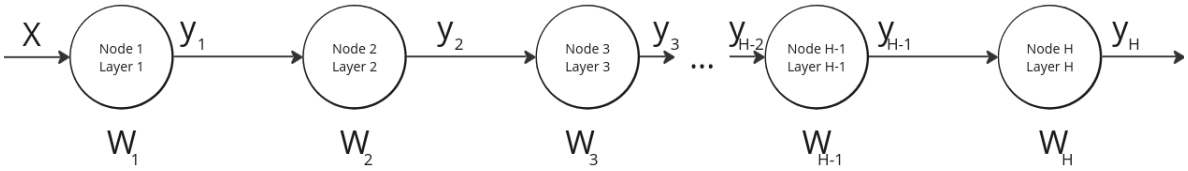


Figure 15: Computational graph

- $\forall i \in \{2 \dots H\}\ w_i^{(j+1)} = w_i^{(j)} - \eta \cdot \frac{\partial L}{\partial w_i}\Big|_{w_i^{(j)}, y_{i-1}}$
- $w_1^{(j+1)} = w_1^{(j)} - \eta \cdot \frac{\partial L}{\partial w_1}\Big|_{w_1^{(j)}, X}$

All gradients and things specific to each layer used in the provided architecture are listed below

4.4.3 Convolutional layer

The convolutional layer is called this way because this layer performs a math transformation called convolution:

$$(f * g)(x) := \int_{\mathbb{R}^n} f(x-y)g(y)dy = \int_{\mathbb{R}^n} f(y)g(x-y)dy$$

However, in convolutional neural networks (CNN) this operation is usually performed iteratively because it's more efficient. To implement the convolutional layer iteratively the sliding window is used.

Let $X \in \mathbb{R}^{N \times D \times H \times W}$ be the batch of input images (or feature maps) if it is an image D usually equals 3 and means the number of colors channels, W and H are the widths and the heights of the image or feature map and N is the batch size (amount of samples of training set simultaneously fed to a neural network in a single forward pass). $K =\in \mathbb{R}^{L \times D \times H_k \times W_k}$, where $W_k = 2 \cdot a + 1$ and $H_k = 2 \cdot b + 1$ is the kernel (sliding window).

The convolution of X and K ($X * K$) is another tensor $Y \in \mathbb{R}^{N \times L \times W' \times H'}$, where [PyTorchContributors, 2023]

$$W' = \left\lfloor \frac{W + 2 \cdot p_1 - d_1 \cdot (W_k - 1) - 1}{s_1} + 1 \right\rfloor$$

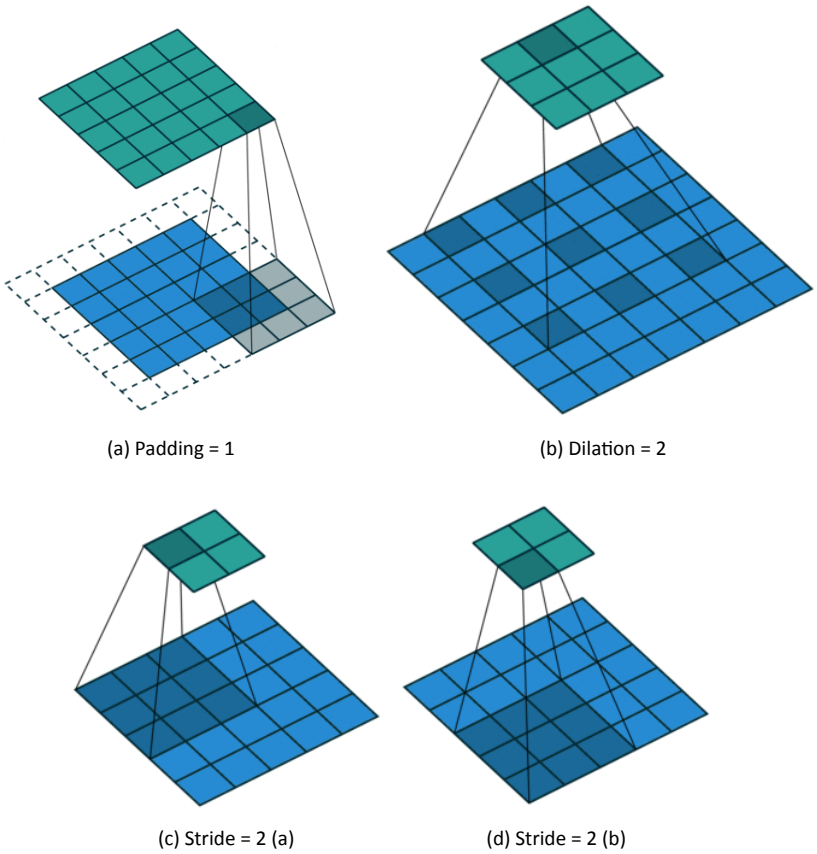
$$H' = \left\lfloor \frac{H + 2 \cdot p_0 - d_0 \cdot (H_k - 1) - 1}{s_0} + 1 \right\rfloor$$

- p_i padding of spatial dimensions of the input tensor
- d_i is dilation. Spacing between kernel elements
- s_i is stride. A stride of the convolution

The visualisation of each parameter is shown on Figure 16 [Prove, 2017]

The elements of the result tensor are computed in the following way:

$$Y_{n,k,h,w} = \sum_{d=0}^{D} \sum_{j=-\lfloor W_k/2 \rfloor}^{\lfloor W_k/2 \rfloor} \sum_{i=-\lfloor H_k/2 \rfloor}^{\lfloor H_k/2 \rfloor} X_{n,d,h \cdot s_0 + i \cdot d_0, w \cdot s_1 + j \cdot d_1} \cdot K_{k,d,i+\lfloor H_k/2 \rfloor, j+\lfloor W_k/2 \rfloor}$$



(a) Padding = 1 (b) Dilation = 2

(c) Stride = 2 (a) (d) Stride = 2 (b)

Figure 16: Visualisation of convolution layer parameters

The gradients of this layer can be computed like this (only the most common case, where $p_0 = \left\lfloor \frac{H_k}{2} \right\rfloor, p_1 = \left\lfloor \frac{W_k}{2} \right\rfloor$ and $s_0 = s_1 = d_0 = d_1 = 1$ is considered because only this case was used in the project's architecture, important to notice that in this case $W' = W$ and $H' = H$):

- To compute $\frac{\partial L}{\partial X}$ it's need to define $\widehat{K} \in \mathbb{R}^{D \times L \times W_k \times H_k} : \widehat{K}_{d,l,w,h} = K_{l,d,h,w}$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} * \widehat{K}$$

- Some kind of convolution is also used to calculate $\frac{\partial L}{\partial K}$, but summation takes place by the size of the batch, not by channels

$$\frac{\partial L}{\partial K}_{l,d,h,w} = \sum_{n=0}^{N} \sum_{i=0}^{H} \sum_{j=0}^{W} X_{n,d,i+h-\lfloor \frac{H_k}{2} \rfloor, j+w-\lfloor \frac{W_k}{2} \rfloor} \cdot \frac{\partial L}{\partial Y}_{n,l,i,j}$$

Important notice
$X_{i,-1,-1}$ is valid notation. It means that padded values are used. The mode of padding can be different in this project reflect mode is used.

4.4.4 Batch normalisation layer

This type of layer was first introduced in 2015 in this paper [Ioffe and Szegedy, 2015]. This layer is used to keep all data in a neural network in one type of distribution. This technique prevents gradient

explosions and helps a neural network to converge faster. The idea of the batch normalization layer is pretty simple. During training update count mean and variance for the batch and apply it to training data and during inference just apply running estimates of those values to the data.

The forward pass of the layer during training can be described in the following way:

1. Count batch statistics:
   - $\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$
   - $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$
2. Normalize the batch: $\hat{x_i} = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$
3. Scale the batch: $y_i = \gamma \cdot \hat{x_i} + \beta$, where γ and β are trainable parameters

During inference instead of μ and σ^2 $E[x]$ and $VAR[x]$ are used which were calculated during training.

Gradients of this layer are pretty simple:

1. $\frac{\partial L}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i} \cdot \gamma$
2. $\frac{\partial L}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot (x_i - \mu) \cdot \frac{-1}{2} \cdot (\sigma^2 + \varepsilon)^{\frac{-3}{2}}$
3. $\frac{\partial L}{\partial \mu} = \left(\sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2 + \varepsilon}}\right) + \frac{\partial L}{\partial \sigma^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu)}{m}$
4. $\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial L}{\partial \sigma^2} \cdot \frac{2(x_i - \mu)}{m} + \frac{\partial L}{\partial \mu} \cdot \frac{1}{m}$
5. $\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \cdot \hat{x}_i$
6. $\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}$

4.4.5 ReLu activation

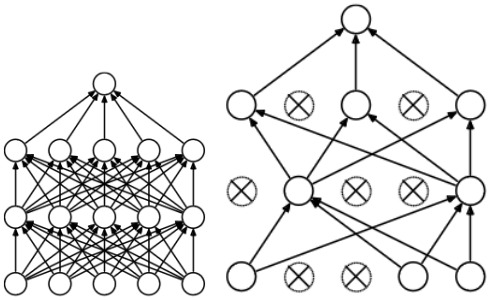
ReLu is used in the project's architecture as an activation function. This function proved itself to be a good activation and some kind of baseline in modern deep learning [Tomar and Laxkar, 2022]

$$ReLu(x) = y = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

The gradient of ReLu is obvious:

$$\frac{\partial L}{\partial x} = \begin{cases} 0 & \text{if } x \leq 0 \\ \frac{\partial L}{\partial y} & \text{if } x > 0 \end{cases}$$

4.4.6 Dropout layer



(a) Standard neural network (b) After dropout

Figure 17: Dropout layer

The main idea of Dropout is to randomly and temporarily exclude neurons from the network during training. At the same time, each neuron has the probability of being ”turned off” at each iteration of network training. This forcibly makes the network less capable of memorizing specific patterns, which helps the network generalize its knowledge to new data.

Using Dropout usually improves the generalizing ability of the model and prevents overfitting, especially in cases where the training sample is small or when the model has a large number of parameters.

If usually neural network works like this:

1. $\overline{z}^{(l)} = \overline{w}^{(l)} \cdot \overline{y}^{(l-1)}$
2. $\overline{y}^{(l)} = f(\overline{z}^{(l)})$

With the dropout layer, its behavior can be described in the following way:

1. $r^{(l)} \sim Bernoulli(p)$
2. $\tilde{y}^{(l)} = r^{(l)} \odot y^{(l)}$, where $\odot$ means element-wise multiplication
3. $\overline{z}^{(l)} = \overline{w}^{(l)} \cdot \tilde{y}^{(l-1)}$
4. $\tilde{y}^{(l)} = f(\overline{z}^{(l)})$

4.4.7 Optimizers

As was written above in neural networks updating weights can be implemented in the following way:

$$W^{(n)} = W^{(n-1)} - \eta \nabla_W \mathcal{L}(W^{(n-1)}, X, y)$$

Where W is the tensor with model weights, X is the input tensor and y is the ground truth tensor.

However, this approach has some disadvantages [Ruder, 2017]:

1. Convergence problem, using classical stochastic gradient descent (SGD) usually doesn't guarantee good convergence.
2. Learning rate should be chosen wisely and updated during training.

3. The same learning rate is applied to all weights, which can be a problem since not all weights are in the same distribution and scale.
4. SGD can lead to a local minimum of a loss function, which sometimes can cause problems during inference.

To solve these problems a lot of different approaches for updating weights were invented. In this project, two of them were tried and one of them was finally used.

SGD Momentum

SGD encounters challenges when navigating through ravines, where the terrain of gradient sharply curves in one dimension more than another, often found near local optima. In such cases, SGD tends to zigzag along the slopes of the ravine, making slow progress toward the local optimum.

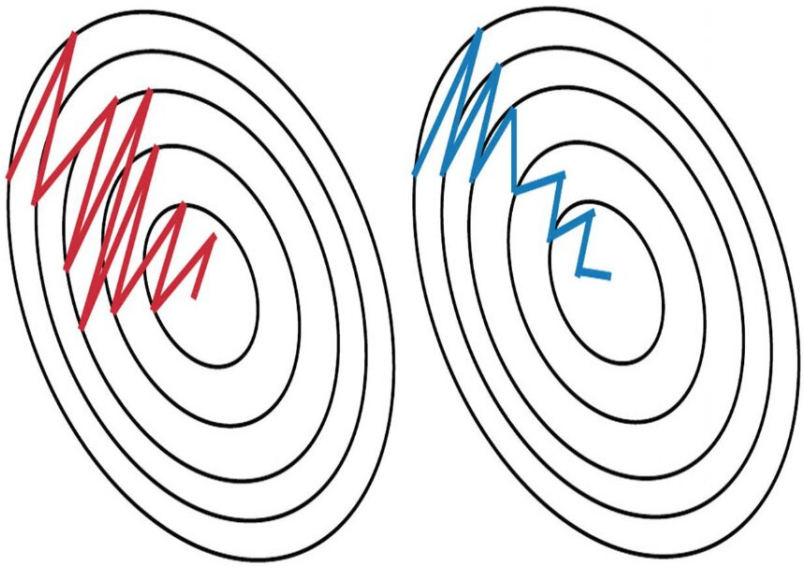
Momentum, however, addresses this issue by accelerating SGD in the direction of relevance while dampening oscillations. This is achieved by incorporating a fraction of the update tensor from the previous time step into the current update tensor.

$$v^{(n)} = \gamma \cdot v^{(n-1)} + \eta \nabla_W \mathcal{L}(W^{(n-1)}, X, y)$$

$$W^{(n)} = W^{(n-1)} - v^{(n)}$$

$\gamma \cdot v^{(n-1)}$ is called momentum. It is usually set to 0.9 [Ruder, 2017].

In other words, parameter updates accumulate momentum, accelerating convergence where gradients coincide and reducing measurement updates with varying gradients. Consequently, the pulse promotes faster convergence and mitigates fluctuations.



(a) SGD without monmentum (b) SGD with momentum

Figure 18: SGD

Adam

If SGD momentum solves the convergence problem, then Adaptive Moment Estimation (Adam) [Kingma and Ba, 2017] solves other problems, like learning rate scheduling, different learning rates for different weights and of course convergence problems. The algorithm of Adam can be described below:

$$g^{(n)} = \nabla_W \mathcal{L}(W^{(n-1)}, X, y) + \lambda \cdot W^{(n-1)}$$

$$m^{(n)} = \beta_1 \cdot m^{(n-1)} + (1 - \beta_1) g^{(n)}, \text{ where } m^{(0)} = 0$$

$$v^{(n)} = \beta_2 \cdot v^{(n-1)} + (1 - \beta_2)(g^{(n)})^2, \text{ where } v^{(n)} = 0$$

$$\widehat{m}^{(n)} = \frac{m^{(n)}}{1 - \beta_1^n}$$

$$\widehat{v}^{(n)} = \frac{v^{(n)}}{1 - \beta_2^n}$$

$$W^{(n)} = W^{(n-1)} - \eta \cdot \frac{\widehat{m}^{(n)}}{\sqrt{\widehat{v}^{(n)}} + \varepsilon}$$

Here's η is the common learning rate, β_1, β_2 parameters for computing running averages of gradient and its square, λ is weight decay coefficient, it adds L2 regularisation for the optimizer. Authors of the algorithm suggest to use $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$

The benefits of Adam are that:

1. It computes adaptive learning rates for each parameter
2. It stores bias-corrected first and second moment estimates of the exponentially decaying average of past gradients
3. It applies L2 regularisation

Even though Adam uses techniques of SGDM it's not obvious which optimizer is better [Kaur, 2024]. It turns out that sometimes SGDM performs better. That's why both optimizers were tried in the project. The choice and its reason are described in the implementation chapter.

4.5 General project design

As it was written in the Project planning chapter the project was divided into three big parts. The code base is shown in Figure 19

4.5.1 The Server

For the server, it was decided to use a simple socket architecture. The server was able to handle multiple requests simultaneously, even though the Duckiebot can't asynchronously send data (a new photo was sent to the server only after the previous one was fully sent), writing data to disk was a bit slower than receiving pictures via WiFi. So a new request could come before the previous was completed.

4.5.2 Jupiter notebooks

The use of Jupiter notebooks is usually in research projects. This technology provides the ability to execute various parts of the code without having to execute the entire pipeline. Since writing ML models is divided into analyzing the dataset, writing the model itself, training and testing it, the ability to run different parts of the code, saving variables and states after other runs is very useful. This feature also allows you to abandon writing unit and integration tests at least at the research stage, since in case of an error, you can immediately see which part of the code is not working, as well as errors similar to differences in the input format are easily identified and corrected.

In addition to all the described below benefits, Jupiter notebooks are easily convertible to regular Python files, which makes migration from the research stage to the deployment extremely easy.

Figure 19: Code base

4.5.3 Deployment on the Duckeibot

The Duckiebot is run under ROS. So to deploy something on it the ROS node should be created. It was decided not to create a new ROS node, but to use the existing one, which was used for an old lane segmentation algorithm. Moreover, the output of the created algorithm is fed into the part of the old lane segmentation algorithm. This was done to use already existing and working code to communicate with the lane-following algorithm. However, from old algorithm part with segmentation was removed and only the part with converting the color mask to required for lane-following algorithm input was saved.

5 Implementation

5.1 Introduction

This chapter is also divided into 3 sections because the implementation of each step of the project has its specificity.

5.2 Implementation of dataset collecting and segmentation

5.2.1 Server implemetation

As it was described in the Design chapter, for the server it was decided to use GoLang and a simple socket server. The using of goroutines was almost the perfect solution for the implementation of asynchronous server [Sniffin, 2022] [Kennedy, 2018]. The technology of the goroutines is extremely efficient and perhaps, even too efficient for this server, however, in addition to the speed of operation, they are extremely easy to implement.

Benchmarks for the implementation of the asynchronous task approach in different programming languages are shown in table 4.

Programming language	Benchmark results in milliseconds
GoLang	22.717915
C++	36.1088
Rust	75.88563631
Python	106.35573348

Table 4: Benchmarks of multithreading in different programming languages

The server was local, so there was no need for a static IP address, however, each day it was needed to change the IP address on the client part.

5.2.2 ROS node implementation

For the client part, it was decided to use pure Python. Although the Duckietown project has the implementation of the MQTT bridge, this technology was relatively slow in comparison to pure Python on the client side and GoLang on the server. As was shown in the previous section Go implementation of Goroutines (as a multithreading paradigm) is one of the most efficient in terms of implementation and runtime speed. So the only task was to develop a client part. Because the Duckietown project uses Python as the main language for development, it was decided to create a simple HTTP client to send data to the Go server on 'pure Python'.

5.2.3 Marking up algorithm implementation

As it was written in the chapters above the EM algorithm was used for marking up. It was decided to use Sklearn's realization of the algorithm. However, the implementation wasn't so easy. Using the whole image wasn't efficient. The upper part of the image consists of a lot of different colors, which brings a noise to mark up. In order to get rid of the noise, it was decided to crop the upper half of the image. This both reduced noise and reduced the amount of data to process, which increased the speed of markup. Also, using the EM algorithm with only one input image per forward pass brought a lot of noise, so it was decided to combine the images into batches of 500 pieces (it could not be increased due to lack of RAM). The result is a relatively high-quality markup.

5.2.4 Validation of the markup

Validation of the result markup was carried out manually. Out of 9,000 images, 500 were discarded due to poor markup. The rest of the images were used in neural network training.

5.2.5 Images in the dataset

The resulting dataset consists only of three types of images.

1. Raw cropped image. A neural network was trained on these images.
2. The mask. It's just a pixel map that was used as a ground truth for network training.
3. An image containing both a raw image and a mask. This part of the dataset is necessary for validation by other people. Since the dataset was published in the public domain, it can be supplemented with other images, because of this, some images can be replaced with less noisy ones. For the convenience of the validation process, this part of the dataset was created.

There is no other data in the dataset.

5.3 Neural network implementation

5.3.1 Choosing hyperparameters

The architecture of the deep learning model was described in the Design chapter, however, the process of choosing hyperparameters of the model is described here. Using grid search for choosing hyperparameters isn't so usual in deep learning, because it consumes a lot of time and computational resources. The first few runs of the training step of the neural network were without a grid search. A few parameters were changed during those runs but that change didn't give good enough results. So it was decided to use grid search for choosing hyperparameters. However because of the lack of computational resources not all combinations of hyperparameters were tested, but 36 combinations with the most important hyperparameters were tried. As validation metrics recall and precision were used:

$$RECALL = \frac{TP}{TP + FN}$$

$$PRECISION = \frac{TP}{TP + FP}$$

Because these metrics can be computed only for binary classification, they were computed twice: for yellow and white pixels. The result of the validation step of training is represented in the table 5. These results are not precise, because to be so, each set of hyperparameters should be validated a few times, however, because of the lack of computational resources on Google Colab these sets were validated just one time each. It took 12 hours to validate these sets and consumed almost all resources of Google Colab.

The weights of colors which are described in the table are weights for a cross-entropy loss function. As it's shown in the table these weights are crucial for the target metrics. This happens because the dataset isn't balanced in terms of color distribution. The amount of pixels with the color of the road is much more than any other color, also the amount of pixels of white color is more than yellow ones. That's why weights were required.

Optimizer	Kernel size	Yellow weight	White weight	Validation loss	Yellow recall	White recall
Adam	3	5.0	5.0	0.157	0.764	0.929
Adam	3	5.0	3.0	0.147	0.750	0.901
Adam	3	5.0	2.0	0.170	0.771	0.734
Adam	3	7.0	5.0	0.139	0.818	0.945
Adam	3	7.0	3.0	0.153	0.809	0.863
Adam	3	7.0	2.0	0.159	0.783	0.804
Adam	3	10.0	5.0	0.140	0.835	0.955
Adam	3	10.0	3.0	0.140	0.846	0.906
Adam	3	10.0	2.0	0.139	0.854	0.830
Adam	5	5.0	5.0	0.140	0.777	0.946
Adam	5	5.0	3.0	0.149	0.765	0.890
Adam	5	5.0	2.0	0.145	0.792	0.794
Adam	5	7.0	5.0	0.174	0.803	0.915
Adam	5	7.0	3.0	0.154	0.780	0.911
Adam	5	7.0	2.0	0.165	0.772	0.745
Adam	5	10.0	5.0	0.133	0.847	0.951
Adam	5	10.0	3.0	0.151	0.821	0.896
Adam	5	10.0	2.0	0.165	0.796	0.826
SGDM	3	5.0	5.0	0.173	0.802	0.904
SGDM	3	5.0	3.0	0.202	0.752	0.804
SGDM	3	5.0	2.0	0.197	0.791	0.794
SGDM	3	7.0	5.0	0.182	0.796	0.893
SGDM	3	7.0	3.0	0.177	0.805	0.854
SGDM	3	7.0	2.0	0.181	0.803	0.775
SGDM	3	10.0	5.0	0.180	0.832	0.917
SGDM	3	10.0	3.0	0.210	0.805	0.834
SGDM	3	10.0	2.0	0.184	0.849	0.874
SGDM	5	5.0	5.0	0.162	0.806	0.907
SGDM	5	5.0	3.0	0.181	0.794	0.811
SGDM	5	5.0	2.0	0.224	0.738	0.690
SGDM	5	7.0	5.0	0.177	0.806	0.895
SGDM	5	7.0	3.0	0.203	0.779	0.819
SGDM	5	7.0	2.0	0.186	0.811	0.768
SGDM	5	10.0	5.0	0.178	0.869	0.933
SGDM	5	10.0	3.0	0.192	0.827	0.814
SGDM	5	10.0	2.0	0.203	0.825	0.787

Table 5: Results of validation step with different hyperparameters

5.3.2 Dataset preprocessing and augmentations

As was described in the collecting of the Dataset section for the neural network input images also cropped in the top. During training, the augmentations were also applied to all images to emulate the light of LEDs (images with real light of LEDs can't be marked up by the EM algorithm. The resulting markup would be extremely noisy). Used augmentations:

1. Random brightness and contrast and random tone curve — to emulate even more different light conditions/
2. RGB shift and HSV shift to emulate LED light.

5.3.3 Libraries

During the implementation of the neural network different libraries were used:

1. PyTorch for implementation of the neural network and training pipeline
2. Albumentation for creating augmentations
3. TorchVision and OpenCV for preprocessing training data
4. TorchLightning for implementation training pipeline

Also, the Tensorboard extension for Jupiter notebooks was used for visualization changes of the target metrics during training.

5.4 Deployment

5.4.1 Introduction

During the deployment step, a lot of problems were discovered. Also single run of the algorithm on the Duckiebot consumed a lot of time to build and start the Docker image. In this section solutions to these problems will be described as well as the general workflow of the deployment step.

5.4.2 Choosing the Docker image

To implement the neural network the PyTorch library with Cuda integration was required. firstly it was decided to take a basic Cuda image and after that add an empty duckietown-core image to it. However, in the Duckietown Github, a basic Duckietown image with included Cuda was found. So it was decided to use this image.

However, during implementation troch2trt and TorchVision libraries were required for the image. Installing these libraries via standard pip instructions wasn't available, so it was decided to include instructions for these installations in the Dockerfile. So finally a new base Docker image was created and published as a template image. Also, this image is used in the project.

5.4.3 Deploying the neural network

To deploy the neural network on the bot two things were required:

1. Import code of the neural network
2. Import weights of the neural network

These steps were done.

After that, the neural network was run in test mode. It wasn't used as input for the lane-following algorithm, but it published the resulting masks into the debug ROS topic, they were later visualized in the special tool. During this step, it was discovered that neural network has two problems:

1. The whole pipeline of image processing took about 120 milliseconds.
2. The neural network required a few iterations to speed up.

The first problem was partly solved, but the second remained actual, so it was decided to run a few iterations of the algorithm before sending a signal that the Duckiebot is ready to work.

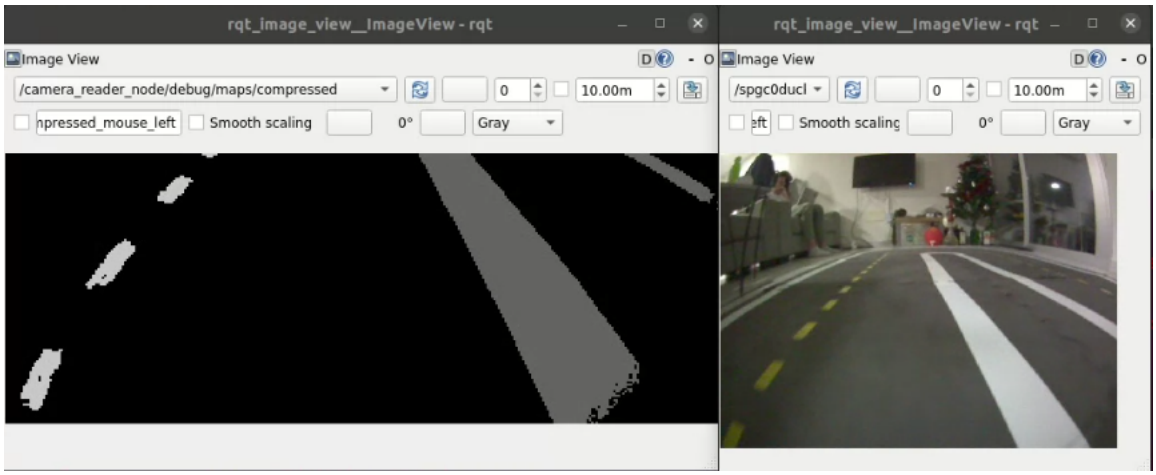


Figure 20: Output of the first run

5.4.4 Torch2trt library

To solve the first problem described in the previous section it was decided to use the torch2trt library. This library provides an opportunity for easy conversion of torch tensors to TensorRT tensors. TensorRT is SDK by Nvidia for high-performance tensor computations. Jetson Nano was developed by Nvidia so there was a hope, that using this library could increase the speed of computations. This hope was justified. Using torch2trt increased performance twice. However, Installing torch2trt wasn't an easy task as it was mentioned in the Docker image subsection. This library couldn't be installed via pip because of the specificity of the hardware, building this library from source code was also tricky because the full installation wasn't available due to outdated software in the base Duckietown docker image, which couldn't be updated because of backward compatibility problems. Luckily partial installation could be performed and this type of installation provided the required speed boost so this half-measure was left.

5.4.5 Merging neural network and lane-following algorithm

The merging process was a pretty difficult step. The lane-following algorithm accepts input as a list of segments. However, the neural network output is a pixel map of road markup. So it was decided to use the existing algorithm of road markup segmentation but without its main part. the road markup segmentation part was removed from the existing algorithm. The pipeline is described below:

1. Output of neural network is transformed into a multichannel image.
2. This image is colored into the desired colors: black, white and yellow.
3. The result is fed into the existing algorithm.

As a result, the time consumed per image increased by 20 milliseconds, but this approach significantly decreased the time for implementation. In addition, because this algorithm was already in use the problem of bugs was also solved.

5.4.6 Conclusion

As a result, the full pipeline for one image takes about 50–70 milliseconds which is a pretty good result. This speed is more than enough for lane following algorithm to make decisions on the next action of the Duckiebot.

5.5 Chapter conclusion

As it was written in this chapter implementation of the project wasn't an easy task. However, the project was done. There was a lot of research done and a lot of code was written, but most of this code was just a utility. The algorithm itself is really small in terms of code volume, however, there was a lot of code behind it, which was required to write this small amount.

6 Testing

6.1 Introduction

The main goal of this project was to develop an algorithm for autonomous marking up of a dataset and create an algorithm for the detection of road markup based on deep learning algorithms. The specifics of the project imply a special approach to code testing, which will be discussed in this chapter. It is important to notice that all parts of the project that are not machine learning algorithms were auxiliary and were run strictly under the supervision of the developer, which makes testing of these parts of the project not optional, but at least questionable.

6.2 Functional testing

6.2.1 Machine learning algorithms automated testing

Testing is included in the mandatory pipeline for creating a machine-learning algorithm. The selection of hyperparameters described in the chapter above, as well as the choice of the model architecture is based on the results of running the model on a separate part of the dataset, which was not used in training. This part is called the test part.

This approach allows one to achieve several goals at once:

1. Check the performance of the algorithm, i.e. whether it falls on different data.
2. Check the compliance of the algorithm with the specified requirements for the format of input and output data
3. Check the correctness of the algorithms, i.e. how well it copes with the task.

If the first two points are checked quite easily (if the neural network does not meet the input/output requirements or crashes on some data, then the algorithm will fail during the run of the test sample). The third point presupposes more complex reasoning. Since machine learning algorithms almost never give a truly correct answer, it is impossible to test such algorithms only for matching the correct answers. For machine learning algorithms, it is normal sometimes to give incorrect answers, however, the purpose of training such algorithms is to minimize error cases. Based on this, hyperparameters and architecture are selected.

Therefore, it can be said that automated functional testing is embedded in the process of developing machine-learning algorithms and does not require additional code in the form of unit tests.

EM algorithm

For the EM algorithm, things get even more complicated. Since there are no correct answers in the training sample, checking the correctness of the algorithm was impossible in automatical mode. Moreover, checks based on expecting the same output for the same input, can't be performed too. Since the initialization of hidden class labels in the EM algorithm occurs randomly, this algorithm may give slightly different answers to the same input data and this cannot be called an incorrect behavior of the algorithm.

Neural networks layers unit testing

For neural networks, it is possible to write unit and integration tests when they are developed from scratch. That is, all the mathematics described in the chapters above is written by the developer himself. However, this project used the PyTorch library, which provides already-written layers of neural networks. Therefore, unit testing of these blocks is unnecessary.

Deliberately incorrect data

The only objective reason for creating functional tests for machine learning algorithms is to check the performance of algorithms with deliberately incorrect data. However, these tests were not written in this project for the following reasons:

1. The neural network is published within the Duckietown repository, the only possible input of this algorithm is a three-channel image of 480*640 pixels. Therefore, checking the input data for compliance with the specified data type is unnecessary. If the algorithm is taken from the repository and used for other purposes, I am not responsible for this in accordance with the MIT license.
2. The neural network is written exclusively for recognizing road markup. If the camera of the Dackiebot is directed at something else, the neural network will not fall but will give an unpredictable result. Checking for the presence of a road in the frame could be integrated into the algorithm, however, this would increase the time cost of processing a single frame. And since Duckiebot is supposed to be used only on a landfill simulating a road, frames with such content can be obtained either as inappropriate using Duckiebot, which is not my area of responsibility, or as a result of an error that causes the bot to leave the landfill, but such cases are exceptional and must be controlled by a human, which is why it was considered checking for the presence of a road to be an unnecessary complication of the algorithm, which will lead to a deterioration in operating time.

6.2.2 Non-machine learning algorithms automated testing

As it was written in the Introduction subsection all non-machine learning algorithms and applications were written only for inner use. Moreover, they were used only under developer control, so writing functional tests for this part of the project was unnecessary.

However, one non-machine learning part was released. It was a ROS node with an implemented neural network in it. But this node mostly consists of code of the old algorithm, so bugs can be only in machine learning part, but as it was written above, the neural network was tested during the training phase so once again the creation of any automated functional tests was redundant.

6.2.3 Manual functional testing

As it was written above it was decided to get rid of using an automated testing approach. However, leaving the project completely untested would be a terrible mistake. As a test, it was decided to do a very simple thing. Start the autopilot and look at its behavior.

As a result of this approach, minor errors were identified, which were immediately corrected. However, more significant problems with the bot have been found. The bot drove out of the lane in turns and also interpreted the white floor as part of the road markings, which sometimes led to unpredictable behavior. These errors are described in more detail in the chapter below and they will not be considered here.

6.3 Non-functional testing

6.3.1 Introduction

In fact, non-functional testing has been discussed throughout this document. Since the speed of the algorithm is a key aspect of its development. Then the speed of the algorithm was measured many times in different conditions.

6.3.2 Speed of the algorithm

The speed of the neural network was checked at the time of its creation, in Google Colab, as well as directly on Duckiebot. Based on this data, decisions were made to use the torch2trt library to reduce the execution time of the data processing pipeline.

Also, according to the results of the benchmarks, an interesting observation was made, despite the shared GPU and CPU memory, the torch library takes time to send data to the GPU memory, which looks strange. This observation paved the way for further research and development of a driver for the bot's camera so that the resulting image was immediately in PyTorch-GPU-tensor format. This does not apply to the current project, but the implementation of such a driver in theory will be able to speed up the algorithm's operation time by at least 3 times.

6.4 User testing

Since the project was a research project, it does not involve an end user. Therefore, no user testing was carried out.

6.5 Summary

At the end of the chapter, it should be noted that the testing in the project was carried out, although not in a completely ordinary execution.

In the project, it was decided to abandon unit and integration testing due to the redundancy of these tests, as well as the time to write them. The neural network was tested during its training, the EM algorithm was tested during its operation, as it was launched under the supervision of the developer. Checking the compatibility of the algorithm with the software was carried out in real conditions, since this approach, on the one hand, does not require time to write testing code, and on the other hand, makes it possible to fully assess the performance of the project.

Benchmarks of the operating time of the neural network, the neural network pipeline and the entire data processing pipeline were also carried out. Based on these data, it was decided to use additional libraries to speed up the algorithm. The benchmark results not only helped to improve the project but also laid the foundation for future developments in Duckietown.

User testing was not performed due to the absence of an end user.

The results of the testing phase were studied and taken into account during the development of the project.

7 Evaluation, Conclusions and Future Work

7.1 Project Objectives

As a result of the project, a few things were made:

1. Collected, marked up and published dataset
2. Created deep learning algorithm for road markup detection
3. The algorithm was deployed on Duckiebot

The dataset is available on my GitHub repository it is also been published in the official Duckietown Slack.

The model is also available on my GitHub with the Docker image for Duckiebot

7.2 Self-Evaluation

During this project, I've learned a lot about computer vision, docker, ROS and the Duckietown project. I also learned how to collect and publish a good dataset.

On the other hand, I've figured out some of my weaknesses. Doing paperwork is pretty hard for me. So this skill should be developed.

7.3 Project Evaluation

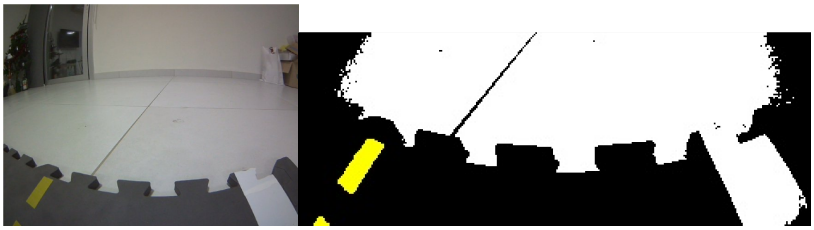
Overall the main goal was achieved. The road markup segmentation algorithm works almost as well as I wanted at the start. It is light-insensitive, doesn't change its behavior with different colors of the Duckiebot's LEDs and takes only 1 millisecond to process one image. However there are a few problems that are not connected to the algorithm itself, but affect the Duckiebot behavior, making an evaluation of the project not so obvious.

1. Transfering data from/to GPU memory is the first problem. The Duckiebot is powered by Jetson Nano by Nvidia. It has shared GPU and CPU RAM, but existing libraries and drivers available for Duckiebot do not allow it to take advantage of this. That is why transferring to and from GPU RAM takes the most time in the pipeline. To solve this problem I have to rewrite the driver which is obviously beyond the scope of the project
2. The current lane-following algorithm and servo motors are the second problem. To make the algorithm fast and precise I cut off the upper half of the input image. However, when I pass the output of my algorithm to the lane following algorithm the Duckiebot sometimes behaves weirdly. This is especially noticeable on the turns of the road. To perform a turn the Duckiebot drives out of its lane, but comes back soon after the turn. The reason for this behavior is that lane following algorithm treats This algorithm is part of the base docker image of the Duckiebot, so I can't change it. And servo motors are obviously beyond the scope of the project and the Computing University course
3. The last problem is the white floor. My algorithm usually treats the white floor as the white line of the road markup. It is not a big problem of the algorithm, but to correctly evaluate the project it is better to use rooms with dark floors for a testing polygon or enlarge a polygon with empty black tiles on the edges. In addition the neural network was tested in a few configurations. You can find the results in table 6. All this runs were pretty good in terms of accuracy, but had a difference in

Image size	NN initialisation time	First run time (average time for full NN pipeline with transfering data to and out of GPU memmory)	Average time per image (average time for full NN pipeline with transfering data to and out of GPU memmory)	Accuracy (percent of time spent, when bot was on right position)
640×120	123 seconds	320 milliseconds (3332 milliseconds)	1.3 milliseconds (113 milliseconds)	90%
320×60	107 seconds	9 milliseconds (672 milliseconds)	1 millisecond (67 milliseconds)	83%
160×30	58 seconds	6.5 milliseconds (514 milliseconds)	1 millisecond (43 milliseconds)	80%

Table 6: Project results

images per second. It was decided to use the second configuration for the resulting approach, as it's pretty accurate, but in the same time not so heavy in computartional time.



(a) Raw image of the road with floor

(b) Mask of the road with floor

(c) Raw image of the road without floor

(d) Mask of the road without floor

Figure 21: Masks for different parts of the road

7.4 Applicability of Findings to the Commercial World

This project is a research project, so it can't be directly applied in the commercial world. However, I've managed to create a good algorithm and collect a pretty huge and well-marked dataset, which can help in developing the Duckietown project. Also, my contribution can help me find a job in computer vision, especially in self-driving car fields.

7.5 Conclusion

As a result of the project the dataset was collected, marked up and published and the algorithm was created, deployed and published. I've learned a few new things, like working with Docker or creating soft for robots. This project inspired me to dive into the world of reinforcement learning to be able to create not only computer vision algorithms but also algorithms like autonomous lane following. The

project was interesting and complicated, sometimes boring. But overall when Duckiebot finally drove by itself I was extremely happy.

7.6 Future work

Because of a lack of money for creating the polygon, I've decided to get rid of the red lines, which mark the beginning of the road intersection zone. So as future work adding a red line to the dataset can be considered. Also, these things can be done (and I'll probably do it, but not in the scope of this project):

1. Combining my algorithm with the object detection algorithm. This step will fully close the lane-following problem because Duckiebot will be able to drive in the traffic lane, but also be able to detect objects (like duckies, which imitate pedestrians).
2. Creating algorithm based on photogrammetry for crossing road intersection. This step in combination with the first one will fully solve the self-driving problem, as the bot will be able to drive on any type of city road: basic road and traffic intersections.
3. Improving lane following algorithm in the base image of Duckiebot.
4. Create a driver for the Duckiebot camera so that it receives images immediately in the PyTorch-GPU-tensor format, so as not to waste time converting the tensor from CPU-tensor format to GPU-tensor. This driver can greatly speed up the work of computer vision algorithms on bots.

References

[Avidan and Shamir, 2007] Avidan, S. and Shamir, A. (2007). Seam carving for content-aware image resizing. *SIGGRAPH*, 26.

[Chen et al., 2017] Chen, L.-C., Papandreou, G., Schroff, F., and Adam, H. (2017). Rethinking atrous convolution for semantic image segmentation.

[Dellaert, 2003] Dellaert, F. (2003). The expectation maximization algorithm.

[Dosovitskiy et al., 2021] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale.

[Guo et al., 2003] Guo, G., Wang, H., Bell, D., Bi, Y., and Greer, K. (2003). Knn model-based approach in classification. volume 2888, pages 986–996.

[Howard et al., 2019] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., and Adam, H. (2019). Searching for mobilenetv3.

[Howard et al., 2017] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications.

[Hu et al., 2019] Hu, J., Shen, L., Albanie, S., Sun, G., and Wu, E. (2019). Squeeze-and-excitation networks.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

[Kaur, 2024] Kaur, H. (2024). Why adam optimizer should not be the default learning algorithm. https://pub.towardsai.net/why-adam-optimizer-should-not-be-the-default-learning-algorithm-a2b8d019eaa0. Accessed: February 21, 2024.

[Kennedy, 2018] Kennedy, W. (2018). Scheduling in go : Part ii - go scheduler. https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html. Accessed: February 21, 2024.

[Kingma and Ba, 2017] Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA. Curran Associates Inc.

[Long et al., 2015] Long, J., Shelhamer, E., and Darrell, T. (2015). Fully convolutional networks for semantic segmentation.

[Ng et al., 2004] Ng, S., Krishnan, T., and Mclachlan, G. (2004). The em algorithm. *Handbook of Computational Statistics: Concepts and Methods*.

[Nikolenko, 2019] Nikolenko, S. I. (2019). Synthetic data for deep learning.

[O'Shea and Nash, 2015] O'Shea, K. and Nash, R. (2015). An introduction to convolutional neural networks.

[Peng et al., 2002] Peng, J., Lee, K., and Ingersoll, G. (2002). An introduction to logistic regression analysis and reporting. *Journal of Educational Research - J EDUC RES*, 96:3–14.

[Perez and Wang, 2017] Perez, L. and Wang, J. (2017). The effectiveness of data augmentation in image classification using deep learning.

[Podpora et al., 2014] Podpora, M., Korba´s, G., and Kawala-Janik, A. (2014). Yuv vs rgb – choosing a color space for human-machine interaction. *Annals of Computer Science and Information Systems*, Vol. 3.

[Prove, 2017] Prove, P.-L. (2017). An introduction to different types of convolutions in deep learning. https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d. Accessed: February 19, 2024.

[PyTorchContributors, 2023] PyTorchContributors (2023). Pytorch docs. https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html. Accessed: February 19, 2024.

[Raschka et al., 2020] Raschka, S., Patterson, J., and Nolet, C. (2020). Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4).

[Ruder, 2017] Ruder, S. (2017). An overview of gradient descent optimization algorithms.

[Sandler et al., 2019] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2019). Mobilenetv2: Inverted residuals and linear bottlenecks.

[Scabini and Bruno, 2021] Scabini, L. F. S. and Bruno, O. M. (2021). Structure and performance of fully connected neural networks: Emerging complex network properties.

[Sniffin, 2022] Sniffin, A. (2022). Applications of goroutines & channels in go. https://alexsniffin.medium.com/applications-of-goroutines-channels-in-go-2e24c478d71d. Accessed: February 21, 2024.

[Snyk, 2024] Snyk (2024). What is the mit license? top 10 questions answered. https://snyk.io/blog/mit-license-explained/. Accessed: February 21, 2024.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

[Sun et al., 2017] Sun, C., Shrivastava, A., Singh, S., and Gupta, A. (2017). Revisiting unreasonable effectiveness of data in deep learning era.

[Tan and Le, 2020] Tan, M. and Le, Q. V. (2020). Efficientnet: Rethinking model scaling for convolutional neural networks.

[Tomar and Laxkar, 2022] Tomar, A. and Laxkar, P. (2022). Differences of tanh, sigmoid and relu activation function in neural network. *International Journal of Scientific Progress and Research*, 80(6).

[Zhang et al., 2017] Zhang, X., Zhou, X., Lin, M., and Sun, J. (2017). Shufflenet: An extremely efficient convolutional neural network for mobile devices.

A Appendix 1

Computing Project Technical Plan

Name: Nechaev Ilia
Course: Computing, Software developing
Supervisor/Supervisory Team: Dr Panayiotis Andreou

Title

An effective low computation cost line detection algorithm for Duckiebot

Summary

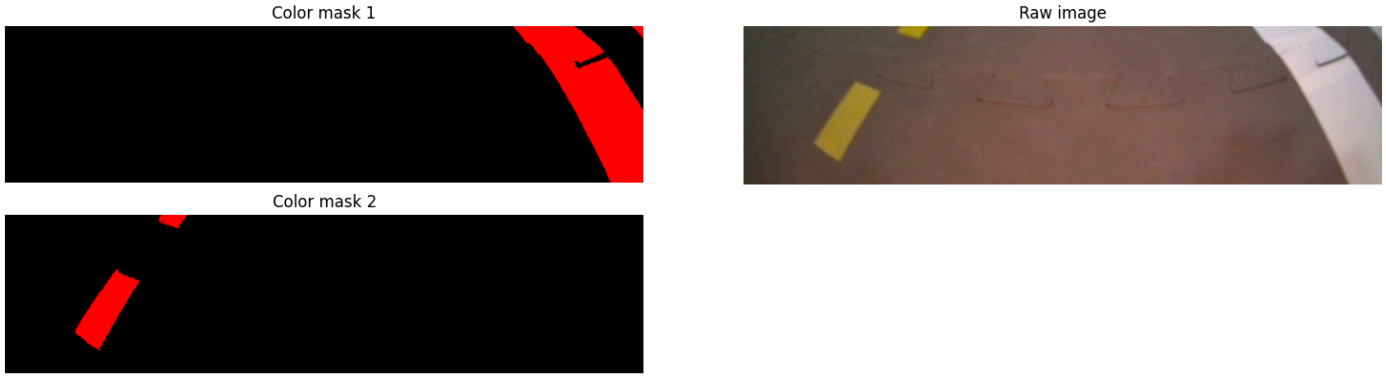
The project is to design a new line detection algorithm for Duckiebot, which will be light-insensitive and light-weighted in computation terms. The current solution used in Duckiebot is extremely unpredictable, it can change its behaviour even with little change of light: open or closed curtains with switched on in-door lights can significantly impact on current solution output.

My solution must meet two requirements: to be light-independent and to be light-weighted in terms of computation speed. To achieve this, I want to use machine learning techniques, but without deep learning models, because they are usually computational-heavy. Also I need to construct the dataset of marked up images of road marking to train my model..

So during this project I will:

1. Construct image dataset of road marking
2. Create (probably computational-heavy) reinforcement learning model for marking up the dataset
3. Create light-independent and light-weighted in terms of computation speed algorithm for detecting lines of road marking

Possible result of my algorithm:



Current solution result:

Deliverables

- Construct a dataset of marked up images of road marking.
- Design and Develop the algorithm of line detection
- Package with ROS (robotic operating system) node, which contains line detection algorithm
- Evaluation of the algorithm compared to others.
- Thesis report

Constraints

- Low-computational power of Duckiebot processor prohibits the use of computationally-heavy AI algorithms

Key Problems

- Constructing the dataset with appropriate line markers that will be used as input
- Inexperience developing applications in ROS, lack of proper documentation
- Low-computational power of Duckiebot processor

System and Work Outline

Methodology: Agile Iterative and Incremental development of the components
Prioritisation:

- Must:
  - Design, develop and deploy algorithm using reinforcement learning technique for marking up dataset
  - Design and develop light insensitive and light-weight algorithm for detecting road marking
- Should:
  - Deploy light insensitive and light-weight algorithm for detecting road marking on Duckeibot
- Could:
  - Crossroad crossing algorithm based on computer vision techniques
- Won't be:
  - Light-weight general-purposes line detection algorithm, which can be used for any type of road markup

MS1. Setup Infrastructure

1. Assembly of DuckieBot [1]
2. Creating test road networks

MS2. Construct Image Segmentation Dataset

1. Setup image repository
2. Develop software for DuckieBot to send images to repository [2]
3. Collect images from DuckieBot camera
4. Download images from external repositories
5. Research on appropriate techniques for markup lines [4][5][7][8]
6. Develop and algorithm to markup lines in collected images [4][5][7][8]

MS3. Design, Develop and Deploy Image Recognition Algorithm

1. Perform scientific literature review on marking algorithms and ML models relevant to the problem, such as the best model, choosing hyperparameters.
2. Design Develop and Deploy computationally heavy algorithm that will create masks using reinforcement learning techniques
3. Design Develop and Deploy light-weight algorithm that will detect lines on road [6][9][10][11][12][13]
4. Test and Evaluate the performance and accuracy of the algorithm on DuckieBot [2][3]
5. Perform optimization/tuning of the algorithm's parameters

MS4. Thesis Writeup

Technical stack:

1. Languages and libraries that will be used:
    a. Python with NumPy, MatPlotLib, SciPy, Pandas, PyTorch, Scikit-learn
    b. May be C++ or Rust for MS2.2 step
2. Other technologies:
    a. Docker
    b. Linux OS (Ubuntu 22.04 and Ubuntu 20.04)
    c. Git

Personal development:

1. Skills that will be developed:
    a. Deploying ML models
    b. Creating ML pipeline
    c. Creating and deploying EM algorithm
    d. Deepening knowledge in classical ML (machine learning)
    e. Deepening knowledge in RL (reinforcement learning)
    f. Deepening knowledge in DL (deep learning)
2. Skills that will be acquired:
    a. Developing applications for ROS
    b. Docker (creating and adjusting docker images)

IDEs:

1. Jupyter Notebooks (in Google Colab) to develop ML models
2. PyCharm by JetBrains for developing ROS packages

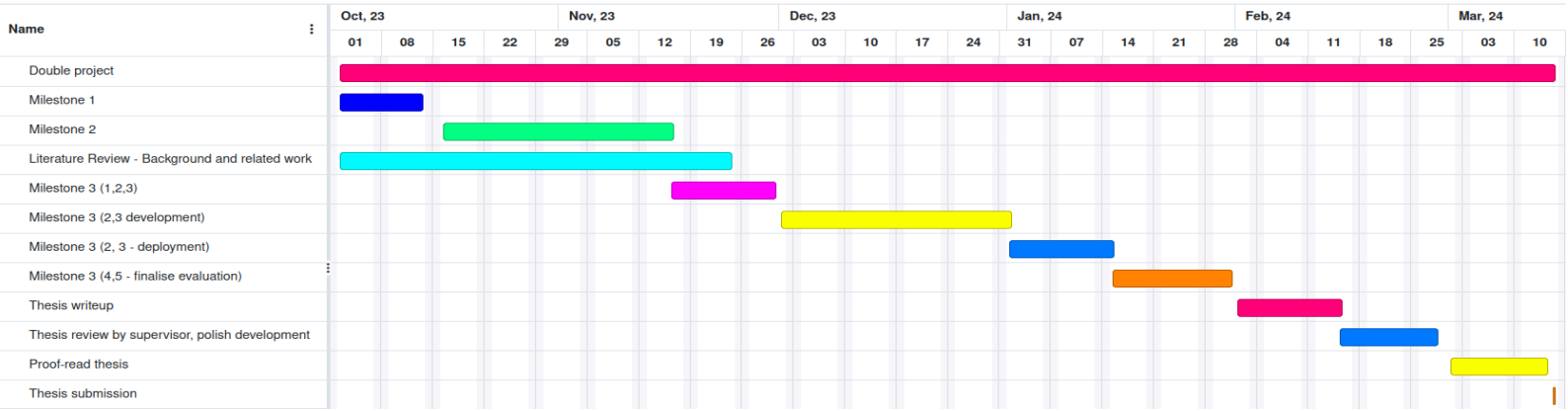
To evaluate resulting ML model I will probably use F1 score [14]:

$F1 = \frac{2 * Recall * Precision}{Recall + Precision}$, where

$Precision = \frac{True\ positive}{True\ positive + False\ positive}$ and

$Recall = \frac{True\ positive}{True\ positive + False\ Negative}$

Project Activities



1. 01.10-15.10 Milestone 1
2. 15.10-15.11 Milestone 2
3. 01.10-24.11 Literature Review - Background and related work
4. 16.11-30.11 Milestone 3. Steps: 1, 2, 3. Design part
5. 01.12-31.12 Milestone 3. Steps: 2, 3. Development part
6. 01.01-15.01 Milestone 3. Steps: 2, 3. Seployment part
7. 15.01-31.01 Milestone 3. Steps: 4, 5. Finalise evaluation
8. 01.02-15.02 Thesis writeup
9. 15.02-28.02 Thesis review by supervisor, polish development
10. 01.03-14.03 Proof-read thesis
11. 15.04 Thesis submission

Risk Analysis

Risk	Severity	Likelihood	Action
Lack of knowledge	Medium	High	Read more articles, books, listen to lectures
Duckiebot crash	High	Low	Fix it by myself or buy broken parts

Options

Other approaches for lane detection:

1. Extremely deep NN (not fit my bot due to lack of computational power)
2. Currently used non-NN approach. It's based on checking if the pixel fits the colour range (not fit my idea, because this approach is light sensitive)

Potential Ethical or Legal Issues

The project does not use individuals and the test road networks will not capture images of people.
There is no danger in using the Duckiebot for others, because it's small (12*20*10 cm) and light, so it can't deal any damage to people or their belongings.

Commercial Analysis

It is a research project. There is no aim to gain profit with it. But algorithm that I want to develop can be used on small bots that should be automated and which can be used in area with marking similar to road one (warehouses for example)

Factor name	Description	Is this a cost or a benefit	Estimated Amount	Estimate of when paid
Duckiebot	Buying the Duckiebot	Cost	~440$ + shipping	Already paid
Road	Rubber carpets and tape to create road	Cost	30€	Already paid
Google colab subscription	Subscription for using google remote server to fit my models	Cost	10$ per month * 9 month = 90 $	This amount is paid during academic year each month

Employability Contribution

Developing a line detection algorithm for the Duckiebot will help me to increase my knowledge in computer vision. Also during this project, I will use a lot of such libraries and frameworks on Python as PyTorch, Scikit-learn, Pandas, NumPy and MatPlotLib, which will increase my knowledge in data science (data preprocessing, machine learning).

References

[1] Duckietown Inc., *Assembly - Duckiebot DB21J,* Duckietown, viewed 1 October 2023
[2] Duckietown Inc., *Duckietown Developer Manual*, Duckietown, viewed 1 October 2023
[3] ROS, *Documentation, ROS, viewed 1 October 2023*
[4] Nikolenko S, Alexandrov T, Chernyavsky I, 2014 'Segmentation of MALDI imaging results based on graphical models'*, SPIIRAS Proceedings, vol.* 2, DOI:10.15622/sp.21.8
[5] Nikolenko S, 2019, *'Synthetic Data for Deep Learning',* Springer
[6] Golovanov S, Kurbanov R, Artamonov A, Davydow A, and Nikolenko S, 2018, 'Building Detection from Satellite Imagery Using a Composite Loss Function', *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Salt Lake City, UT, USA, pp. 219-2193, DOI: 10.1109/CVPRW.2018.00040.
[7] Yuzhikov V, 2012, *Restoring defocused and blurred images (translated from Russian), Habr, viewed 1 October 2023*
[8] Avidan S and Shamir A, 2007, 'Seam carving for content-aware image resizing', A*CM SIGGRAPH papers,* DOI:10.1145/1275808.1276390
[9] Nikolenko S, Kadurin A, Arkhangelskaya E, 2017, '*Deep Learning',* Piter
[10] Goodfellow I, Bengio Y and Courville A, 2016, '*Deep Learning*' MIT Press

[11] Brownlee J, 2019, '*Deep Learning for Computer Vision: Image Classification, Object Detection, and Face Recognition in Python*', Independently Published
[12] Andrew G, Zhu H, Zhu M, Chen B, Kalenichenko D, Wang W, Weyand T, Andreetto M and Adam H, 2017, 'MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications', Computing Research Repository
[13] Zhang X, Zhou X, Lin M and Jian Sun, 2018, 'ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices', IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 2018 pp, DOI:10.1109/CVPR.2018.00716
[14] Thomas Wood, *F-Score,* DeepAI, viewed 6 October 2023