UNIVERSITY OF U ZAGREBU

FACULTY OF ORGANIZATION AND INFORMATICS

VARAŽDIN

Rene Maruševec

AUTONOMOUS VEHICLES AS A MULTI-AGENT SYSTEM

MASTER'S THESIS

Varaždin, 2020

UNIVERSITY OF ZAGREBU

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

Rene Maruševec

Student ID number: 0016116768

Study programme: Databases and Knowledge Bases

AUTONOMOUS VEHICLES AS A MULTI-AGENT SYSTEM

MASTER'S THESIS

Mentor:

Assoc. Prof. Dr. Sc. Markus Schatten

Varaždin, September 2020

Rene Maruševec

Statement of Authenticity

I hereby state that my Master's thesis is an original result of my work and that I did not use any sources other than those stated in it through the process of writing it. Through the creation of this work ethically appropriate and acceptable methods and techniques of work have been used.

The author has confirmed this by accepting the regulation in the FOI radovi system

---

Abstract

This thesis is focused on perception in autonomous driving. Specifically, problems such as Lane Lines detection, Traffic Sign Classification, and Object detection are introduced. The thesis covers implementation of steps for detecting the lane lines. After that, some advanced methods for lane line detection are listed. In the next chapter deep learning methods are covered in theory and model for Traffic Sign Classification problem is trained and implemented as Docker service. In the last chapter of this thesis, an overview of several Object Detection algorithms is given.

Keywords: autonomous vehicles; self-driving cars; artificial intelligence; deep learning; lane lines detection; object detection; traffic sign classifier; convolutional neural networks; multi-agent systems; python; TensorFlow;

Table of context

1. Introduction

Autonomous vehicles (AV) will be the next big revolution that will change our lives forever, as the invention of the wheel, motor, and later the car did in the past. AV's are possible now because computational power is drastically increased in the last decade and the result of that is the use of Artificial Intelligence (AI) and specifically Deep Learning (DL) as key technologies for Self-driving cars. There are several addressed benefits that AV's will bring according to [1]. First and most important, they will make our roads, this is important because cars are the most dangerous means of transport. Autonomous vehicles will improve parking in congested cities be-cause driver-less vehicles could simply drop passengers off and move on. AV's will make the daily commute more bearable and productive because the robot will take driving task from us so we can focus on other tasks. With the majority of self-driving vehicles being electric, and given the fact that autonomous vehicles drive more efficiently than humans, emissions will be reduced drastically. This is not the end of the list of benefits, and one thing is sure, AV's will make our lives better in many ways.

Localization and Mapping, Scene Understanding, Movement Planning, Human-Robot Interaction, and Communication are the main topics in this field. These problems are solved in different ways and different approaches were taken because there are many companies out there in AV game [2]. Some big names are Google's sister company Waymo, Tesla, Apple, Cruise, and others. Tesla [3] have different approach than others and their networks learn from the most complicated and diverse scenarios in the world, iteratively sourced from their fleet of nearly 1 million vehicles in real time. A full build of Autopilot neural networks involves 48 networks that take 70,000 GPU hours to train. Together, they output 1,000 distinct tensors (predictions) at each timestep. Other companies are focused on Lidar based solutions.

In this thesis focus is on scene understanding in autonomous driving, actually perception problems are the main topic. Specifically, Lane Lines detection, Traffic Sign Classification, and Object detection problems are introduced. The main motivation for this topic is because perception problems and prediction of actions are the hardest problems in AV and in this topic, there is a lot of space for improvements. In the first chapter of this paper, an introduction to Autonomous vehicles is given and benefits are presented with listed companies that are working in the field. In the second chapter, the Lane Lines Detection problem is covered in detail. Implementation of lane lines detection algorithm is explained with all concept used, and at the end of the chapter concepts for advanced lane lines detection are listed as a part of the future work section. In the next chapter Traffic Sign Classifier is presented. In the first part, related work is listed and after that important theoretical concepts for understanding Convolutional Neural Networks are listed and later explained in detail. After theory used Python libraries are introduced and at the end whole implementation of the Traffic Sign Classifier is explained from loading the dataset and creating the model to training and testing. In the last section, Traffic Sign Classifier is implemented as a Docker service. The last chapter of this paper is a little bit different from the previous two. In the last chapter Object detection problem is presented but without implementation. In this chapter, several Object detection approaches will be introduced and compared.

2. Autonomous driving

In order to limit the number of accidents due to driver's faults and to improve driving comfort, many applications have been developed during the last decades. One of them can be considered as the next major evolution for the coming years: the development of an autonomous or partially autonomous vehicle. Accordingly to [4] mission of self-driving cars are to improve access, increase efficiency, and save lives. This mission started not so long ago in 2005 on DARPA Grand Challenge where Stanford University's self-driving car Stanley won this desert race.

2.1. Benefits from autonomous driving

Today there are more than 40 companies [5] which are actively investing in autonomous vehicle technology. It is important to mention some of the most influential traditional car manufacturers like Hyundai, Audi, Ford, GM, Volvo, Jaguar, and others. A very important car manufacturer of the new generation is Tesla plus the odd tech heavyweight like Waymo parent company Google, Amazon, Apple, Nvidia, Microsoft, Baidu, and Bosch. Along with them, there are also autonomous driving startups Aptiv, Cruise, Uber, Lyft and a lot of small startups specialized in small areas like 3D maps and LiDAR technologies which are listed in [2]. On Figure 1 overview of companies in AV Applications are presented. As a result, it is likely to see the first

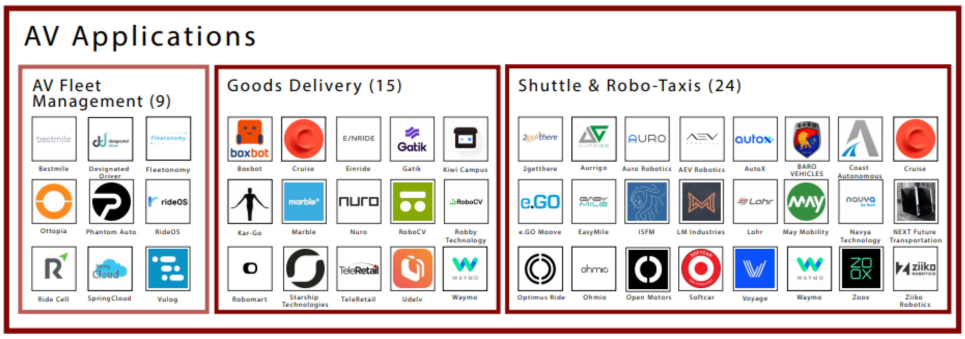


Figure 1: Companies working on Autonomous Vehicles (AV) Applications [2]

truly autonomous, commercially viable autonomous cars hit the market in the coming years. There are five ways how Self-Driving Cars could make out World better.

Some benefits are given in [1] in following order:

- **Making our roads safer.** Statistics, from the National Highway Traffic Safety Administration (NHTSA) show that driver error is by far the biggest cause of road traffic accidents, due to factors like miscalculations, errors of judgment, speeding, drink-driving, and phone use. In fact, an alarming 94 percent [6] of serious crashes are down to human error. Fully autonomous vehicles would take human error out of the equation, thereby making our roads safer not just for drivers, but also passengers, cyclists, and pedestrians. For that reason, the NHTSA has championed the safety benefits of autonomous vehicles.

- **Easing parking woes.** Autonomous vehicles will improve parking in congested cities because driver-less vehicles could simply drop passengers off and move on. This becomes increasingly realistic if the motorists of the future choose not to purchase their own self-driving vehicle, but instead make use of automated ride-sharing or taxi services. As an added bonus, with fewer people needing to find a parking space, land that is currently used for huge car parks could be repurposed for much-needed housing or attractive public spaces. With this in mind, the city of Chandler in Arizona has already changed its zoning laws to accommodate autonomous vehicles; developers will now be able to build properties with fewer parking spaces, so long as they provide suitable curb-side passenger loading zones. In other words, the buildings of the future may no longer be flanked by huge, ugly car parks, and will instead favor dedicated areas for drop-offs and pick-ups.

- **Making the daily commute more bearable and productive.** When we reach the point where human intervention behind the wheel is no longer needed, autonomous vehicles will drastically improve the daily commute. Instead of sitting behind the wheel, people will be able to stretch out in the back, get ahead on some work, or simply relax and catch up on your latest Netflix obsession. Statistically Americans spend 19 full working days a year stuck in traffic on their commute, that is a lot of time commuters will be able to claw back for themselves.

- **Reducing emissions.** In the future, with the majority of self-driving vehicles being electric or hybrids, and given the fact that autonomous vehicles drive more efficiently than humans, emissions will be reduced drastically. That's the hope, anyway. One study by the University of California, Davis suggests that automated vehicles could reduce greenhouse gas emissions by as much as 80 percent by 2050 – but only if the vehicles are electric and shared. The transition towards more shared vehicles is important, because if autonomous vehicles are not shared then the worrying prediction is emissions could actually increase (by as much as 50 percent by 2050, according to the same study). This is because self-driving cars could make car ownership possible for people who cannot drive, potentially increasing the number of vehicles and miles driven. Therefore, this benefit is far from a foregone conclusion.

- **Cutting domestic air travel.** Volvo's 360c concept car [7], which was unveiled in 2018, provides a glimpse of a future in which self-driving cars are no longer just cars – they're places to work, sleep, and relax. Almost a cross between a hotel room and the first-class cabin of a plane. Volvo's vision is one of a luxurious, driver-less future in which a car can pick you up and take you wherever you need to go, potentially eliminating the need for short-haul, domestic air travel. With Volvo's concept, you would be able to order a car, pre-order food and drink, and then recline and relax in the back while the car transports you to your destination. You then arrive fresh-faced and relaxed, having endured none of the stress and exhaustion of plane travel. Particularly for journeys that could be made overnight, this vision could seriously disrupt the travel industry – and cut the carbon footprint associated with travel.

2.2. Levels of automation

There are 5 levels of driving automation presented in Figure 2. Google's sister company Waymo drove 20 million miles with its Level 4 vehicle until 2020. While Tesla with Level 3 vehicles is near the checkpoint of 2 Billion Miles driven [4].

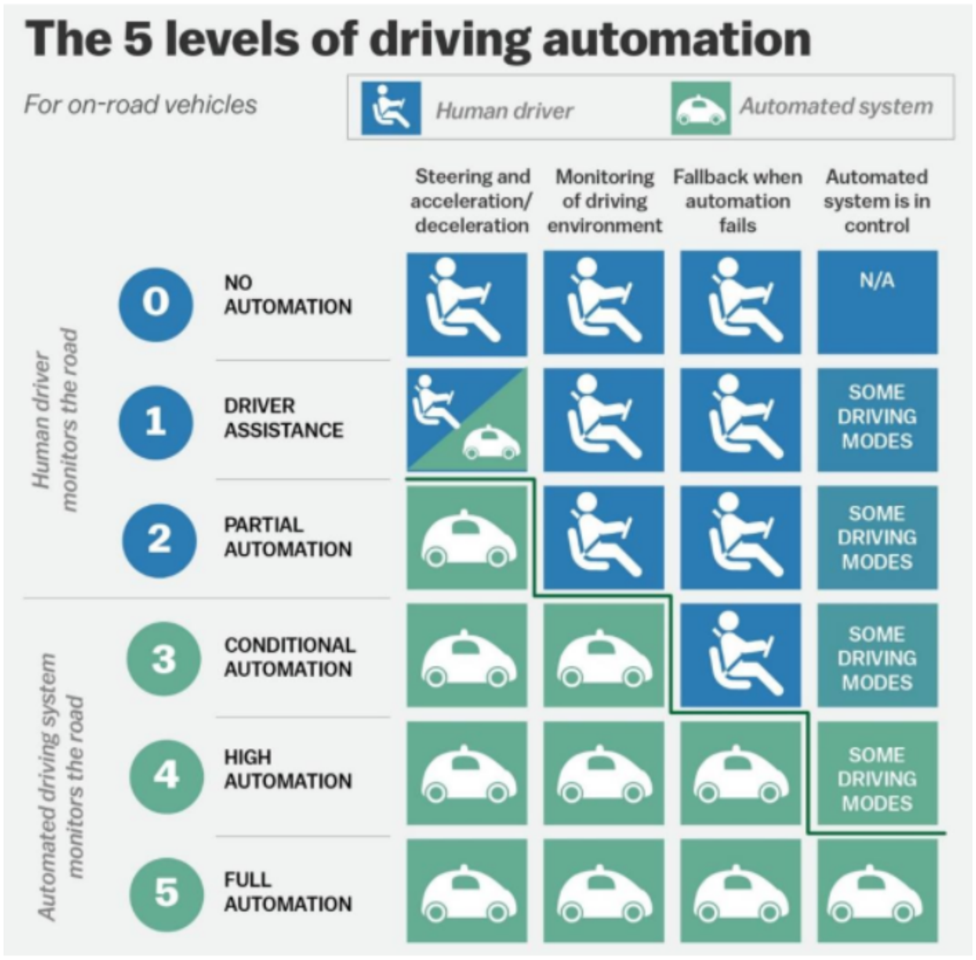

The 5 levels of driving automation
For on-road vehicles
Human driver
Automated system
Steering and acceleration/ deceleration
Monitoring of driving environment
Fallback when automation fails
Automated system is in control
Human driver monitors the road
Automated driving system monitors the road
0 NO AUTOMATION
1 DRIVER ASSISTANCE
2 PARTIAL AUTOMATION
3 CONDITIONAL AUTOMATION
4 HIGH AUTOMATION
5 FULL AUTOMATION
N/A
SOME DRIVING MODES

Figure 2: The 5 levels of driving automation [4]

In [8] they detected these autonomous challenges:

- Localization and Mapping (Where am I?);
- Scene Understanding (Where/who/what/why of everyone else?);
- Movement Planning (How do I get form A to B);
- Human-Robot Interaction (What is the physical and mental state of the driver?);
- Communication (How do I convey intent to the driver and to the world?);

2.3. Sensors

There are two approaches to these challenges presented in [4]:

- Vision Sensors and Deep Learning
  - Pros:
    * Highest resolution information
    * Feasible to collect data and scale and learn
    * Roads are designed for human eyes
    * Cheap
  - Cons:
    * Not accurate without a lot of data
    * Not explainable, not consistent
- Lidar and Maps.
  - Pros:
    * Explainable, consistent
    * Accurate
  - Cons:
    * Dose not improve over time
    * Lidar’s are expensive

In the next chapter, the Scene Understanding problem, specifically the lane detection problem is explained and implemented using a Vision approach but without deep learning.

2.4. Testing polygons

Testing is not a one and done task, but rather a never ending feedback loop that includes structured testing, simulation, and public road operations. Because of that Waymo [9] created their own testing polygon in the middle of Merced County in California, you’ll find it at Castle, a former Air Force Base that is shown in Figure 3. They created tiny city where they control everything that happens on the streets. They manage how many cars zip down the roads and how fast they are going. They dictate how many cyclists are on a roundabout or whether they follow the road rules. The “weather” around the vehicle can change multiple times a day from blue skies and sunshine one minute to heavy rain showers the next, but only if they want to. When they develop a scenario to test new software, they can use either one of these tools or a combination of them. For example, after executing a test on our private track, they can then create and run hundreds of variations of that scenario in simulation. Our simulation technology allows us to do this in a matter of seconds right at our developers’ desks as they work on new features for the Waymo Driver.

Figure 3: Waymo testing polygon

In Europe the biggest test track is located in Hungary [10]. The Zalaegerszeg test track is unique, the traditional test track features focusing on driving and driving stability are implemented together with the research and development infrastructure elements connection with future vehicles on multi-level system for validation. The proving ground provides not only dynamics tests for conventional vehicle, but it also allows validation tests for autonomous vehicles and electric vehicles.

3. Lane Lines Detection

In Autonomous driving, lane detection and tracking are very crucial treatments to locate the vehicle and to track its position on the road. In this chapter firstly some related articles for lane detection are presented. Later in this chapter, the implementation of a basic lane detection algorithm is explained in detail and at the end possibilities for improvement are listed.

3.1. Related work

Paper [11] proposes an unconventional approach for multi-lane detection and tracking based on a reactive multi-agent system. Given method is based on a confidence map instead of lane marking features, and a multi-agent model instead of the geometric fitting. This approach takes better account of the specific features of road markings, and more precisely, parts defined by clothoid.
[12] present a robust lane-detection-and-tracking algorithm to deal with challenges scenarios such as a lane curvature, worn lane markings, lane changes, and emerging, ending, merging, and splitting lanes. A comparative study to find a good real-time lane marking classifier is presented first. Once the detection is done, the lane markings are grouped into lane-boundary hypotheses. Left and right lane boundaries are grouped separately to effectively handle merging and splitting lanes. A fast and robust algorithm, based on random-sample consensus and particle filtering are proposed to generate a large number of hypotheses in real-time. The generated hypotheses are evaluated and grouped based on a probabilistic framework. The suggested framework effectively combines a likelihood-based object-recognition algorithm with a Markov-style process (tracking) and can also be applied to general-part-based object-tracking problems.
In Part 4 of the DeepPiCar article [13] Open-CV is used to detect color, edges, and line segments. Then compute steering angles, so that PiCar can navigate itself within a lane. This project is different from the others because here is used remote control car but principles for lane detection are more or less the same as in [14]. The good thing about this article is that the deep learning part of lane detection is described in Part 5 and Part 6. Our approach is similar to these two articles so in the next chapter our approach will be presented.

3.2. Used Python Libraries

In this section used Python Libraries are introduced.

3.2.1. OpenCV

OpenCV (Open Source Computer Vision Library) [15] is an open-source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc. OpenCV has more than 47 thousand people of the user community and an estimated number of downloads exceeding 18 million. The library is used extensively in companies, research groups, and governmental bodies.

3.2.2. Numpy

Numpy [16] is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors.

3.2.3. Sys

System-specific parameters and functions [17] module provide access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

3.3. Lane Lines detection in practice

Lane detection is a basic Computer Vision problem so we will take video as input and try to process it to detect lane lines on it and as a result, we will display detected lanes on video. To do this we will use OpenCV and NumPy. This task is also a part of Udacity's course Self-Driving Car Engineer so videos for testing are from [18]. In Figure 4 we can see one frame from test video. The basic idea is to take each frame from the video and detect lanes on it.

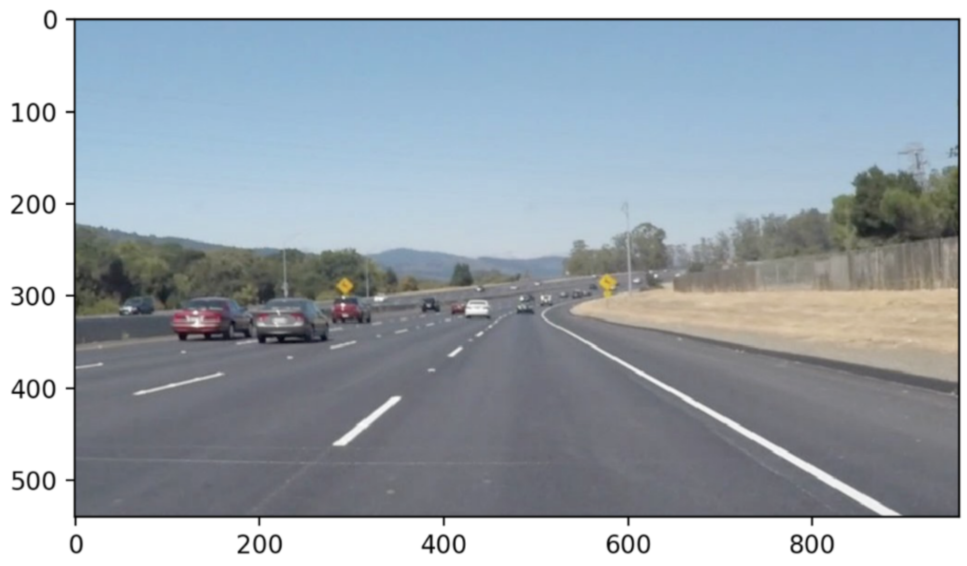


Figure 4: Original frame from video

To do this we used this code. Now we can access each frame using variable 'frame'.

```
import cv2

video = cv2.VideoCapture("solidWhiteRight.mp4")
while(video.isOpened()):
    _, frame = video.read()
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
capture.release()
cv2.destroyAllWindows()
```

3.3.1. Canny edge detection

Next step is to detect the edges, but how to do this? The image is a matrix of color data for each pixel. To detect edges we can check where to color values change very quickly and Canny edge detection does exactly that. Canny Edge Detection essentially detects areas of the image that have a strong gradient in the image's color function. Canny also adds a pair of intensity threshold parameters which indicate generally how strong an edge must be to be detected [14].
To make this task simpler we will convert the image to grayscale as you can see in Figure 5. And now we just need to apply canny edge detection on this image to get something like Figure 6.

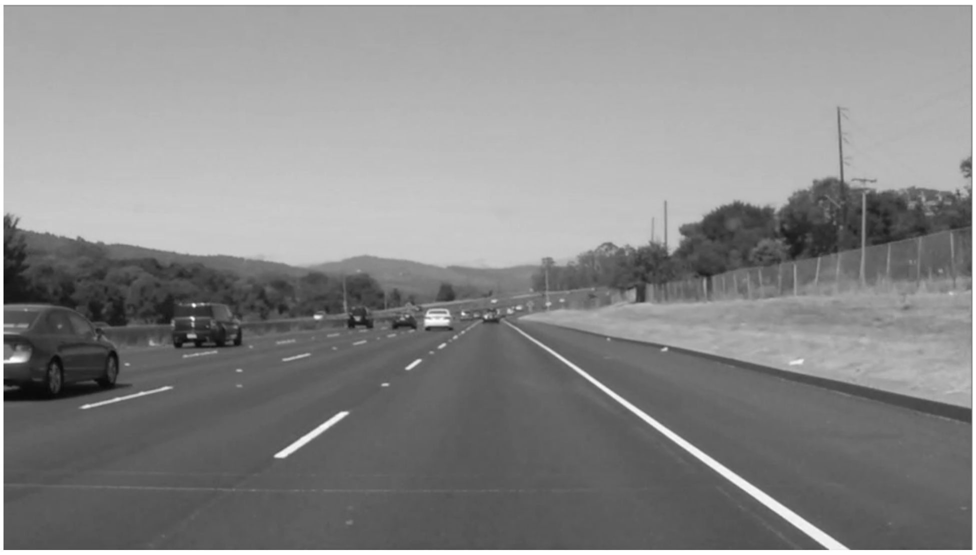


Figure 5: Frame converted to gray scale

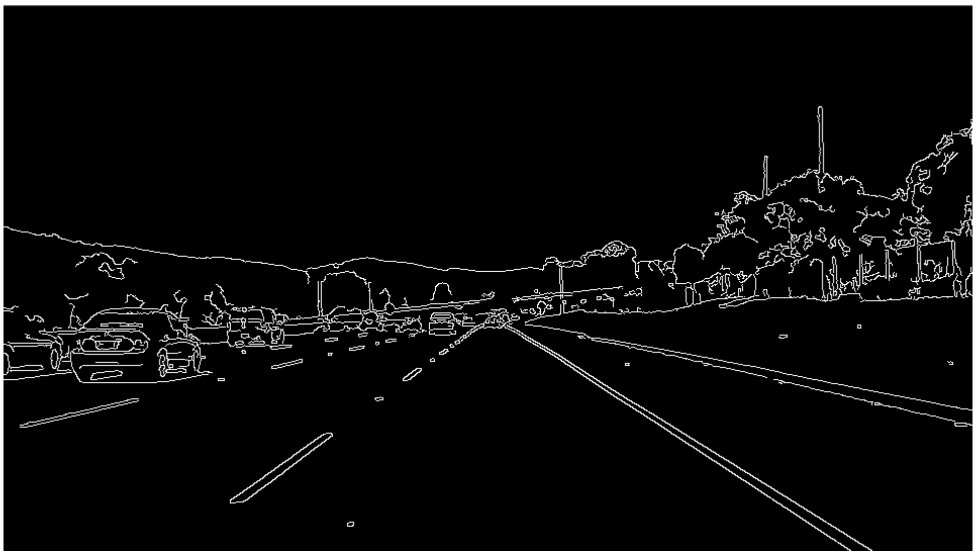


Figure 6: Canny edge detection

For this step, we only need those lines of code. That looks an easy task but here is one problem. To detect useful edges we need to make sure that we pick the right parameters. In this article [19] there are some tips and good practices for parameter tuning. To sum it up the only way is that we need to try a lot of variants and pick what works best in our situation.

```
gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
kernel = 5
blur = cv2.GaussianBlur(gray, (kernel, kernel), 0)

low_threshold = 50
high_threshold = 150
canny = cv2.Canny(gray, low_threshold, high_threshold)
```

3.3.2. Region of interest

With canny we will get a lot of detected edges but how to know which are form lanes? We will use some facts to make it easier. In our case lanes will always be vertical and in the middle of images. So, we will choose our region of interest with a triangle that begins at the bottom left corner of the image, proceeds to the center of the image at the horizon, and then follows another edge to the bottom right corner of the image. Figure 7 shows our cropped image and you can see that now we have only edges of lanes in our image. Here is also important to take the right parameters for the triangle. The following code is used to crop the image.

```
height = canny.shape[0]
width = canny.shape[1]
mask = np.zeros_like(canny)
region = np.array([[
    (0, height),
    (width / 2, height / 2),
    (width, height), ]], np.int32)

match_mask_color = 255
cv2.fillPoly(mask, region, match_mask_color)
cropped_img = cv2.bitwise_and(canny, mask)
```

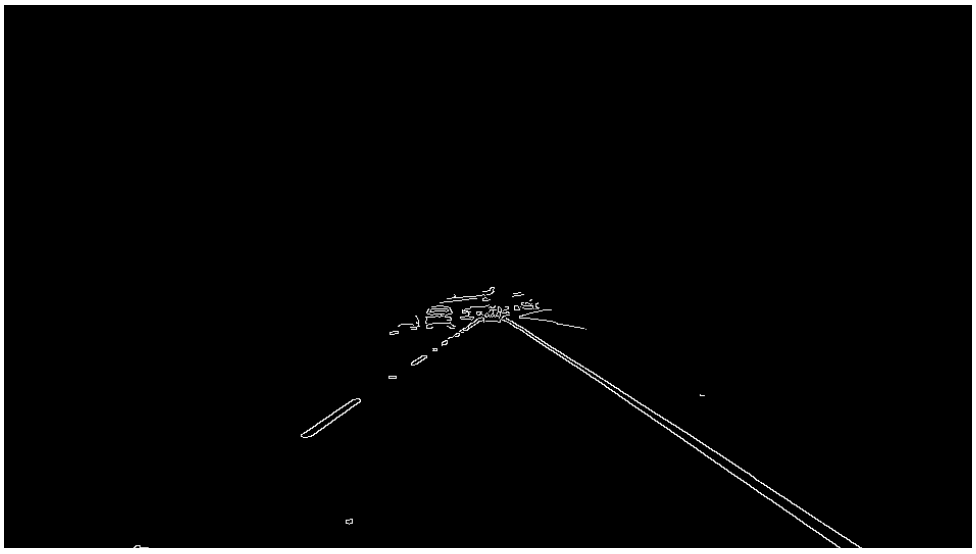


Figure 7: Cropped image with the region of interest

3.3.3. Detecting lanes with Hough Transforms

The main concept we will be using here is called a Hough Transform. Using a Hough Transform, we will transform all of our edge pixels into a different mathematical form. Once the transformation is complete, each edge pixel in “Image Space” will have become a line or curve in “Hough Space”. In Hough Space, each line represents a point from Image Space, and each point represents a line from Image Space [14]. To the point, we now do not need to solve for a line that intersects all nearby edge pixels. Instead, we can simply solve for the intersections between lines in Hough Space, and transform that intersection point back into Image Space to obtain a line which intersects enough edge pixels. The inputs to the Hough Transform can be varied to alter which lines are considered a real feature of the scene and which are just clutter. The following algorithm will generate a listing of all the lines which are part of the scene.

```
    lines = cv2.HoughLinesP(
        cropped_img,
        rho=2,
        theta=np.pi / 180,
        threshold=115,
        lines=np.array([]),
        minLineLength=30,
        maxLineGap=25
    )
```

The last thing we will do with line detection is to render the detected lines back onto the image. As a result, we will get Figure 8.

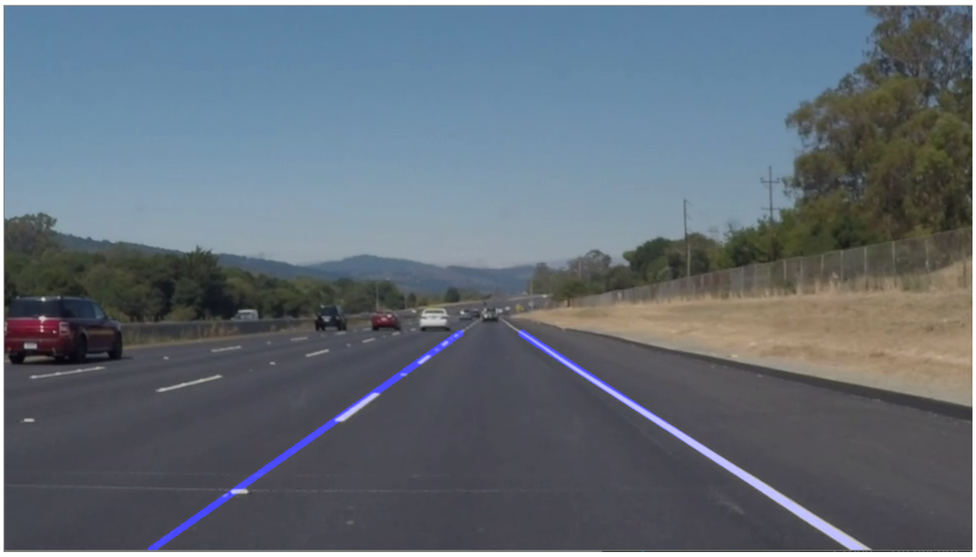


Figure 8: Detected lines

3.3.4. Docker service

The whole lane lines detection algorithm is created as a Docker service that accepts one parameter which is the path to the video file. In the code below the main function is showed in which all methods for lane detection are called.

```
def main(videoName):
    video = cv2.VideoCapture(videoName)
    while(video.isOpened()):
        _, frame = video.read()
        lines = finding_lanes(frame)
        detected_lines = draw_lines(frame, lines)
        combo_image = cv2.addWeighted(frame, 0.8, detected_lines, 1, 1)
        cv2.imshow("result", combo_image)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    capture.release()
    cv2.destroyAllWindows()


if __name__ == "__main__":
    videoName = sys.argv[1]
    main(videoName)
```

3.4. Advanced Lane Lines detection

There have been many approaches proposed for lane detection, and most of them are based on three main approaches. First, road marking features are extracted from images by using a gradient detector or a dark-light-dark transition detector. Secondly, an estimation of the geometrical model is performed by studying the extracted point distribution with different kinds of models like line model, polynomial model, spline model. Thirdly, a deep learning approach with a trained neural network that will detect lanes.
In this chapter, the first simple approach was presented, but future work will be based on the other two approaches. That will lead to fun new challenges such as camera calibration, image distortion, warping, color spaces, gradients, lane curvature, and convolutional neural networks.

4. Traffic Sign Calssifier

Traffic sign recognition is an example of a multiple class recognition problem. In this chapter related work in Traffic Sign Classification is listed and after that important theoretical concepts are presented. Later in this chapter recognition of traffic signs based on convolutional neural networks (CNNs) is presented. A CNN architecture called LeNet-5 [20] was successfully trained and tested first on handwritten digits MNIST dataset [21] and later on German Traffic Sign Benchmark [22].

4.1. Related Work

Paper Convolutional Neural Networks for Croatian Traffic Signs Recognition [23] proposes CNNs approach to this problem by learning meaningful representations directly from the data, so the learned representations are optimal for the specific classification problem, thus eliminating the need for hand-designed image features. LeNet-5 was successfully trained for handwritten digits recognition and tested on the MNIST dataset yielding state-of-art results at the time. An improved and larger CNN was later developed and current state-of-the-art results on the GTSRB dataset [22] were obtained.

In the paper [24] the approach that won the final phase of the German traffic sign recognition benchmark was described. This method is the only one that achieved a better than-human recognition rate of 99.46%. They used fast, fully parameterizable GPU implementation of a Deep Neural Network (DNN) that does not require careful design of pre-wired feature extractors, which are rather learned in a supervised way. Combining various DNNs trained on differently preprocessed data into a Multi-Column DNN (MCDNN) further boosts recognition performance, making the system insensitive also to variations in contrast and illumination.

Udacity implemented basic LeNet-5 for classification MNIST dataset with handwritten digits is given in [25]. This Convolutional Neural Networks with MNIST dataset is the current state-of-art architecture for the image classification tasks.

Stanford University created an excellent online course CS231n: Convolutional Neural Networks for Visual Recognition available on [26]. In this course, all theoretical concepts are explained very well. In the next section, all important concepts are presented.

4.2. Convolutional Neural Networks in theory

Artificial Neural Networks (ANN) [27] is a supervised learning system built of a large number of simple elements, called neurons or perceptrons. Each neuron can make simple decisions and feeds those decisions to other neurons, organized in interconnected layers. Together, the neural network can emulate almost any function, and answer practically any question, given enough training samples and computing power. A “shallow” neural network has only three layers of neurons:

- **Input layer** that accepts the independent variables or inputs of the model
- **One Hidden layer**
- **Output layer** which generates predictions

A Deep Neural Network (DNN) [27] has a similar structure, but it has two or more "hidden layers" of neurons that process inputs Figure 9. Shallow neural networks are able to tackle complex problems, but deep learning networks are more accurate and improve in accuracy as more neuron layers are added. Additional layers are useful up to a limit of 9-10, after which their predictive power starts to decline. Today most neural network models and use a deep network of between 3-10 neuron layers.

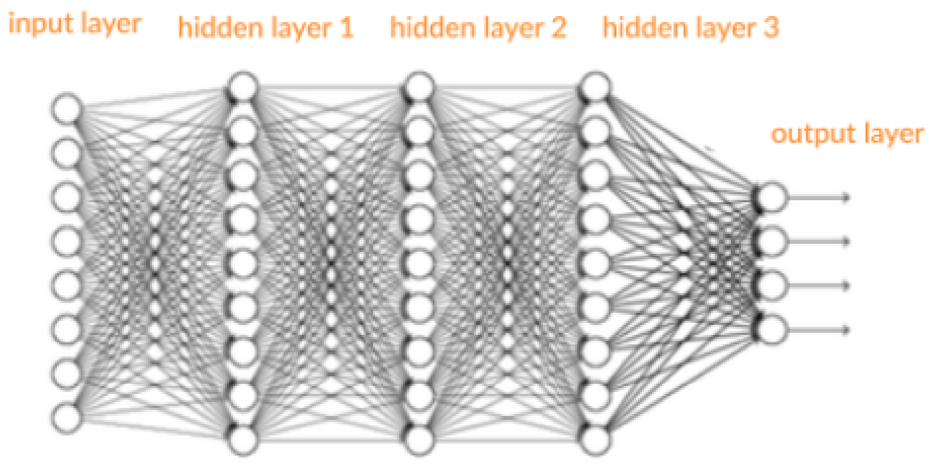


Figure 9: Deep neural network [27]

4.2.1. Neural Networks basic concepts

Here is a glossary of basic terms that are important to know for neural networks:

- **Inputs** - Source data fed into the neural network, with the goal of making a decision or prediction about the data. Inputs to a neural network are typically a set of real values; each value is fed into one of the neurons in the input layer [27].
- **Traning Set** - A set of inputs for which the correct outputs are known, used to train the neural network [27].
- **Validation Set** - A group of unseen examples which is used to test neural network to see how it performs [27].
- **Test Set** - A group of unseen examples which is used to test neural network only at the end when all parameters are finally tuned [27].

- **Outputs** - Neural networks generate their predictions in the form of a set of real values or boolean decisions. Each output value is generated by one of the neurons in the output layer [27].

- **Neuron/perceptron** - Just like a neuron 10 forms the basic element of our brain, a neuron forms the basic structure of a neural network. Just think of what we do when we get new information. When we get the information, we process it, and then we generate an output. Similarly, in case of a neural network, a neuron receives an input, processes it, and generates an output that is either sent to other neurons for further processing or it is the final output [28].

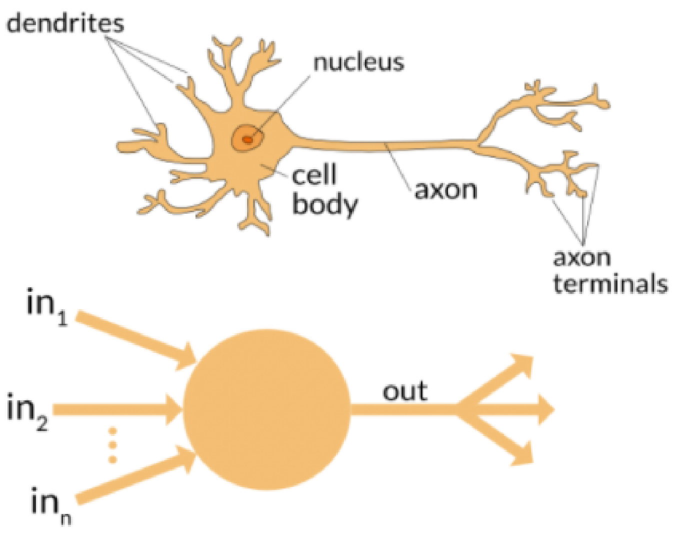


Figure 10: Biological Neuron and Artificial Neuron [28]

- **Weight Space** - Each neuron is given a numeric weight. The weights, together with the activation function, define each neuron's output. Neural networks are trained by fine-tuning weights, to discover the optimal set of weights that generates the most accurate prediction.

- **Bias and Variance** - When training neural networks, like in other machine learning techniques, a balance between bias and variance is very important. Bias measures how well the model fits the training set—able to correctly predict the known outputs of the training examples. Variance measures how well the model works with unknown inputs that were not available during training. Another meaning of bias is a “bias neuron” which is used in every layer of the neural network. The bias neuron holds the number 1 and makes it possible to move the activation function up, down, left, and right on the number graph [27].

- **Activation Function** - Each neuron accepts part of the input and passes it through the activation function. Common activation functions are sigmoid, TanH, and ReLu. Activation functions help generate output values within an acceptable range, and their non-linear form is crucial for training the network [27].

- **Forward Pass** - The forward pass takes the inputs, passes them through the network, and allows each neuron to react to a fraction of the input. Neurons generate their outputs and pass them on to the next layer until eventually, the network generates an output [27].
- **Error Function** - Defines how far the actual output of the current model is from the correct output. When training the model, the objective is to minimize the error function and bring output as close as possible to the correct value [27].
- **Cost Function** - When we build a network, the network tries to predict the output as close as possible to the actual value. We measure this accuracy of the network using the cost/loss function. The cost or loss function tries to penalize the network when it makes errors.

  Our objective while running the network is to increase our prediction accuracy and to reduce the error, hence minimizing the cost function. The most optimized output is the one with the least value of the cost or loss function.

  If I define the cost function to be the mean squared error, it can be written as –

  C= 1/m (y – a)2 where m is the number of training inputs, a is the predicted value and y is the actual value of that particular example.

  The learning process revolves around minimizing the cost [28].
- **Gradient Descent** - Gradient descent is an optimization algorithm for minimizing the cost. To think of it intuitively, while climbing down a hill you should take small steps and walk down instead of just jumping down at once. Therefore, what we do is, if we start from a point where the mouse is pointed in Figure 11, we move down a little i.e. Loss and update our position to smaller Loss and we keep doing the same till we reach the bottom. Consider bottom to be the minimum cost point [28].

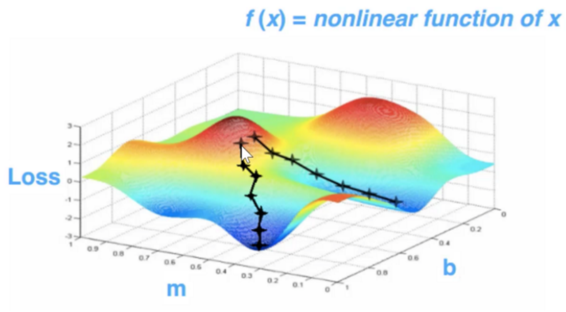


Figure 11: Gradient Descent [29]

- **Learning Rate** - The learning rate is defined as the amount of minimization in the cost function in each iteration. In simple terms, the rate at which we descend towards the minimum of the cost function is the learning rate. We should choose the learning rate very carefully since it should neither be very large (On Figure 12 Big learning rate) that the optimal solution is missed and nor should be very low that it takes forever for the network to converge [28].

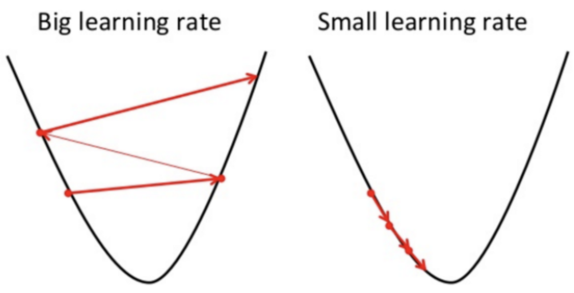


Figure 12: Learning Rate [29]

- **Batches** - While training a neural network, instead of sending the entire input in one go, we divide in input into several chunks of equal size randomly. Training the data on batches makes the model more generalized as compared to the model built when the entire data set is fed to the network in one go [28].
- **Backpropagation** - In order to discover the optimal weights for the neurons, we perform a backward pass, moving back from the network's prediction to the neurons that generated that prediction. This is called backpropagation. Backpropagation tracks the derivatives of the activation functions in each successive neuron, to find weights that brings the loss function to a minimum, which will generate the best prediction. This is a mathematical process called gradient descent [27].
- **Hyperparameters** - A hyperparameter is a setting that affects the structure or operation of the neural network. In real deep learning projects, tuning hyperparameters is the primary way to build a network that provides accurate predictions for a certain problem. Common hyperparameters include the number of hidden layers, the activation function, and how many times (epochs) training should be repeated [27].
- **Epochs** - An epoch is defined as a single training iteration of all batches in both forward and backpropagation. This means 1 epoch is a single forward and backward pass of the entire input data.

  The number of epochs you would use to train your network can be chosen by you. It's highly likely that more number of epochs would show higher accuracy of the network, however, it would also take longer for the network to converge. Also, you must take care that if the number of epochs is too high, the network might be over-fit [28].
- **Dropout** - A typical approach in machine learning when improving generalization consists of combining different architectures. However, that can be computationally quite expensive. The dropout method suggests randomly disabling some hidden units during training, thus generating a large set of combined virtual classifiers without the computational overhead [15]. For a simple multilayer perceptron with N neurons in one hidden layer, 2 N virtual architectures would be generated when applying dropout. Usually, half the units are disabled (Figure 13) in each learning iteration[23].

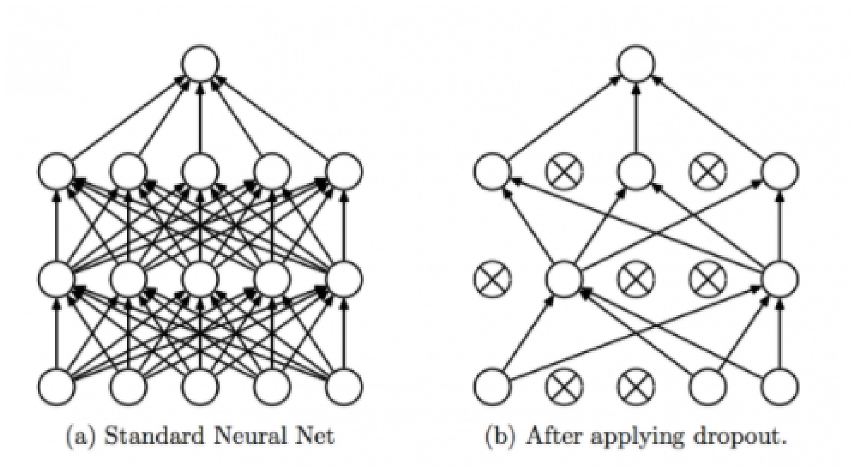


Figure 13: Dropout [28]

4.2.2. Multilayer Perceptron

A perceptron is a binary classification algorithm modeled after the functioning of the human brain—it was intended to emulate the neuron. The perceptron, while it has a simple structure, has the ability to learn and solve very complex problems.
A multilayer perceptron (MLP) is a group of perceptrons, organized in multiple layers, that can accurately answer complex questions. Each perceptron in the first layer (on the left) sends signals to all the perceptrons in the second layer, and so on. An MLP contains an input layer, at least one hidden layer, and an output layer [27].

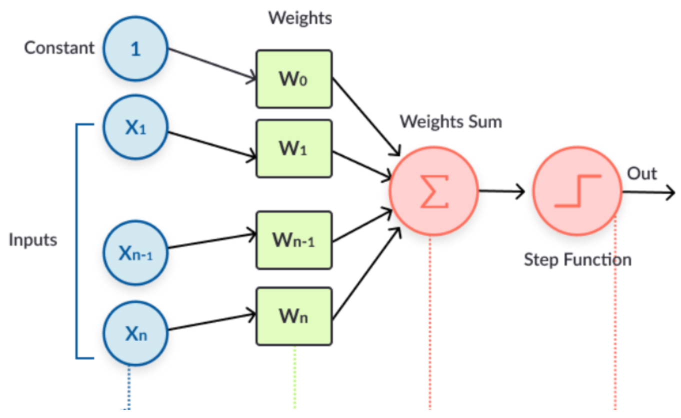


Figure 14: The Perceptron Learning Process [27]

The perceptron in Figure 14 learns as follows:

1. Takes the inputs which are fed into the perceptrons in the input layer, multiplies them by their weights, and computes the sum.
2. Adds the number one, multiplied by a “bias weight”. This is a technical step that makes it possible to move the output function of each perceptron (the activation function) up, down, left, and right on the number graph.

3. Feeds the sum through the activation function—in a simple perceptron system, the activation function is a step function.

4. The result of the step function is the output.

4.2.3. Backpropagation

After a neural network is defined with initial weights, and a forward pass is performed to generate the initial prediction, there is an error function which defines how far away the model is from the true prediction. There are many possible algorithms that can minimize the error function—for example, one could do a brute force search to find the weights that generate the smallest error. However, for large neural networks, a training algorithm is needed that is very computationally efficient. Backpropagation is that algorithm—it can discover the optimal weights relatively quickly, even for a network with millions of weights [27].

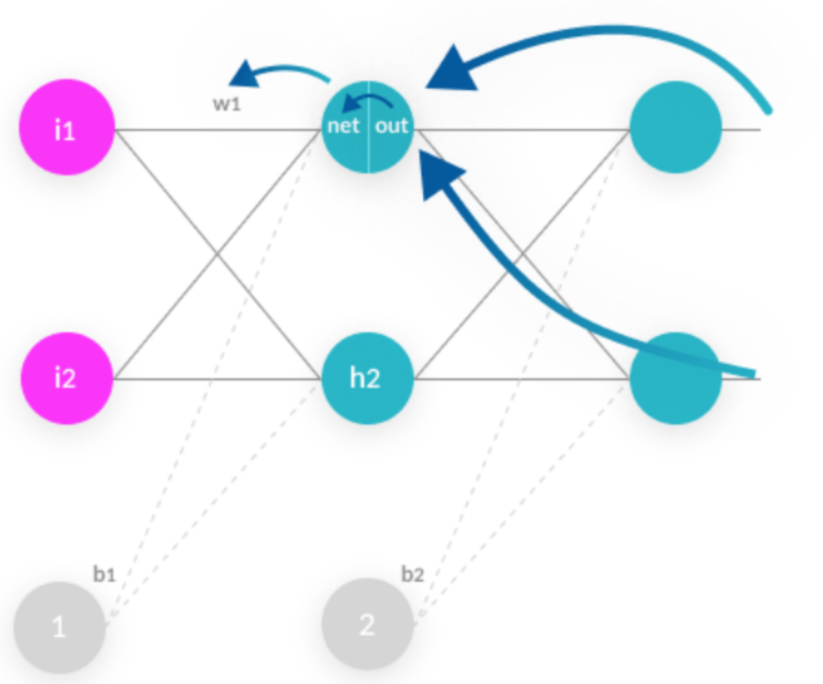


Figure 15: How Backpropagation Works [27]

Here are the steps how Backpropagation works shown in Figure 15:

1. **Forward pass**—weights are initialized and inputs from the training set are fed into the network. The forward pass is carried out and the model generates its initial prediction.

2. **Error function**—the error function is computed by checking how far away the prediction is from the known true value.

3. **Backpropagation with gradient descent**—the backpropagation algorithm calculates how much the output values are affected by each of the weights in the model. To do this,

it calculates partial derivatives, going back from the error function to a specific neuron and its weight. This provides complete traceability from total errors, back to a specific weight which contributed to that error. The result of backpropagation is a set of weights that minimize the error function.

4. **Weight update**—weights can be updated after every sample in the training set, but this is usually not practical. Typically, a batch of samples is run in one big forward pass, and then backpropagation performed on the aggregate result. The batch size and number of batches used in training, called iterations, are important hyperparameters that are tuned to get the best results. Running the entire training set through the backpropagation process is called an epoch [27].

4.2.4. Activation function

Once the linear component is applied to the input, a non-linear function is applied to it. This is done by applying the activation function to the linear combination. The activation function translates the input signals to output signals. The output after application of the activation function would look something like $f(a * W1 + b)$ where $f()$ is the activation function.

In the below Figure 16 we have "n" inputs given as X_1 to X_n and corresponding weights W_{k1} to W_{kn}. We have a bias given as b_k. The weights are first multiplied to its corresponding input and are then added together along with the bias. Let this be called as *u*.

$$u = w * x + b$$

The activation function is applied to u i.e. f(u) and we receive the final output from the neuron as $yk = f(u)$ [28].

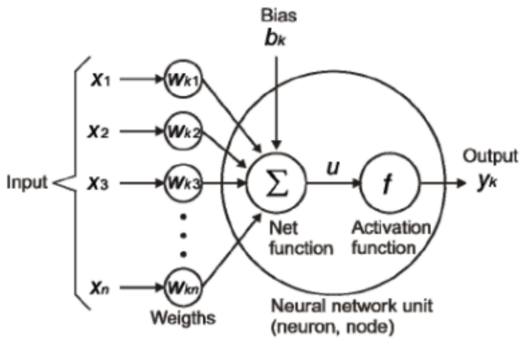


Figure 16: Activation Function [28]

- **Sigmoid** - One of the most common activation functions used is Sigmoid which is shown in Figure 17. It is defined as:

$$sigmoid(x) = 1/(1 + e - x)$$

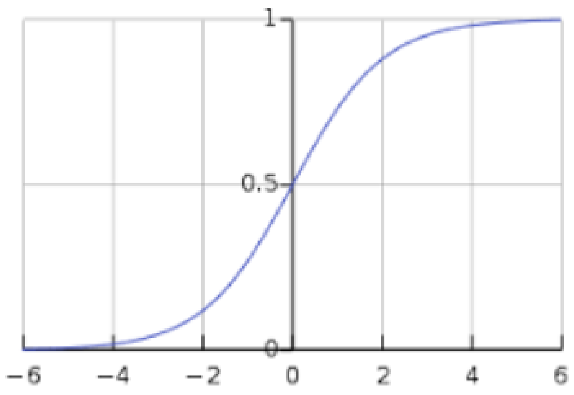


Figure 17: Sigmoid Function [28]

The sigmoid transformation generates a more smooth range of values between 0 and 1. We might need to observe the changes in the output with slight changes in the input values. Smooth curves allow us to do that and are hence preferred over step-functions [28].

- **ReLU (Rectified Linear Units)** -Instead of sigmoids, the recent networks prefer using ReLu activation functions for the hidden layers. The function is defined as:

  f(x) = max(x,0)

  The output of the function is X when X>0 and 0 for X<=0. The function looks like Figure 18:

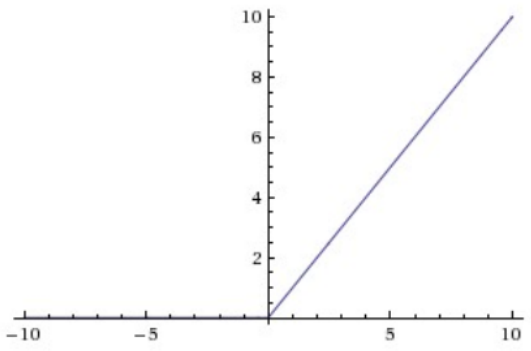


Figure 18: ReLU Function [28]

  The major benefit of using ReLU is that it has a constant derivative value for all inputs greater than 0. The constant derivative value helps the network to train faster [28].

- **Sofmax** - Softmax activation functions [28] are normally used in the output layer for classification problems. It is similar to the sigmoid function, with the only difference being that the outputs are normalized, to sum up to 1. The sigmoid function would work in case of a binary output, however in case we have a multiclass classification problem, softmax makes it really easy to assign values to each class which can be easily interpreted as probabilities.

In Figure 19 is shown that if we are trying to identify a 6 which might also look a bit like 8. The function would assign values to each number as below. We can easily see that the highest probability is assigned to 6, with the next highest assigned to 8 and so on.

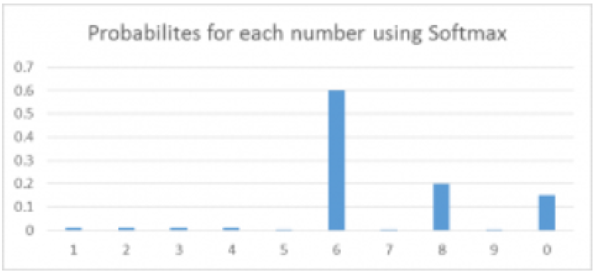


Figure 19: Softmax Function [28]

4.2.5. Bias

In artificial neural networks, the word bias has two meanings:

- It can mean a bias neuron, which is part of the structure of the neural network
- It can mean bias as a statistical concept, which reflects how well the network is able to generate predictions based on the training samples you provide.

4.2.5.1. The bias neuron

In each layer of the neural network, a bias neuron is added, which simply stores a value of 1. The bias neuron makes it possible to move the activation function left, right, up, or down on the number graph as you can see in Figure 21. Without a bias neuron, each neuron takes the input and multiplies it by its weight, without adding anything to the activation equation. This means, for example, it is not possible to input a value of zero and generate an output of two. In many cases it’s necessary to move the entire activation function to the left or to the right, upwards or downwards, to generate the required output values; the bias neuron makes this possible [27].

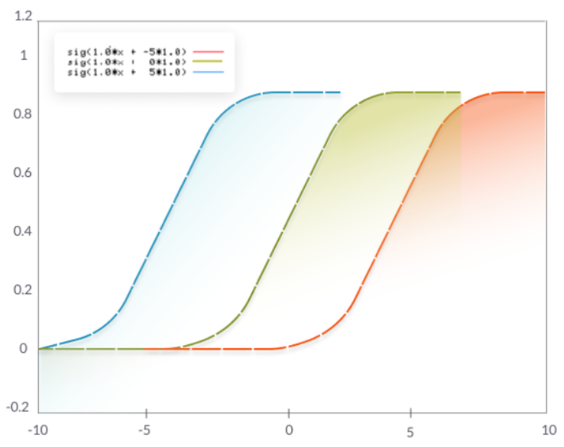


Figure 20: The bias neuron [27]

4.2.5.2. Bias vs. Variance in Neural Networks

To understand bias and variance firstly check the Traning and Validation set and error function definitions in subsection Neural Networks basic concepts.

Bias reflects how well the model fits the training set. A high bias means the neural network is not able to generate correct predictions even for the examples it trained on. **Variance** reflects how well the model fits unseen examples in the validation set. A high variance means the neural network is not able to correctly predict for new examples it hasn't seen before [27].

The perfect solution for the neural network is low Bias and low Variance

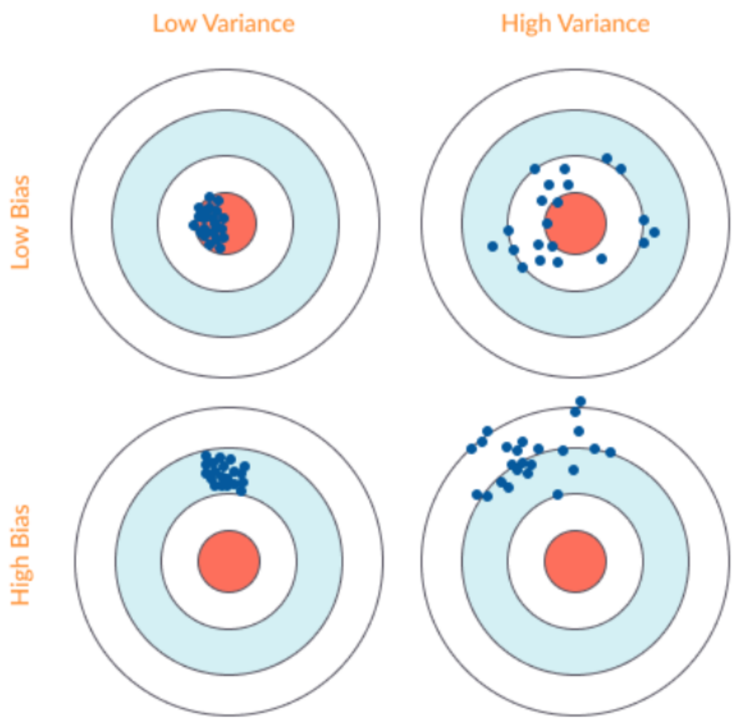


Figure 21: Bias and Variance Illustrated [27]

4.2.6. Overfitting and Underfitting

Overfitting happens when the neural network is good at learning its training set but is not able to generalize its predictions to additional, unseen examples. This is characterized by low bias and high variance. Underfitting happens when the neural network is not able to accurately predict the training set, not to mention for the validation set. This is characterized by high bias and high variance [27].

4.2.6.1. Methods to avoid Overfitting

- **Retraining neural networks**—running the same model on the same training set but with different initial weights, and selecting the network with the best performance.
- **Multiple neural networks**—training several neural network models in parallel, with the same structure but different weights, and averaging their outputs.
- **Early stopping**—training the network, monitoring the error on the validation set after each iteration, and stopping training when the network starts to overfit the data.
- **Regularization**—adding a term to the error function equation, intended to decrease the weights and biases, smooth outputs, and make the network less likely to overfit.
- **Tuning performance ratio**—similar to regularization, but using a parameter that defines by how much the network should be regularized.

4.2.6.2. Methods to avoid Underfitting

- **Adding neuron layers or inputs**—adding neuron layers, or increasing the number of inputs and neurons in each layer, can generate more complex predictions and improve the fit of the model.
- **Adding more training samples or improving quality**—the more training samples you feed into the network, and the better they represent the variance in the real population, the better the network will perform.
- **Dropout**—randomly “kill” a certain percentage of neurons in every training iteration. This ensures some information learned is randomly removed, reducing the risk of overfitting.
- **Decreasing regularization parameter**—regularization can be overdone. By using a regularization performance parameter, you can learn the optimal degree of regularization, which can help the model better fit the data.

4.2.7. Hyperparameters

Hyperparameters determine how the neural network is structured, how it trains, and how its different elements function. Optimizing hyperparameters is an art: there are several ways, ranging from manual trial and error to sophisticated algorithmic methods [27].

4.2.7.1. Model Parameter vs. Hyperparameter

- **A model parameter** is internal to learn your network and is used to make predictions in a production deep learning model. The objective of the training is to learn the values of the model parameters.
- **A hyperparameter** is an external parameter set by the operator of the neural network. For example, the number of iterations of training, the number of hidden layers, or the activation function. Different values of hyperparameters can have a major impact on the performance of the network.

Table 1: List of Common Hyperparameters

Hyperparameters related to neural network structure	Hyperparameters related to the training algorithm
Number of hidden layers	Learning rate
Dropout	Epoch, iterations and batch size
Activation function	Optimizer algorithm
Weights initialization	Momentum

([27])

4.2.7.2. Methods for Hyperparameter Tuning

In a neural network experiment, you will typically try many possible values of hyperparameters and see what works best. In order to evaluate the success of different values, retrain the network, using each set of hyperparameters, and test it against your validation set. If your training set is small, you can use cross-validation—dividing the training set into multiple groups, training the model on each of the groups then validating it on the other groups. According to [27] there are common methods used to tune hyperparameters:

1. **Manual hyperparameter tuning**—an experienced operator can guess parameter values that will achieve very high accuracy. This requires trial and error.
2. **Grid search**—this involves systematically testing multiple values of each hyperparameter and retraining the model for each combination.
3. **Random search**—using random hyperparameter values is actually more effective than manual search or grid search.
4. **Bayesian optimization**—a method that trains the model with different hyperparameter values over and over again, and tries to observe the shape of the function generated by different parameter values. It then extends this function to predict the best possible values. This method provides higher accuracy than random searches.

4.2.8. Classification

There are numerous, highly effective classification algorithms; neural networks are just one of them. The unique strength of a neural network is its ability to dynamically create complex prediction functions, and solve classification problems in a way that emulates human thinking. For certain classification problems, neural networks can provide improved performance compared to other algorithms. However, because neural networks are more computationally intensive and more complex to set up, they may be overkill in many cases. In article [27] types of classification algorithms are given:

- **Logistic regression** (binary)—analyzes a set of data points and finds the best fitting model to describe them. Easy to implement and very effective for input variables that are well known, and closely correlated with the outcome.
- **Decision tree** (multiclass)—classifies using a tree structure with if-then rules, running the input through a series of decisions until it reaches a termination condition. Able to model complex decision processes and is highly intuitive, but can easily overfit the data.
- **Random forest** (multiclass)—an ensemble of decision trees, with automatic selection of the best performing tree. Provides the strength of the decision tree algorithm without the problem of overfitting.
- **Naive Bayes classifier** (multiclass)—a probability-based classifier. Calculates the likelihood that each data point exists in each of the target categories. Simple to implement and accurate for a large set of problems, but sensitive to the set of categories selected.
- **k-Nearest neighbor** (multiclass)—classifies each data point by analyzing its nearest neighbors among the training examples. Simple to implement and understand, effective for many problems, especially those with low dimensionality. Provides lower accuracy compared to supervised algorithms, and is computationally intensive.

4.2.8.1. Neural Network Classification Pros and Cons

Pros: Neural networks are very effective for high dimensionality problems, or with complex relations between variables. For example, neural networks can be used to classify and label images, audio, and video, perform sentiment analysis on text, and classify security incidents into risk categories. **Cons:** Neural networks are theoretically complex, difficult to implement, requiring careful fine-tuning, and computationally intensive. Unless you're a deep learning expert, you will usually derive more value from another classification algorithm if it can provide similar performance [27].

4.2.9. Regression

Regression models have proven useful in modeling problems and providing predictions. This is the classic linear regression function:

$$Y =_1 +_2 X_2 + ... +_k X_k$$

In a regression model, the inputs are called independent values ($X_{1..K}$in the equation above). The output is called the dependent value (y in the equation above). There are weights called coefficients, which determine how much each input value contributes to the result, or how important it is ($_{1..K}$ in the equation above). Neural networks are able to model complex problems, using a learning process that emulates the human brain. Neural networks can generate a model that approximates any regression function. Moreover, most regression models do not fit the data perfectly, and neural networks can generate a more complex model that will provide higher accuracy [27].

Types of Regression Analysis:

- Linear regression—suitable for dependent values which can be fitted with a straight line (linear function).
- Polynomial regression—suitable for dependent variables that can be fitted by a curve or series of curves.
- Logistic regression—suitable for dependent variables that are binary, and therefore not normally distributed.
- Stepwise regression—an automated technique that can deal with the high dimensionality of independent variables.
- Ridge regression—a regression technique that helps with multicollinearity, independent variables that are highly correlated. It adds a bias to the regression estimates, penalizing coefficients using a shrinkage parameter.
- Lasso regression—like Ridge regression, shrinks coefficients to solve multicollinearity, however, it also shrinks the absolute values, meaning some of the coefficients can become zero. This performs “feature selection”, removing some variables from the equation.
- ElasticNet regression—combines Ridge and Lasso regression, and is trained with L1 and L2 regularization, trading off between the two techniques.

4.2.9.1. Regression in Neural Networks

Neural networks are a far more complex mathematical structure than regression models, but they are reducible to regression equations. Essentially, any regression equation can be modeled by a neural network. For example, a very simple neural network, which takes them several inputs, multiplies them by weights, and passes them through a step function, is equivalent to logistic regression. A slightly more complex neural network can be constructed to model a multi-class regression classification, using the Softmax activation function to generate probabilities for each class, which can be normalized, to sum up to 1. If the data is complex and the regression function cannot model it accurately, a neural network can create a more complex mathematical structure that will accurately represent the data. In these more complex cases, neural networks can have much more predictive power and can perform much better than a regression function [27].

4.2.10. Convolutional Neural Networks

Convolutional neural networks represent a specialization of generic neural networks, where the individual neurons form a mathematical approximation of the biological visual receptive field. Visual receptive fields corresponding to small regions of the input that are processed by the same unit. The receptive fields of the neighboring neurons overlap, allowing thus the robustness of the learned representation to small translations of the input. Each receptive field learns to react to a specific feature (automatically learned as a kernel). By combining many layers, the network forms a classifier that is able to automatically learn relevant features and is less prone to translational variance in data. In this section, the specific layers (convolutional and pooling layers) of CNNs will be explained. A CNN showed in Figure 22 is finally built by combining many convolutional and pooling layers, so the number of output in each successive layer grows, while size of images on the output is reducing. The output of the last CNN layer is a vector image representation [23].

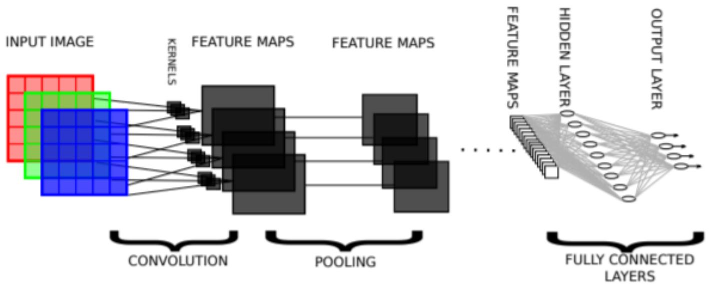


Figure 22: CNN Architecture [23]

4.2.10.1. Convolution

Convolutional layers compute feature maps by convolving the previous layer with a specific kernel. Let M^l be a feature map of layer *l* and M^{l-1} a feature map of the previous layer. The width and height of a feature map is indicated with M^w and M^h while the width and height of kernels are indicated with K^w and K^h. Let S^w and S^h represent the horizontal and vertical steps of the kernel during a convolution operation. The sizes of the output feature maps (of the current layer) are then dependent on the sizes of feature maps from the previous layer, kernels, and stepping factors. The output width and height are given by Eq (1) and Eq. (2), correspondingly.

$$M_w^l = \frac{M_w^{l-1} - K_w}{S_w} + 1 \quad (1)$$

$$M_h^l = \frac{M_h^{l-1} - K_h}{S_h} + 1 \quad (2)$$

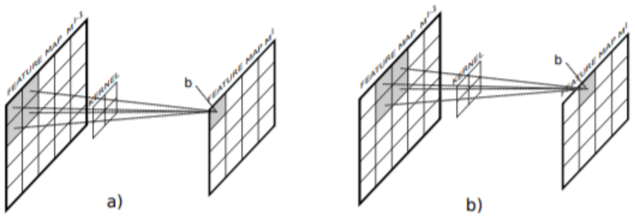


Figure 23: Illustrated convolution step [23]

After each convolution, a bias is added to each element and the result is passed through an activation function. Convolutional layers can either have full connectivity or sparse connectivity. In the case of full connectivity, each feature map of the current layer is connected with every feature map from the previous layer. Each connection is represented by a kernel.
Sparse connectivity (Figure 24) is a way of connecting feature maps in convolution operations where each feature map from the current layer is connected only to a specific subset of feature maps from the previous layer. The benefits of this approach are reduced computational complexity and improved generalization, as the network is forced to learn different features. When using fully connected convolutional layers there is a chance that the network will learn a less diverse set of features according to [23].

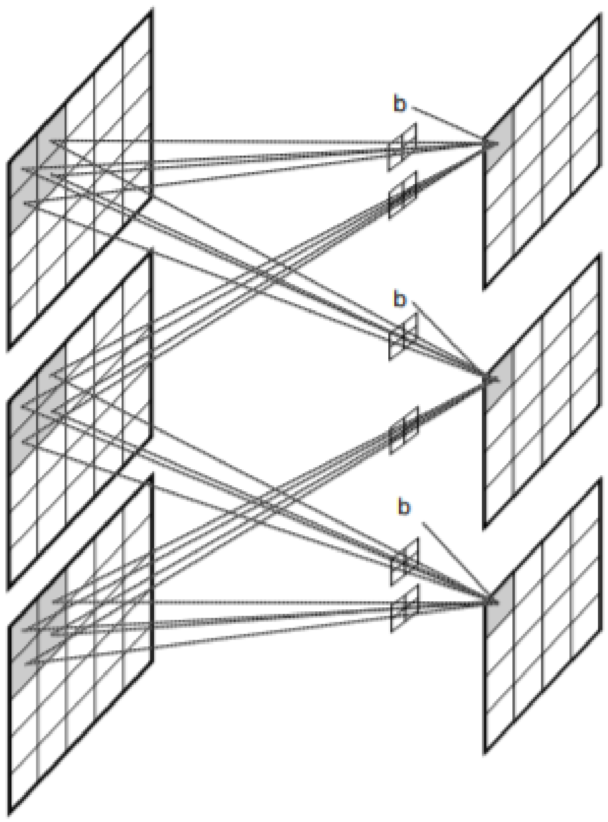


Figure 24: Sparse connectivity of a convolutional layer [23]

4.2.10.2. Filters

A filter in a CNN is like a weight matrix with which we multiply a part of the input image to generate a convoluted output. Let's assume we have an image of size 28*28. We randomly assign a filter of size 3*3, which is then multiplied with different 3*3 sections of the image to form what is known as a convoluted output. The filter size is generally smaller than the original image size. The filter values are updated like weight values during backpropagation for cost minimization.

Consider the below Figure 25. Here the filter is a 3*3 matrix which is multiplied with each 3*3 section of the image to form the convolved feature.

4.2.10.3. Pooling

It is common to periodically introduce pooling layers in between the convolution layers. This is basically done to reduce the number of parameters and prevent over-fitting. The most common type of pooling is a pooling layer of filter size(2,2) using the MAX operation. What it would do is, it would take the maximum of each 4*4 matrix of the original image which is shown in Figure 26. Average pooling is another pooling technique, but max pooling has shown to work better in practice [28].

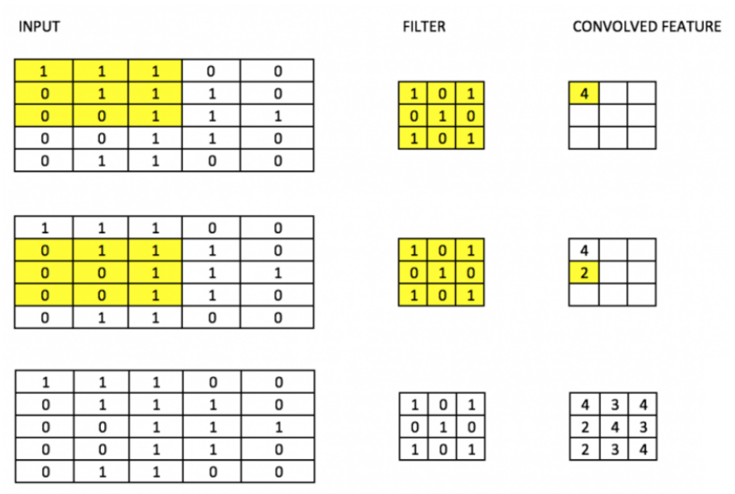


Figure 25: Filters [28]

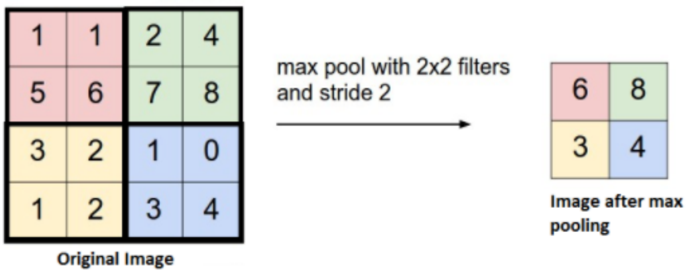
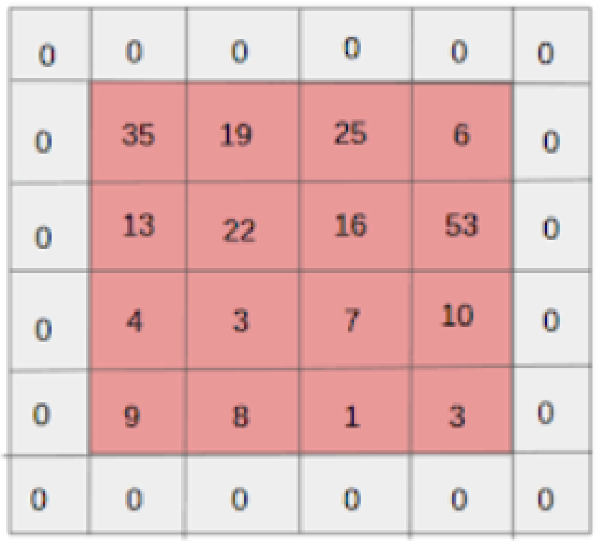


Figure 26: Illustrated pooling [28]

4.2.10.4. Padding

Padding refers to adding an extra layer of zeros across the images so that the output image has the same size as the input. This is known as the same padding, you can see it in Figure 27. After the application of filters, the convolved layer in the case of the same padding has the size equal to the actual image.

Valid padding refers to keeping the image as such a having all the pixels of the image which are actual or “valid”. In this case, after the application of filters the size of the length and the width of the output keeps getting reduced at each convolutional layer [28].



0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figure 27: Same padding [28]

4.2.11. LeNet-5

LeNet is a convolutional neural network structure proposed by Yann LeCun et al. in 1998. In general, LeNet refers to lenet-5 and is a simple convolutional neural network. Convolutional neural networks are a kind of feed-forward neural network whose artificial neurons can respond to a part of the surrounding cells in the coverage range and perform well in large-scale image processing [20]. Example of LeNet-5 structure are given in Figure 28.

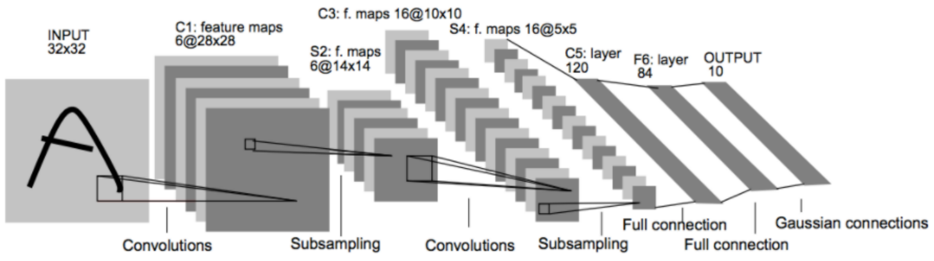


Figure 28: LeNet-5 Architecture [25]

4.3. Used Python Libraries

In this section, all used Python Libraries are listed and shortly introduced.

4.3.1. TensorFlow

TensorFlow [30] is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in Machine Learning and developers easily build and deploy Machine Learning powered applications. TensorFlow is used for implementation of Traffic Sign Classifier.

4.3.2. OpenCV

OpenCV (Open Source Computer Vision Library) [15] is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc. OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 18 million. The library is used extensively in companies, research groups and by governmental bodies.

4.3.3. Matplotlib

Matplotlib [31] is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+. There is also a procedural "pylab" interface based on a state machine (like OpenGL), designed to closely resemble that of MATLAB, though its use is discouraged.

4.3.4. Numpy

Numpy [16] is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors.

4.3.5. Pandas

Pandas [32] is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license. The name is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals.

4.3.6. Sckit-learn

Scikit-learn (sklearn) [33] is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

4.3.7. Sys

System-specific parameters and functions [17] module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

4.3.8. Glob

The glob [34] module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but *, ?, and character ranges expressed with [] will be correctly matched. This is done by using the os.scandir() and fnmatch.fnmatch() functions in concert, and not by actually invoking a subshell. Note that unlike fnmatch.fnmatch(), glob treats filenames beginning with a dot (.) as special cases. (For tilde and shell variable expansion, use os.path.expanduser() and os.path.expandvars().)

4.3.9. Pickle

The pickle [35] module implements binary protocols for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” 1 or “flattening”; however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

4.4. Implementation of Traffic Sign Classifier

Accordingly to [36] in this section Traffic Sign Classifier Implementation will be explained. Firstly all specifications of dataset will be presented and after that model architecture will be explained and after that implementation of Traffic Sign Classifier as a Docker service will be explained in detail.

4.4.1. Loading data

The following code is for loading data from pickle files which were stored to appropriate variables for the train, validation, and test sets. From German Traffic Sign Benchmarks (GTSRB) [22] images are taken and sorted in those three sets.

```python
import pickle

training_file = './data/train.p'
validation_file= './data/valid.p'
testing_file =  './data/test.p'

with open(training_file, mode='rb') as f:
    train = pickle.load(f)
with open(validation_file, mode='rb') as f:
    valid = pickle.load(f)
with open(testing_file, mode='rb') as f:
    test = pickle.load(f)

X_train, y_train = train['features'], train['labels']
X_valid, y_valid = valid['features'], valid['labels']
X_test, y_test = test['features'], test['labels']
```

4.4.2. Dataset Summary

The pickled data is a dictionary with 4 key/value pairs:

- ’features’ is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).
- ’labels’ is a 1D array containing the label/class id of the traffic sign. The file signnames.csv contains an id -> name mappings for each id.
- ’sizes’ is a list containing tuples, (width, height) representing the original width and height of the image.

- 'coords' is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. These coordinates assume the original image. The pickled data contains resized versions (32x32) of the images.

```
import numpy as np

n_train = len(X_train)
n_valid = len(X_valid)
n_test = len(X_test)

image_shape = X_train[0].shape
unique_classes, n_samples = np.unique(y_train,
                                          return_index=False,
                                          return_inverse=False,
                                          return_counts=True)
n_classes = len(unique_classes)

print("Number of training examples =", n_train)
print("Number of validation examples =", n_valid)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print("Number of classes =", n_classes)
```

And here is a basic summary of the dataset which was the result of the code above:

- The size of the training set is 34799 pictures
- The size of the validation set is 4410 pictures
- The size of the test set is 12630 pictures
- The shape of a traffic sign image is (32, 32, 3)
- The number of unique classes/labels in the data set is 43

Here is an exploratory visualization of the data set. It is a bar chart in Figure 29 showing how the data is distributed alongside our 43 unique label points.

```
import matplotlib.pyplot as plt

labels_unique = []
for i in range(n_classes):
    labels_unique.append(label_signs[i])

index = random.randint(0, len(X_train))
```

```
fig = plt.figure(figsize=(15, 6))
ax = fig.add_subplot(111)
ax.set_title('Sample distribution of Labels')
# ax.set_xlabel('Classes')
ax.set_xticks(np.arange(len(labels_unique)))
ax.set_xticklabels(labels_unique, rotation='vertical')
ax.set_ylabel('Number of Samples')
plt.bar(range(n_classes), n_samples)
plt.show()
```

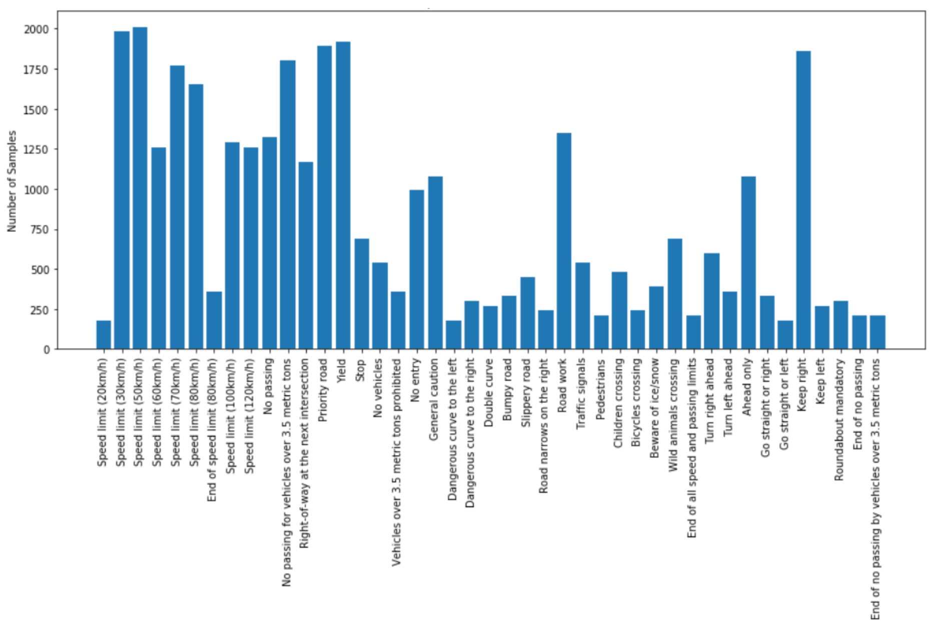


Figure 29: Sample distribution of Labels [36]

As we can see in Figure 29 labels marked as Speed limit(30km/h), Speed limit(50km/h), Speed limit(70km/h), and Speed limit(80km/h), along with No Passing for vehicles over 3.5 metric tones, Priority work, Yield, Road Work, Keep right are among the top labels with the majority of samples. Whereas, certain labels like Speed limit(20km/h), Dangerous Curve to the left, Pedestrians, and Go Straight or left are among the few labels with the least amount of samples.

4.4.3. Pre-process the Dataset

To pre-process the data, the following steps were taken on all train, validation, and test set:

- **Converting to grayscale** - This worked well for both authors Sermanet and LeCun as described in their article [37]. For this part of the code, it also helped reduce training time by a lot as the color channels were dropped from 3 to 1. Signs in grayscale are shown in Figure 5.

```
# Convert to grayscale
def convert_RGB_GRY(data, keepdims):
    input_rgb = data
    input_gry = np.sum(data/3, axis=3, keepdims=keepdims)
    return input_rgb, input_gry

X_train_rgb, X_train_gry = convert_RGB_GRY(X_train, keepdims=True)
X_valid_rgb, X_valid_gry = convert_RGB_GRY(X_valid, keepdims=True)
X_test_rgb, X_test_gry = convert_RGB_GRY(X_test, keepdims=True)

# Visualize rgb vs grayscale
n_rows = 8
n_cols = 10
offset = 9000
fig, axs = plt.subplots(n_rows,n_cols, figsize=(18, 14))
fig.subplots_adjust(hspace = .1, wspace=.001)
axs = axs.ravel()
for j in range(0,n_rows,2):
    for i in range(n_cols):
        index = i + j*n_cols
        image = X_train_rgb[index + offset]
        axs[index].axis('off')
        axs[index].imshow(image)
    for i in range(n_cols):
        index = i + j*n_cols + n_cols
        image = X_train_gry[index + offset - n_cols].squeeze()
        axs[index].axis('off')
        axs[index].imshow(image, cmap='gray')
```

- **Normalizing the data to the range (-1,1)** - This was done using the line of code $(dataset-128)/128$ which is shown in the code below. The resulting dataset mean wasn't exactly zero, but it was reduced from being around 82.677589037 to roughly around -0.354081335648. This helps during training time as it reduces the possibility of having a wider distribution in the data which would have made it more difficult and to train using a singular learning rate. Normalized and original images are shown in Figure 31.

```
# Normalize the train, valid and test datasets to (-1,1)
X_train_normalized = (X_train - 128)/128
X_valid_normalized = (X_valid - 128)/128
X_test_normalized = (X_test - 128)/128

print("Original shape:", X_train.shape)
print("Normalized shape:", X_train_normalized.shape)
```

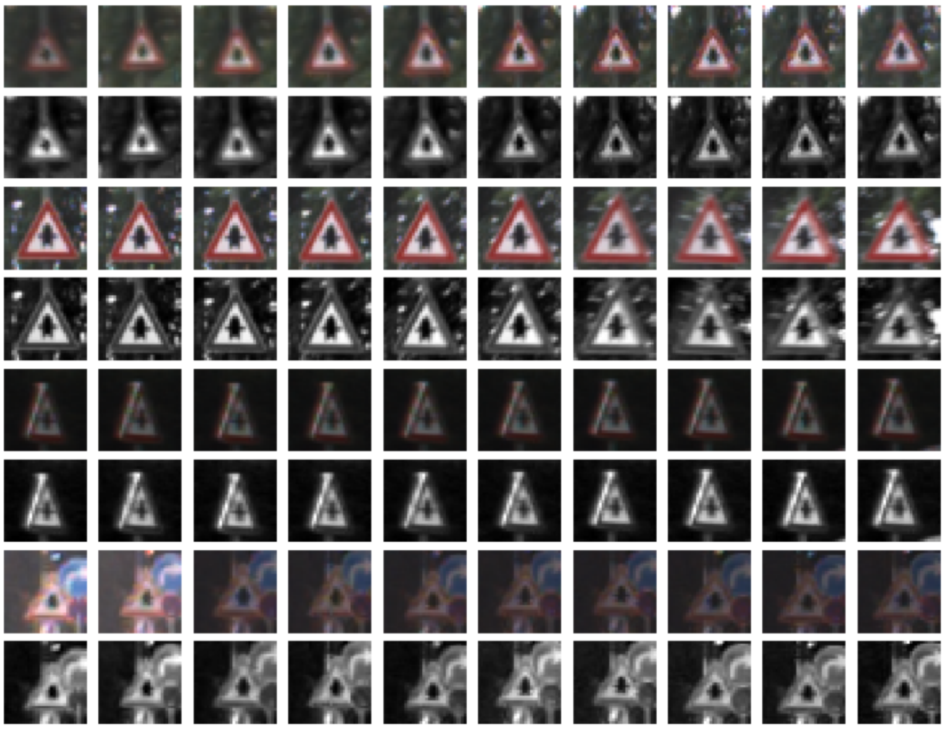


Figure 30: Signs converted to grayscale [36]

```
fig, axs = plt.subplots(1,2, figsize=(10, 3))
axs = axs.ravel()

axs[0].axis('off')
axs[0].set_title('normalized')
axs[0].imshow(X_train_normalized[0].squeeze(), cmap='gray')

axs[1].axis('off')
axs[1].set_title('original')
axs[1].imshow(X_train[0].squeeze(), cmap='gray')
```

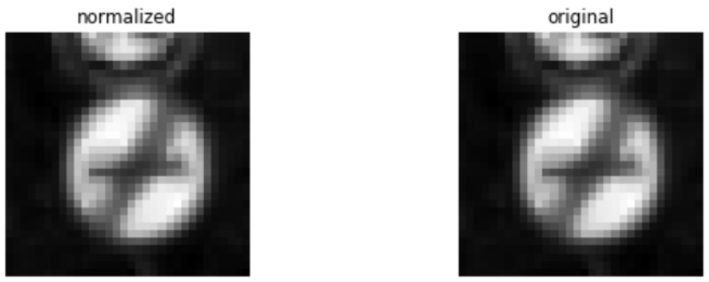


Figure 31: Normalized image vs. Original [36]

- **Shuffle the training set** -The last step to do is to shuffle the training set which is obtained with the following lines of code. By shuffling data, we ensure that each data point creates an "independent" change on the model, without being biased by the same points before them.

```
# Shuffle the training set.
from sklearn.utils import shuffle

X_train, y_train = shuffle(X_train, y_train)
```

4.4.4. Model Architecture - LeNet-5

In Table 2 the implementation of LeNet CNN is given according to [36] with these parameters:

- This LeNet architecture accepts a 32x32xC image as input, where C is the number of color channels. For this case after preprocessing, initially, C would have been 3 due to three layers of RGB Color channel but now as images are in left gray color channel, therefore C = 1.
- ReLU is used as Activation Function.
- Number of Epochs is set to 60 and Batch Size to 100.
- Learning rate: 0.0009
- Dropout keep probability: 0.5
- Architecture is given in Table 2.

Table 2: LeNet Architecture

Layer	Description
Input	32x32x1 Gray Scale image (preprocessed)
Layer 1 - Convolution 5x5	[1,2,2,1] stride, valid padding, output shape 28x28x6
RELU	Activation Method
Max pooling	2x2 stride, outputs 14x14x6 (refered as subsampling in the paper)
Layer 2 - Convolution 5x5	[1,2,2,1] stride, valid padding, output shape 10x10x16
ReLU	Activation Method
Max pooling	2x2 stride, outputs 5x5x16 (refered as subsampling in the paper)
Layer 3 - Convolution 5x5	[1,2,2,1] stride, valid padding, output shape 1x1x400
ReLU	Activation Method
Flatten & Merge	Flatten the output shape of the Layer 2 Max Pooling and Layer 3 Convlution output of 1x800. The easiest way to do is by using tf.contrib.layers.flatten.
Layer 4 - Fully Connected	This will have 800 outputs
Dropout	Here we drop and kep probabilities
Layer 5 - Fully Connected (Logits)	This should result in 43 outputs as there are 43 total classes.

([36])

The following code is implementation of LeNet architecture which is defined above.

```
import tensorflow as tf
from tensorflow.contrib.layers import flatten

EPOCHS = 60
BATCH_SIZE = 100

def LeNet(x):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    W1 = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu,
    stddev = sigma))
    x = tf.nn.conv2d(x, W1, strides=[1, 1, 1, 1], padding='VALID')
    b1 = tf.Variable(tf.zeros(6), name="b1")
```

```
x = tf.nn.bias_add(x, b1)
print("1st Layer shape:",x.get_shape())

# Activation.
x = tf.nn.relu(x)

# Max Pooling. Input = 28x28x6. Output = 14x14x6.
x = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='VALID', name='conv1')
x1 = x
print("1st Layer after max pooling shape:",x1.get_shape())

# Layer 2: Convolutional. Output = 10x10x16.
W2 = tf.Variable(tf.truncated_normal(
shape=(5, 5, 6, 16), mean = mu,
stddev = sigma)
)
x = tf.nn.conv2d(x, W2, strides=[1, 1, 1, 1], padding='VALID')
b2 = tf.Variable(tf.zeros(16), name="b2")
x = tf.nn.bias_add(x, b2)
print("2nd Layer shape:",x.get_shape())

# Activation.
x = tf.nn.relu(x)

# Max Pooling. Input = 10x10x16. Output = 5x5x16.
x = tf.nn.max_pool(
x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
    padding='VALID', name='conv2'
)
x2 = x
print("2nd Layer after max pooling shape:",x2.get_shape())

# Layer 3: Convolutional. Output = 1x1x400.
W3 = tf.Variable(tf.truncated_normal(shape=(5, 5, 16, 400),
mean = mu, stddev = sigma))
x = tf.nn.conv2d(x, W3, strides=[1, 1, 1, 1], padding='VALID')
b3 = tf.Variable(tf.zeros(400), name="b3")
x = tf.nn.bias_add(x, b3)
print("3rd Layer shape:",x.get_shape())

# Activation.
x = tf.nn.relu(x)
```

```
    x3 = x

    # Layer 4: Flatten. Input = 5x5x16. Output = 400.
    flat_layer2 = flatten(x2)
    print("After Flattening 2nd Layer post max pooling shape:",
    flat_layer2.get_shape())

    # Flatten x. Input = 1x1x400. Output = 400.
    flat_layer3 = flatten(x3)
    print("After Flattening 3rd Layer after activation shape:",
    flat_layer3.get_shape())

    # Concat layer2flat and x. Input = 400 + 400. Output = 800
    x = tf.concat([flat_layer3, flat_layer2], 1)
    print("After Concatenating Flat Layers shape:",x.get_shape())

    # Dropout
    x = tf.nn.dropout(x, keep_prob)
    print("Dropout Layer shape:",x.get_shape())

    # TODO: Layer 4: Fully Connected. Input = 800. Output = 43.
    W4 = tf.Variable(tf.truncated_normal(shape=(800, 43),
    mean = mu, stddev = sigma))
    b4 = tf.Variable(tf.zeros(43), name="b4")
    logits = tf.add(tf.matmul(x, W4), b4)
    print("Logits shape:",logits.get_shape())

    return logits

print('done')
```

4.4.5. Training the Model and validation

In the code below x is a placeholder for a batch of input images. y is a placeholder for a batch of output labels.

```
x = tf.placeholder(tf.float32, (None, 32, 32, 1))
y = tf.placeholder(tf.int32, (None))
x_final_test = tf.placeholder(tf.float32, (None, 32, 32, 1))
x_final_graph = tf.placeholder(tf.float32, (None, 32, 32, 1))
keep_prob = tf.placeholder(tf.float32) # probability to keep units
one_hot_y = tf.one_hot(y, 43)
```

A validation set can be used to assess how well the model is performing. A low accuracy on the training and validation sets imply underfitting. High accuracy on the training set but low accuracy on the validation set implies overfitting.

Below is the code used for training and evaluation.

```
rate = 0.0009

logits = LeNet(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y,
logits=logits)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
saver = tf.train.Saver()

def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE],
        y_data[offset:offset+BATCH_SIZE]
        accuracy = sess.run(
                accuracy_operation,
                feed_dict={x: batch_x, y: batch_y, keep_prob: 1.0}
            )
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

In code below represents the execution of training the model and those steps are performed:

1. Running the training data through the training pipeline to train the model.
2. Before each epoch, the training set is shuffled.
3. After each epoch, the loss, and accuracy of the validation set is measured.
4. After the training model is saved.

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)
    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={
                    x: batch_x, y: batch_y, keep_prob: 0.5
                })

        valid_acc = evaluate(X_valid, y_valid)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(valid_acc))
        print()

    saver.save(sess, 'lenet')
    print("Model saved")
```

```
EPOCH 60 ...
Validation Accuracy = 0.965
```

Last step is to evaluate the model performance on the test set and this step is done only once.

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    test_accuracy = evaluate(X_test, y_test)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
```

```
Test Accuracy = 0.942
```

4.4.6. Test a Model on New Images

The model is tested on new Images which was later used for the implementation of Docker service. Predicted signs are shown in Figure 32 and final model results after training were given in Tables 3 and 4.

The model seems to be 0.06078% sure about the predicted labels. Well, this still is not the most robust model. This can be tested to improve by incorporating more features in the dataset preprocessing like brightness adjustments, data augmentation, image resizing, and stabilizing.

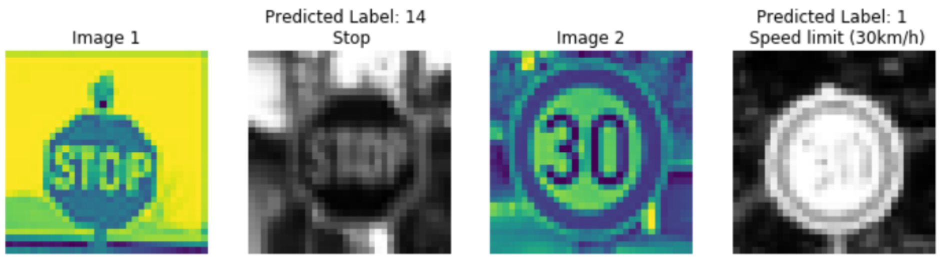


Figure 32: Predicted labels [36]

Table 3: Softmax Probabilities of predicted labels for image 1 in Figure 32

Label	Softmax probabilities
Right-of-way at the next intersection	
Speed limit (20km/h)	0.06078682
Speed limit (30km/h)	0.02236222
Speed limit (50km/h)	0.02236222
Speed limit (60km/h)	0.02236222

([36])

Table 4: Softmax Probabilities of predicted labels for image 2 in Figure 32

Label	Softmax probabilities
Speed limit (30km/h)	0.06078682
Speed limit (20km/h)	0.02236222
Speed limit (50km/h)	0.02236222
Speed limit (60km/h)	0.02236222
Speed limit (70km/h)	0.02236222

([36])

4.4.7. Traffic Sign Classifier as Docker service

After training the model in this subsection implementation of the final Docker service is presented. Python script for Traffic Sign Classification accepts one parameter which is the path to the folder with images for the test. Unfortunately, because of TensorFlow this script must have LeNet architecture and loaded training set to execute the sign prediction which is shown in the code below. For a simpler and faster implementation of this service, PyTorch can be used. In PyTorch, only the trained model needs to be loaded to execute the sign prediction but this is not implemented in this project.

```
import pickle
import sklearn
from sklearn.utils import shuffle
import matplotlib.image as mpimg
import pandas as pd
import glob
import cv2
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.contrib.layers import flatten
import sys

#training data location
training_file = "train.p"

with open(training_file, mode='rb') as f:
    train = pickle.load(f)

X_train, y_train = train['features'], train['labels']


def convert_RGB_GRY(data, keepdims):
    input_rgb = data
    input_gry = np.sum(data/3, axis=3, keepdims=keepdims)
    return input_rgb, input_gry


X_train_rgb, X_train_gry = convert_RGB_GRY(X_train, keepdims=True)

X_train = X_train_gry

X_train_normalized = (X_train - 128)/128

X_train = X_train_normalized

X_train, y_train = shuffle(X_train, y_train)


def LeNet(x):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    W1 = tf.Variable(tf.truncated_normal(
        shape=(5, 5, 1, 6), mean=mu, stddev=sigma))
```

```
x = tf.nn.conv2d(x, W1, strides=[1, 1, 1, 1], padding='VALID')
b1 = tf.Variable(tf.zeros(6), name="b1")
x = tf.nn.bias_add(x, b1)
# print("1st Layer shape:", x.get_shape())

# Activation.
x = tf.nn.relu(x)

# Max Pooling. Input = 28x28x6. Output = 14x14x6.
x = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[
                   1, 2, 2, 1], padding='VALID', name='conv1')
x1 = x
# print("1st Layer after max pooling shape:", x1.get_shape())

# Layer 2: Convolutional. Output = 10x10x16.
W2 = tf.Variable(tf.truncated_normal(
    shape=(5, 5, 6, 16), mean=mu, stddev=sigma))
x = tf.nn.conv2d(x, W2, strides=[1, 1, 1, 1], padding='VALID')
b2 = tf.Variable(tf.zeros(16), name="b2")
x = tf.nn.bias_add(x, b2)
# print("2nd Layer shape:", x.get_shape())

# Activation.
x = tf.nn.relu(x)

# Max Pooling. Input = 10x10x16. Output = 5x5x16.
x = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[
                   1, 2, 2, 1], padding='VALID', name='conv2')
x2 = x
# print("2nd Layer after max pooling shape:", x2.get_shape())

# Layer 3: Convolutional. Output = 1x1x400.
W3 = tf.Variable(tf.truncated_normal(
    shape=(5, 5, 16, 400), mean=mu, stddev=sigma))
x = tf.nn.conv2d(x, W3, strides=[1, 1, 1, 1], padding='VALID')
b3 = tf.Variable(tf.zeros(400), name="b3")
x = tf.nn.bias_add(x, b3)
# print("3rd Layer shape:", x.get_shape())

# Activation.
x = tf.nn.relu(x)
x3 = x
```

```
    # Layer 4: Flatten. Input = 5x5x16. Output = 400.
    flat_layer2 = flatten(x2)
    # print("After Flattening 2nd Layer post max pooling shape:",
    #       flat_layer2.get_shape())

    # Flatten x. Input = 1x1x400. Output = 400.
    flat_layer3 = flatten(x3)
    # print("After Flattening 3rd Layer after activation shape:",
    #       flat_layer3.get_shape())

    # Concat layer2flat and x. Input = 400 + 400. Output = 800
    x = tf.concat([flat_layer3, flat_layer2], 1)
    # print("After Concatenating Flat Layers shape:", x.get_shape())

    # Dropout
    x = tf.nn.dropout(x, keep_prob)
    # print("Dropout Layer shape:", x.get_shape())

    # TODO: Layer 4: Fully Connected. Input = 800. Output = 43.
    W4 = tf.Variable(tf.truncated_normal(
        shape=(800, 43), mean=mu, stddev=sigma))
    b4 = tf.Variable(tf.zeros(43), name="b4")
    logits = tf.add(tf.matmul(x, W4), b4)
    # print("Logits shape:", logits.get_shape())

    return logits


x = tf.placeholder(tf.float32, (None, 32, 32, 1))
y = tf.placeholder(tf.int32, (None))
x_final_test = tf.placeholder(tf.float32, (None, 32, 32, 1))
x_final_graph = tf.placeholder(tf.float32, (None, 32, 32, 1))
keep_prob = tf.placeholder(tf.float32)  # probability to keep units
one_hot_y = tf.one_hot(y, 43)

logits = LeNet(x)

# predict new images


def pipeline(file):
    global X_final_test
    my_images = []
    for i, img in enumerate(glob.glob(file)):
        X_final_test_name.append(img)
        image = cv2.imread(img)
        # print('Input Image ' + str(i) + ' ' + str(image.shape))
        axs[i].axis('off')
        axs[i].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
        my_images.append(image)
    my_images = np.asarray(my_images)
    my_images_gry = np.sum(my_images/3, axis=3, keepdims=True)
    processed_image = (my_images_gry - 128)/128
    # print('After Processing Image: ' + str(processed_image.shape))

    return processed_image


# fetch only sign names
label_signs = pd.read_csv('./signnames.csv').values[:, 1]

# reading in an image
X_final_test = []
X_final_test_name = []
x_final_graph = []

fig, axs = plt.subplots(2, 3, figsize=(3, 2))
fig.subplots_adjust(hspace=.2, wspace=.2)
axs = axs.ravel()


softmax_logits = tf.nn.softmax(logits)
top_k = tf.nn.top_k(softmax_logits, k=3)

def main(folderName):
    X_final_test = pipeline('./' + folderName + '/*.png')
    X_final_graph = X_final_test

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        saver = tf.train.import_meta_graph('./lenet.meta')
        saver.restore(sess, "./lenet")
        my_softmax_logits = sess.run(softmax_logits, feed_dict={
            x: X_final_test, keep_prob: 1.0})
        my_top_k = sess.run(top_k, feed_dict={
                x: X_final_test, keep_prob: 1.0
            })

        for i, image in enumerate(X_final_test):
            pred_label = my_top_k[1][i][0]
            index = np.argwhere(y_train == pred_label)[0]
            print('Predicted Label: {} \n {}'.format(
                pred_label, label_signs[pred_label]))

if __name__ == "__main__":
    folderName = sys.argv[1]
    main(folderName)
```

Final result of service is:

```
Predicted Label: 37
Go straight or left
Predicted Label: 6
End of speed limit (80km/h)
Predicted Label: 26
Traffic signals
Predicted Label: 30
Beware of ice/snow
Predicted Label: 25
Road work
```

5. Object detection

In this chapter, the third perception problem will be presented, and this is the object detection problem. This chapter will be different than the other two because there won't be the implementation of Object detection, instead of that in this chapter, several Object detection approaches will be introduced and compared according to [38] and few other articles.

In the previous chapter Convolutional Neural Networks are explained in theory so this is also important to understand for this chapter. First method Object Detection using Hog Features was the first solution for the Object detection problem that uses the Machine Learning technique Support vector machine. Other approaches are separated into two groups. In the first group, the Object detection problem is solved as a Classification problem, and these methods are Region-based Convolutional Neural Networks (R-CNN), Spatial Pyramid Pooling (SPP-net), Fast R-CNN, and Faster R-CNN. All the listed methods handled detection as a classification problem by building a pipeline where first object proposals are generated and then these proposals are sent to classification/regression heads. In the second group of methods, there are a few methods that pose detection as a regression problem. Two of the most popular ones are You Only Look Once (YOLO) and Single Shot Detector (SSD). These methods use Regression to solve Object detection problem. In the next sections, all listed methods are explained.

5.1. Object Detection using Hog Features

Navneet Dalal and Bill Triggs published a paper [39] in which they introduced a Histogram of Oriented Gradients (HOG) features. Hog features are computationally inexpensive and are good for many real-world problems. On each window obtained from running the sliding window on the pyramid, they calculate Hog Features which are fed to a Support vector machine (SVM) to create classifiers. They run this in real-time on videos for pedestrian detection, face detection, and so many other object detection use-cases. In [40] an implementation of HOG features is given.

Accordingly to [41] here are 6 steps for object detection with HOG features:

1. P positive samples need to be sampled from training data of the object(s) which are wanted to detect and HOG descriptors need to be extracted from these samples.
2. N negative samples need to be sampled from a negative training set that does not contain any of the objects that are wanted to detect and HOG descriptors need to be extracted from these samples as well. In practice N » P.
3. Training a Linear Support Vector Machine on positive and negative samples is the next step.
4. Apply hard-negative mining. For each image and each possible scale of each image in the negative training set, the sliding window technique is applied and the window was slid

across the image. At each window, HOG descriptors are computed and classifier was applied. If classifier (incorrectly) classifies a given window as an object (and it will, there will absolutely be false-positives), record the feature vector associated with the false-positive patch along with the probability of the classification. This approach is called hard-negative mining.

5. Next step is to take the false-positive samples found during the hard-negative mining stage, sort them by their confidence (i.e. probability) and re-train the classifier using these hard-negative samples. (Steps 4-5 can be iteratively applied, but in practice, one stage of hard-negative mining usually [not always] tends to be enough. The gains inaccuracy on subsequent runs of hard-negative mining tend to be minimal.)

6. Classifier is now trained and can be applied to the test dataset. Again, just like in Step 4, for each image in the test set, and for each scale of the image, the sliding window technique is applied. At each window, HOG descriptors are extracted and classifiers applied. If the classifier detects an object with a sufficiently large probability, record the bounding box of the window. After finished scanning the image, non-maximum suppression needs to be applied to remove redundant and overlapping bounding boxes.

5.2. Region-based Convolutional Neural Networks

In [38] object detection was modeled into a classification problem, success depends on the accuracy of classification. After the rise of deep learning, the obvious idea was to replace HOG based classifiers with a more accurate convolutional neural network based classifier. However, there was one problem. CNNs were too slow and computationally very expensive. It was impossible to run CNNs on so many patches generated by a sliding window detector. R-CNN solves this problem by using an object proposal algorithm called **Selective Search** which reduces the number of bounding boxes that are fed to the classifier to close to 2000 region proposals. The selective search uses local cues like texture, intensity, color, and/or a measure of insideness, etc to generate all the possible locations of the object. After that feeding, these boxes to CNN based classifier is performed. The fully connected part of CNN takes a fixed-sized input so, all the generated boxes need to be resized (without preserving aspect ratio) to a fixed size (224×224 for VGG) and feed to the CNN part. Hence, there are 3 important parts of R-CNN a:

1. Run Selective Search to generate probable objects.
2. Feed these patches to CNN, followed by SVM to predict the class of each patch.
3. Optimize patches by training bounding box regression separately.

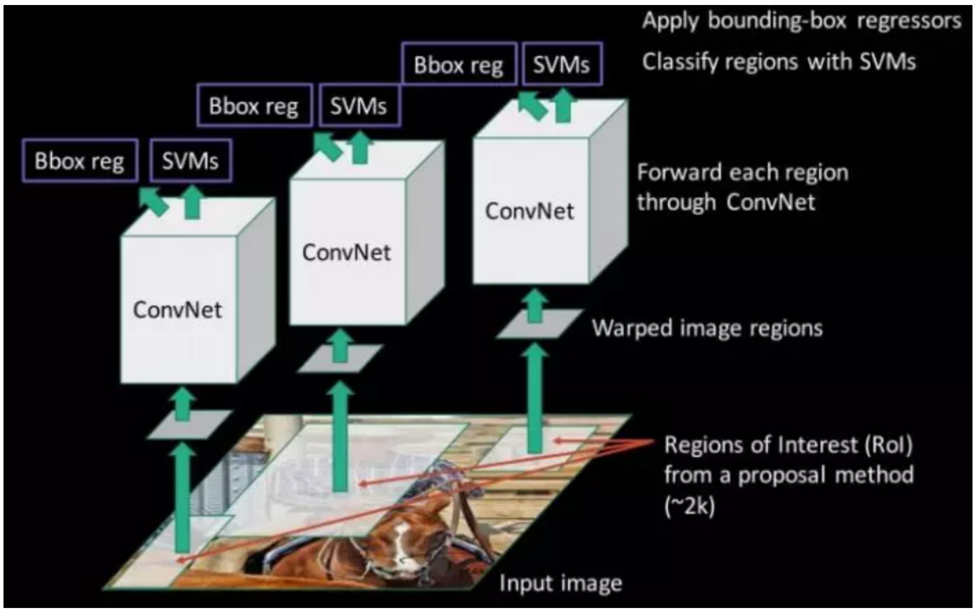

Apply bounding-box regressors
Classify regions with SVMs
Bbox reg
SVMs
Bbox reg
SVMs
Bbox reg
SVMs
ConvNet
ConvNet
ConvNet
Forward each region through ConvNet
Warped image regions
Regions of Interest (RoI) from a proposal method (~2k)
Input image

Figure 33: Applying binding box regressors [38]

5.3. Spatial Pyramid Pooling

Still, RCNN was very slow. Because running CNN on 2000 region proposals generated by Selective search takes a lot of time. SPP-Net tried to fix this. With SPP-net, we calculate the CNN representation for the entire image only once and can use that to calculate the CNN representation for each patch generated by Selective Search. This can be done by performing a pooling type of operation on just that section of the feature maps of the last convolutional layer that corresponds to the region. The rectangular section of the convolutional layer corresponding to a region can be calculated by projecting the region on the convolutional layer by taking into account the downsampling happening in the intermediate layers(simply dividing the coordinates by 16 in case of VGG) [38].

There was one more challenge: to generate the fixed size of input for the fully connected layers of the CNN so, SPP introduces one more trick. It uses spatial pooling after the last convolutional layer as opposed to traditionally used max-pooling. SPP layer divides a region of any arbitrary size into a constant number of bins and the max pool is performed on each of the bins. Since the number of bins remains the same, a constant size vector is produced as demonstrated in Figure 34.

However, there was one big drawback with the SPP net, it was not trivial to perform back-propagation through the spatial pooling layer. Hence, the network only fine-tuned the fully connected part of the network [38]. SPP-Net paved the way for more popular Fast RCNN which we will see next.

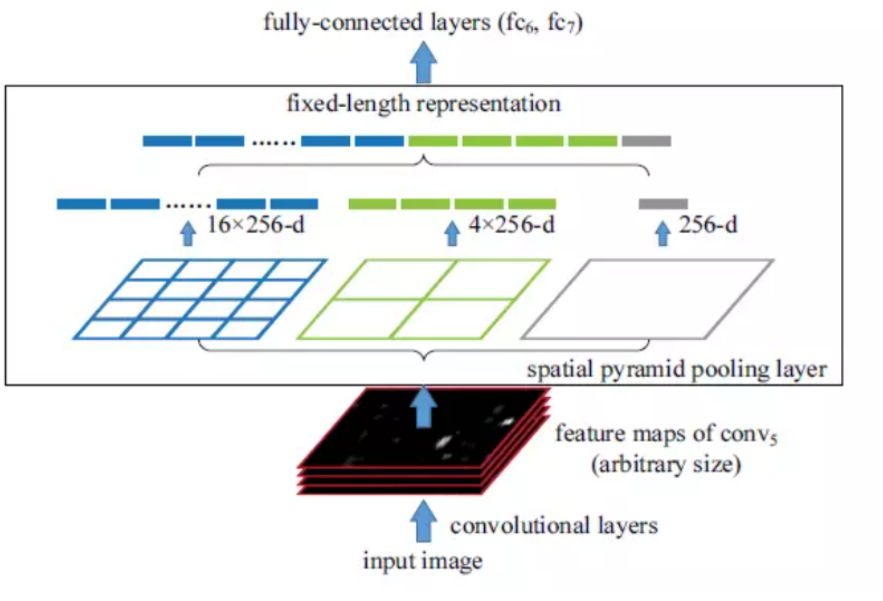

fully-connected layers (fc$_6$, fc$_7$)
fixed-length representation
16×256-d
4×256-d
256-d
spatial pyramid pooling layer
feature maps of conv$_5$ (arbitrary size)
convolutional layers
input image

Figure 34: Spatial Pyramid Pooling Network Arhitecture [38]

5.4. Fast R-CNN

Fast RCNN uses the ideas from SPP-net and RCNN and fixes the key problem in SPP-net i.e. they made it possible to train end-to-end. To propagate the gradients through spatial pooling, It uses a simple back-propagation calculation which is very similar to max-pooling gradient calculation with the exception that pooling regions overlap and therefore a cell can have gradients pumping in from multiple regions.

One more thing that Fast RCNN did that they added the bounding box regression to the neural network training itself. So, now the network had two heads, a classification head, and a bounding box regression head. This multitask objective is a salient feature of Fast-R-CNN as it no longer requires training of the network independently for classification and localization. These two changes reduce the overall training time and increase the accuracy in comparison to SPP net because of the end to end learning of CNN [38].

5.5. Faster R-CNN

The most widely used state of the art version of the R-CNN family — Faster R-CNN was first published in 2015. The slowest part in Fast RCNN was Selective Search or Edge boxes. Faster RCNN replaces selective search with a very small convolutional network called Region Proposal Network to generate regions of Interest that is shown in Figure 35.

To handle the variations in aspect ratio and scale of objects, Faster R-CNN introduces the idea of anchor boxes. At each location, the original paper uses 3 kinds of anchor boxes for scale 128x 128, 256×256, and 512×512. Similarly, for aspect ratio, it uses three aspect ratios 1:1, 2:1, and 1:2. So, In total at each location, there are 9 boxes on which RPN predicts the

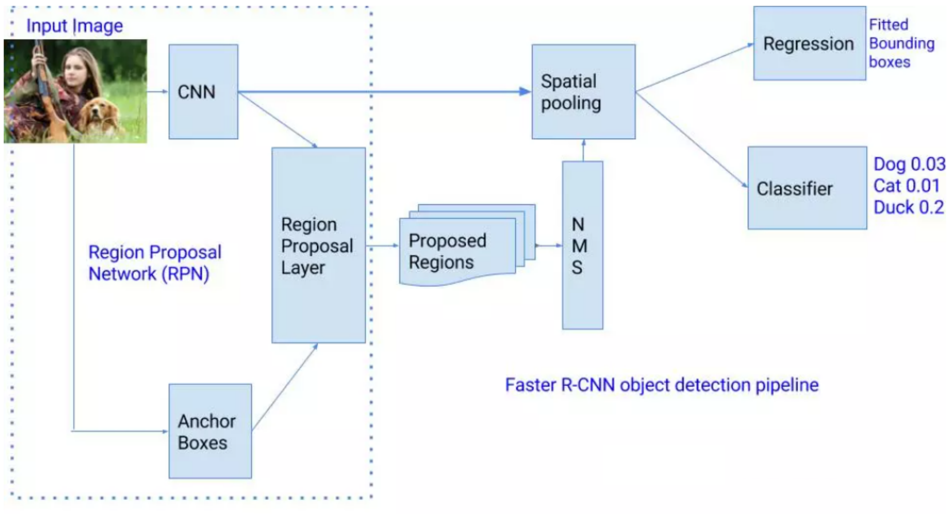


Figure 35: Faster R-CNN Architecture [38]

probability of it being background or foreground. Bounding box regression is applied to improve the anchor boxes at each location. So, RPN gives out bounding boxes of various sizes with the corresponding probabilities of each class. The varying sizes of bounding boxes can be passed further by apply Spatial Pooling just like Fast-RCNN. The remaining network is similar to Fast-RCNN. Faster-RCNN is 10 times faster than Fast-RCNN with a similar accuracy of datasets like VOC-2007. That's why Faster-RCNN has been one of the most accurate object detection algorithms [38].

In paper[42] Faster R-CNN is explained in detail and in article [43] same algorithm is explained but in more beginner-friendly way.

5.6. YOLO

For YOLO, detection is a simple regression problem that takes an input image and learns the class probabilities and bounding box coordinates. YOLO divides each image into a grid of S x S and each grid predicts N bounding boxes and confidence. The confidence reflects the accuracy of the bounding box and whether the bounding box actually contains an object(regardless of class). YOLO also predicts the classification score for each box for every class in training. You can combine both the classes to calculate the probability of each class being present in a predicted box [38].

So, total of SxSxN boxes are predicted. However, most of these boxes have low confidence scores and if we set a threshold say 30% confidence, we can remove most of them as shown in Figure 36.

Notice that at runtime, we have run our image on CNN only once. Hence, YOLO is super fast and can be run real-time. Another key difference is that YOLO sees the complete image at once as opposed to looking at only generated region proposals in the previous methods. So, this contextual information helps in avoiding false positives. However, one limitation for YOLO is that it only predicts 1 type of class in one grid hence, it struggles with very small objects [38].

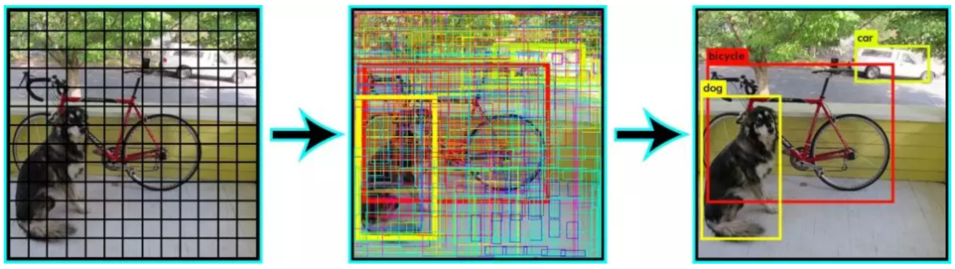


Figure 36: YOLO object detection [38]

In the article [44] YOLOv3 is explained with open source implementation Darknet. More accurate and but also fast implementation YOLOv4 is given in paper[45] and this implementation currently the best implementation of the YOLO algorithm.

5.7. Single Shot Detector

Single Shot Detector achieves a good balance between speed and accuracy. SSD runs a convolutional network on input image only once and calculates a feature map. Now, we run a small 3×3 sized convolutional kernel on this feature map to predict the bounding boxes and classification probability. SSD also uses anchor boxes at various aspect ratios similar to Faster-RCNN and learns the off-set rather than learning the box. In order to handle the scale, SSD predicts bounding boxes after multiple convolutional layers. Since each convolutional layer operates at a different scale, it is able to detect objects of various scales [38].

In the article, [46] SSD is explained in detail on a real-world example.

5.8. Comparison of algorithms

That's a lot of algorithms. Currently, Faster-RCNN is the choice for very high accuracy. However, if you are running it on IoT devices, SSD is a better recommendation. Finally, if accuracy is not too much of a concern but speed is, YOLO will be the way to go. First of all a visual understanding of speed vs accuracy trade-off shown in Figure 37.

SSD seems to be a good choice as it is possible to run it on a video and the accuracy trade-off is very little. However, it may not be that simple, shown in Figure 38 that compares the performance of SSD, YOLO, and Faster-RCNN on various sized objects. At large sizes, SSD seems to perform similarly to Faster-RCNN. However, look at the accuracy numbers when the object size is small, the gap widens.

The choice of the right object detection method is crucial and depends on the problem you are trying to solve and the set-up. Object Detection is the backbone of many practical applications of computer vision such as autonomous cars, security and surveillance, and many industrial applications [38].

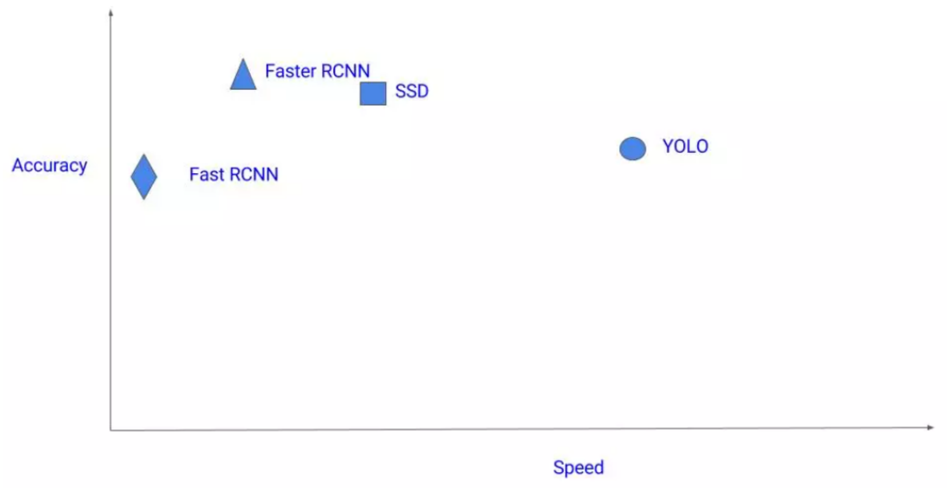


Figure 37: Object detection algorithms compared by speed and accuracy [38]

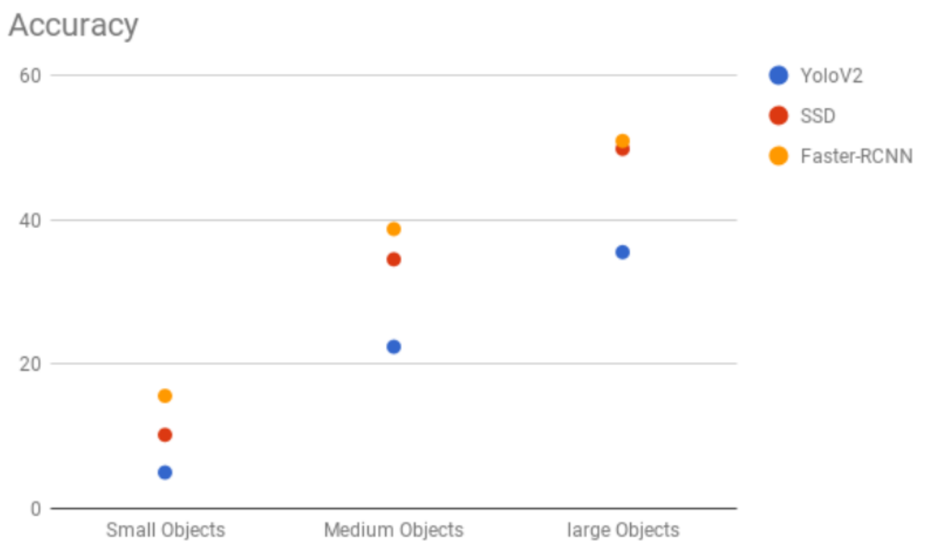


Figure 38: Accuracy of Object detection algorithms compared on different size objects [38]

6. Conclusion

As is shown in the first part the Autonomous vehicle field is very big and complex. The result of this thesis is two docker services that can be used in a robust autonomous driving system. One service is created for lane lines detection and another for traffic sign classification. In the last chapter, an overview of Object detection algorithms is given and a lot of different approaches are taken for this task which means it is one of the key tasks for autonomous driving and it can also be applied in other fields.

6.1. Lane Lines Detection

In the lane line detection algorithm one of the biggest problems was parameter tuning. But, this is a very basic and simple implementation so parameters can be tuned perfectly for straight roads but when a car comes to the cornering algorithm will not perform that good. On the other side if parameters are tuned for the curved road on a straight road algorithm won't work. To solve this problem more advanced techniques can be used.

For this task, the space for improvement is really big. The created task can be improved in several ways. Firstly, with a more advanced robotics approach. An estimation of the geometrical model can be implemented by extracting points of distribution with different kinds of models like line model, polynomial model, spline model. That will lead to fun new challenges such as camera calibration, image distortion, warping, color spaces, gradients, sliding windows, and lane curvature. Secondly, a Deep learning approach with a trained convolutional neural network that will detect lanes and after that lane changing action can be implemented. The deep learning approach is actually used in real-world applications.

6.2. Traffic Sign Classifier

Traffic Sign Classifier is the best basic task to start with deep learning and convolutional neural networks. The amount of theory and new concepts might be a problem for complete beginners but it is worth learning it to have a full picture of what is going on. Convolutional neural network architecture LeNet-5 that is used is the foundation of modern state-of-the-art deep learning-based computer vision. Creating and training models on the GTSRB dataset was the first task for developing docker service. GTSRB dataset is heavily used in Traffic sign classification projects. This is the case because the LeNet-5 architecture which is used on more concrete examples with handwritten digits MNIST dataset can be easily applied on the GTSRB dataset. To create more accurate model parameters tuning is a key factor in this task like in lane lines detection. One problem that might happen while training the model on a larger dataset is that can take a long time on PC so recommended is to train the model on Google Colab or similar platforms. The best result for Test Accuracy that we obtained was 0.942. For higher accuracy parameters can be set better or instead LeNet-5 different model architecture can be implemented like VGGNet.

6.3. Final thoughts

Autonomous driving tasks have proven to be a great way through which deep learning can be studied and applied through a hands-on approach. It is good for beginners because they can choose easier tasks and on the other hand advanced students can choose more complex tasks or simple simulation applications implemented for several tasks that are working together. Another possibility for advanced students is to try multi-task learning which is a big convolutional neural network designed to solve various tasks such as traffic sign detection and classification, vehicle and pedestrian detection, etc. Or beyond perception, there are localization, mapping, control, path planning, etc. It is worth studying some of the tasks because it is a perfect use case for applying AI algorithms and it is the perfect preparation for future jobs because a lot of companies are working on these tasks.

References

[1] B. Marr, *5 Ways Self-Driving Cars Could Make Our World (And Our Lives) Better*, en. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2020/07/17/5-ways-self-driving-cars-could-make-our-world-and-our-lives-better/ (visited on 18/08/2020).

[2] *Orsay Consulting | Resources*, en. [Online]. Available: https://www.orsayconsulting.net/mobility-resources-en (visited on 20/08/2020).

[3] *Tesla Autopilot*. [Online]. Available: https://www.tesla.com/autopilotAI.

[4] L. Fridman, *Self-Driving Cars State-of-the-Art*, 2019. [Online]. Available: https://www.dropbox.com/s/7in6e07mqiynvqi/self_driving_cars_state_of_the_art.pdf?dl=0 (visited on 17/05/2020).

[5] *Autonomous Vehicles & Car Companies l CB Insights*, en-US, Mar. 2020. [Online]. Available: https://www.cbinsights.com/research/autonomous-driverless-vehicles-corporations-list/ (visited on 18/08/2020).

[6] M. Lynberg, *Automated Vehicles for Safety*, publisher: NHTSA, Sep. 2017. [Online]. Available: https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety (visited on 21/08/2020).

[7] *Volvo's futuristic 360c concept 'car' is at once a hot-desk, hotel room and flight cabin*, publisher: NARGESS SHAHMANESH BANKS, Sep. 2018. [Online]. Available: https://www.wallpaper.com/lifestyle/volvo-360c-autonomous-concept-car-review (visited on 18/08/2020).

[8] L. Fridman, *Self-Driving Cars*, 2018. [Online]. Available: https://www.dropbox.com/s/d8exfakm7hw9zl1/self_driving_cars_2018.pdf?dl=0 (visited on 17/05/2020).

[9] *Waypoint - The official Waymo blog: The Waymo Driver's training regimen: How structured testing prepares our self-driving technology for the real world*. [Online]. Available: https://blog.waymo.com/2020/09/the-waymo-drivers-training-regime.html (visited on 11/09/2020).

[10] *Track vision*, en-US. [Online]. Available: https://zalazone.hu/en/track-vision/ (visited on 11/09/2020).

[11] M. Revilloud, D. Gruyer and M. Rahal, 'A new multi-agent approach for lane detection and tracking', in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm: IEEE, May 2016, pp. 3147–3153, ISBN: 9781467380263. DOI: `10.1109/ICRA.2016.7487482`. [Online]. Available: `http://ieeexplore.ieee.org/document/7487482/` (visited on 17/05/2020).

[12] Z. Kim, 'Robust Lane Detection and Tracking in Challenging Scenarios', *IEEE Transactions on Intelligent Transportation Systems*, vol. 9, no. 1, pp. 16–26, Mar. 2008, ISSN: 1558-0016. DOI: `10.1109/TITS.2007.908582`.

[13] D. Tian, *DeepPiCar — Part 4: Autonomous Lane Navigation via OpenCV*, en, May 2019. [Online]. Available: `https://towardsdatascience.com/deeppicar-part-4-lane-following-via-opencv-737dd9e47c96` (visited on 17/05/2020).

[14] M. Hardwick, *Simple lane detection with opencv*, Aug. 2018. [Online]. Available: `https://medium.com/@mrhwick/simple-lane-detection-with-opencv-bfeb6ae54ec0` (visited on 19/05/2020).

[15] *OpenCV*. [Online]. Available: `https://opencv.org/` (visited on 30/08/2020).

[16] *NumPy*. [Online]. Available: `https://numpy.org/` (visited on 30/08/2020).

[17] *Sys — System-specific parameters and functions — Python 3.8.5 documentation*. [Online]. Available: `https://docs.python.org/3/library/sys.html` (visited on 30/08/2020).

[18] R. Keenan, A. Paster and B. Bucholtz, *CarND-LaneLines-P1*, original-date: 2016-10-07T02:12:37Z. [Online]. Available: `https://github.com/udacity/CarND-LaneLines-P1` (visited on 17/05/2020).

[19] M. Ahir, *Finding the right parameters for your Computer Vision algorithm*, Jan. 2017. [Online]. Available: `https://medium.com/@maunesh/finding-the-right-parameters-for-your-computer-vision-algorithm-d55643b6f954` (visited on 19/05/2020).

[20] Y. LeCun, *MNIST Demos on Yann LeCun's website*. [Online]. Available: `http://yann.lecun.com/exdb/lenet/` (visited on 19/08/2020).

[21] Y. LeCun, C. Cortes and C. Burges, *MNIST handwritten digit database*. [Online]. Available: `http://yann.lecun.com/exdb/mnist/` (visited on 19/08/2020).

[22] *German Traffic Sign Benchmarks*. [Online]. Available: `http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset` (visited on 18/08/2020).

[23] V. Vukotić, J. Krapac and S. Šegvić, 'Convolutional Neural Networks for Croatian Traffic Signs Recognition', Zagreb, Croatia: University of Zagreb - Faculty of Electrical Engineering and Computing, Sep. 2014, p. 6. [Online]. Available: `https://www.fer.unizg.hr/_download/repository/0012-0016.pdf`.

[24] D. Ciresan, U. Meier, J. Masci and J. Schmidhuber, 'Multi-column deep neural network for traffic sign classification', Galleria 2, Manno - Lugano 6928, Switzerland: IDSIA - USI - SUPSI, 2012, p. 15. DOI: `10.1016/j.neunet.2012.02.023`. [Online]. Available: `http://people.idsia.ch/~juergen/nn2012traffic.pdf`.

[25] D. Silver, *CarND-LeNet-Lab*, original-date: 2016-12-16T15:31:26Z, Aug. 2020. [Online]. Available: `https://github.com/udacity/CarND-LeNet-Lab` (visited on 18/08/2020).

[26] A. Karpathy, *Stanford University CS231n: Convolutional Neural Networks for Visual Recognition*. [Online]. Available: `http://cs231n.stanford.edu/` (visited on 18/08/2020).

[27] *Complete Guide to Artificial Neural Network Concepts & Models*, en-US. [Online]. Available: `https://missinglink.ai/guides/neural-network-concepts/complete-guide-artificial-neural-networks/` (visited on 22/08/2020).

[28] D. Gupta, *25 Must Know Concepts for Beginners in Deep Learning & Neural Network*, May 2017. [Online]. Available: `https://www.analyticsvidhya.com/blog/2017/05/25-must-know-terms-concepts-for-beginners-in-deep-learning/` (visited on 22/08/2020).

[29] J. Zhang, *The Overview of Gradient Descent Algorithm*, zh-cn. [Online]. Available: `/2019/01/the-overview-of-gradient-descent-algorithm/` (visited on 22/08/2020).

[30] *TensorFlow*, en. [Online]. Available: `https://www.tensorflow.org/` (visited on 22/08/2020).

[31] *Matplotlib: Python plotting — Matplotlib 3.3.1 documentation*. [Online]. Available: `https://matplotlib.org/` (visited on 30/08/2020).

[32] *Pandas - Python Data Analysis Library*. [Online]. Available: `https://pandas.pydata.org/` (visited on 30/08/2020).

[33] *Scikit-learn: Machine learning in Python — scikit-learn 0.23.2 documentation*. [Online]. Available: `https://scikit-learn.org/stable/` (visited on 30/08/2020).

[34] *Glob — Unix style pathname pattern expansion — Python 3.8.5 documentation*. [Online]. Available: `https://docs.python.org/3/library/glob.html` (visited on 30/08/2020).

[35] *Pickle — Python object serialization — Python 3.8.5 documentation*. [Online]. Available: `https://docs.python.org/3/library/pickle.html` (visited on 30/08/2020).

[36] N. Rastogi, *Lenet-traffic-sign-classifier*, original-date: 2019-07-30T22:58:34Z, Jul. 2020. [Online]. Available: `https://github.com/neelrast/lenet-traffic-sign-classifier` (visited on 18/08/2020).

[37] P. Sermanet and Y. LeCun, 'Traffic sign recognition with multi-scale Convolutional Networks', in *The 2011 International Joint Conference on Neural Networks*, ISSN: 2161-4407, Jul. 2011, pp. 2809–2813. DOI: `10.1109/IJCNN.2011.6033589`.

[38] *Zero to Hero: Guide to Object Detection using Deep Learning: Faster R-CNN,YOLO,SSD*, en-US, Dec. 2017. [Online]. Available: `https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/` (visited on 30/08/2020).

[39] N. Dalal and B. Triggs, 'Histograms of Oriented Gradients for Human Detection', in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, San Diego, CA, USA: IEEE, 2005, pp. 886–893, ISBN: 9780769523729. DOI: `10.1109/CVPR.2005.177`. [Online]. Available: `http://ieeexplore.ieee.org/document/1467360/` (visited on 01/09/2020).

[40] H. S. Chadha, *Udacity-CarND-Vehicle-Detection-and-Tracking*, original-date: 2018-03-12T12:27:52Z, Sep. 2020. [Online]. Available: `https://github.com/harveenchadha/Udacity-CarND-Vehicle-Detection-and-Tracking` (visited on 11/09/2020).

[41] A. Rosebrock, *Histogram of Oriented Gradients and Object Detection*, en-US, Nov. 2014. [Online]. Available: `https://www.pyimagesearch.com/2014/11/10/histogram-oriented-gradients-object-detection/` (visited on 11/09/2020).

[42] S. Ren, K. He, R. Girshick and J. Sun, 'Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks', *arXiv:1506.01497 [cs]*, Jan. 2016, arXiv: 1506.01497. [Online]. Available: `http://arxiv.org/abs/1506.01497` (visited on 30/08/2020).

[43] S. Ananth, *Faster R-CNN for object detection*, en, Jul. 2020. [Online]. Available: `https://towardsdatascience.com/faster-r-cnn-for-object-detection-a-technical-summary-474c5b857b46` (visited on 30/08/2020).

[44] J. Świeżewski, *YOLO Algorithm and YOLO Object Detection: An Introduction*, en-GB, May 2020. [Online]. Available: `https://appsilon.com/object-detection-yolo-algorithm/` (visited on 30/08/2020).

[45] A. Bochkovskiy, C.-Y. Wang and H.-Y. M. Liao, 'YOLOv4: Optimal Speed and Accuracy of Object Detection', *arXiv:2004.10934 [cs, eess]*, Apr. 2020, arXiv: 2004.10934. [Online]. Available: `http://arxiv.org/abs/2004.10934` (visited on 30/08/2020).

[46] J. Hui, *SSD object detection: Single Shot MultiBox Detector for real-time processing*, en, Dec. 2018. [Online]. Available: `https://medium.com/@jonathan_hui/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06` (visited on 30/08/2020).

List of Figures

List of Tables