University of Tübingen
Self-Driving Cars Lecture Notes

Prof. Dr.-Ing. Andreas Geiger

Winter Term 2021/22

Abstract

These lecture notes have been written collectively by the class of winter 2021/2022 and curated by the TAs of the self-driving cars lecture. If you find an error, please email the TA responsible for the section.

1 Introduction

1.1 Introduction

The dream of autonomous driving is almost as old as the automobile itself. Long before the invention of highly complex computer systems and precise cameras and sensors, people were fascinated by the idea that cars in the future could operate completely without the influence of a human driver. The idea was obvious, since driving a car appears at first glance to be a very simple activity. Unlike very specialized tasks such as recognizing certain diseases on X-rays or predicting stock prices, driving is an activity that almost any human can easily perform. In addition, there is a very small number of possible actions that have to be performed while driving, which are basically limited to accelerating, braking and steering.

However, it quickly becomes apparent that this challenge was much more complex than anticipated and that some visual or cognitive tasks are much more difficult to accomplish for a computer than for a human. These problems and possible approaches to solve them are the content of this course.

1.1.1 Why Self-Driving Cars?

But why should we even bother with this topic? What are the advantages if the majority of cars could operate without human intervention in the future? One obvious reason is, of course, an increase in convenience. It would no longer be necessary to concentrate on the road and traffic for the entire journey and one could use the time for other activities. But there are numerous other reasons that justify the goal of autonomous driving and research in this area.

In 2017 alone, more than 1.3 million people were killed in traffic accidents worldwide[1]. In most cases, the reasons for this are not technical failure but human error such as speeding, driving under the influence of drugs or alcohol and distraction. These accident factors could be significantly reduced by autonomous vehicles. In addition, driving behavior could be significantly improved and thus made safer through faster reaction times and communication between vehicles. In this way, it would be possible to prevent a large proportion of traffic accidents and associated injuries or fatalities.

In addition, self-driving vehicles could also improve the mobility of people with physical disabilities. In the U.S., 45% of people with disabilities still work, but operating a vehicle is often a challenge for them. Autonomous driving could help make everyday life easier for people with disabilities.

The new technology could also lead to improvements from an environmental point of view. More efficient driving, for example by avoiding unnecessary acceleration and braking, could reduce fuel consumption and thus emissions.

The way we currently use the available vehicles is highly inefficient. On average, a car is parked 95% of the time, wasting space and resources that could be used elsewhere. Autonomous vehicles could enable completely new car sharing and car pooling approaches. The number of cars could be reduced significantly and the existing cars could be used much more efficiently

[1] In this interactive map from the World Health Organization, numerous data about traffic accidents are presented: https://extranet.who.int/roadsafety/death-on-the-roads

Figure 1: Benefits of autonomous driving [19]

1.1.2 Challenges for Autonomous Driving

There are many reasons to push ahead with research on autonomous driving, and a lot of money and effort has already been invested to develop ever better and more intelligent driving systems (see Section 1.2). However, there have also been some setbacks in this development that have highlighted the challenges of autonomous driving. In 2018, for example, there was a fatal traffic accident when an autonomous vehicle from Uber collided with a pedestrian.

Perhaps we have underestimated the cognitive and visual abilities humans need for driving. In Section 1.1.1 we were confronted with the figure of 1.3 million road deaths worldwide, which is of course a tragically high number. However, if we calculate the performance of an average human driver, we get one fatality per 100 million miles, which is an error rate of 0.000001%. In the development of most computer systems, the target reliability is much lower. So this error rate is not easy to improve, especially when you consider some of the challenges that a self-driving car will face on the road. Poor visibility due to weather events such as snow or heavy rain and poor lighting at night poses problems even for modern cameras and sensors and can lead, for example, to traffic signs not being recognized or situations being incorrectly assessed. Unstructured roads or worn lane markings make it difficult to navigate the vehicle through traffic. Integrated machine learning models require an enormous amount of training data containing as many different scenarios as possible with which the car can be confronted. However, even with very large data sets, it is easy to encounter rare or previously unseen events where the model cannot make decisions. Especially human drivers or pedestrians often behave unpredictably or irrationally and many situations cannot be estimated with certainty.

In addition to the technical challenges for self-driving cars, completely new, ethical questions also arise. In road traffic, situations very often arise in which the driver has to make a decision in just a few seconds. For a human driver, it is usually not possible to assess the situation and make a rational decision in this short time. A computer, however, is able to gather and process information much faster. However, this raises the question of how the vehicle should behave in certain situations, especially in dangerous situations that could end fatally for some of the participants. For example, if a self-driving car is about to have a likely fatal collision with a pedestrian and still has the option to steer the vehicle off the roadway, but this would then result in the death of the occupants in the car, this presents a non-trivial ethic dilemma. People have been thinking about such situations for over a hundred years - for example, in the famous Trolley Problem of 1905 - but until now they have mostly been theoretical thought experiments. With the technical possibilities of autonomous driving, however, these scenarios could become reality and it must be decided how the car should behave in these situations. On the website moralmachine[2], several of these moral dilemmas are presented and users can choose between two options or even develop new scenarios themselves (See Fig. 2). This is an interesting way to think about different ethical choices and to get a feeling of how difficult these choices can be.

[2] https://www.moralmachine.net

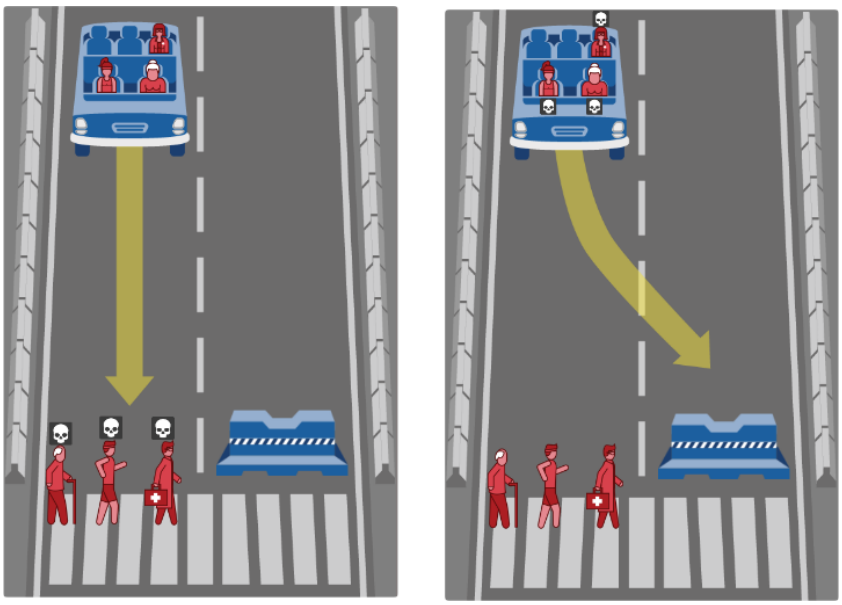


Figure 2: One of the possible moral dilemma on https://www.moralmachine.net

1.2 History of Self-Driving

In the last chapter, we looked at the reasons for the development of self-driving cars and some of the challenges that need to be addressed. Even if it seems that self-driving has only gained importance in recent years, there were already ambitions to automate driving early in the history of the automobile. In this chapter, we will look at this history of autonomous driving.

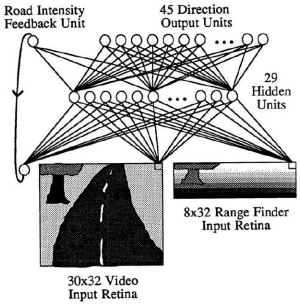
1.2.1 The Early Beginnings

When we look at the history of autonomous driving, it makes sense to start at the beginning of driving itself. In 1886, Karl Benz developed the first automobile with an internal combustion engine, the so-called "Benz Patent-Motorwagen Nummer 1" (See Fig. 3). Weighing 265 kilograms and consuming 10 liters per kilometer, this vehicle was able to reach a top speed of 16 kilometers per hour. This invention (which was initially viewed with great skepticism) laid the foundation for a large part of our mobility today and at the same time for research into autonomous driving.

Figure 3: Benz Patent-Motorwagen Nummer 1 - The first automobile with an internal combustion engine built by Carl Benz.

In the following decades, cars were driven exclusively by human drivers, but in 1925 the first steps towards autonomous driving were taken with the Phantom car - also called the "American Wonder". A driverless car, followed by an operator in another vehicle, was piloted through heavy traffic in New York City. However, since a human driver still had complete control over the car and was merely controlling it remotely, this invention was still far from what we think of as fully autonomous driving.

(a) Navlab 1



(b) ALVINN (Autonomous Land Vehicle in a Neural Network)

Figure 4: Two early vision based self driving approaches

The next milestone on the way to autonomous driving were cars that could move with the help of magnetic cables built into the road. Two examples of this approach were RCA Labs' Wire Controlled Car in 1960 and the Citroen DS19 in 1970. This technology was very different from today's approaches to autonomous driving because it required the necessary technology to be permanently integrated into the road instead of using visual and sensory technology to navigate.

1.2.2 Towards Real Autonomy

In the 1980s, the development of autonomous vehicles began, which did not depend on cables or a certain infrastructure but functioned with vision-based navigation. "Navlab" (Fig. 70a), a project of Carnegie Mellon University, was one of the first vehicles to use this technology. Between 1986 and 1995 this vehicle was further developed. With the Navlab 5 (1995) it was possible to drive with an autonomy of 98% 2850 miles across the USA. Another example of the development of vision-based driving was ALVINN (Autonomous Land Vehicle in a Neural Network) in 1988, which used a fully connected neural network with two layers to convert road images into vehicle commands (Fig. 70b). ALVINN was capable of driving 90 miles at a time autonomously, reaching top speeds of 70 miles per hour. In 1986, a similar system was developed by the German Bundeswehr and was called "VaMoRs". It enabled longitudinal and lateral guidance with lateral acceleration feedback and could speed up to 36 km/h.

1.2.3 First Technological Revolution and Darpa Challenge

While the development of autonomous vehicles gradually progressed and the systems developed became better and better, three technological revolutions were of particular importance and led to enormous improvements in this field of research. The first of these technological revolutions was the development of GPS and IMU around the year 2000. Through the use of GPS systems, the position of a vehicle could be determined worldwide to within one meter. Since this is still not sufficient to navigate the car reliably in road traffic, so-called Inertial Measurement Units (IMUs) were used. These devices measure the acceleration forces in a vehicle and make it possible to improve the accuracy of GPS navigation to up to 5 cm. Combined with digital road maps, it was now possible to locate each vehicle very precisely and navigate through traffic.

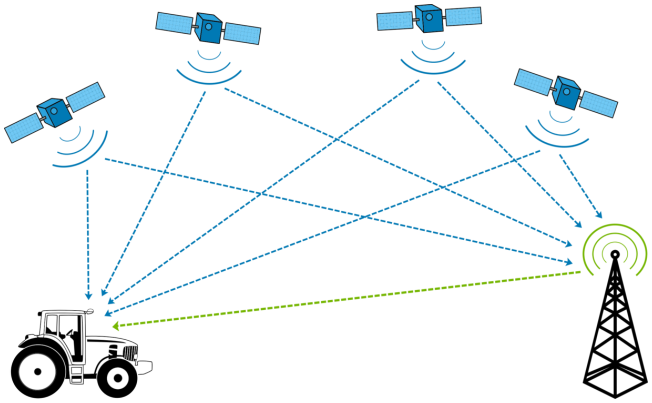


Figure 5: GPS system to navigate a vehicle.

In many research areas, large competitions and challenges were important factors for new innovations. For example, the ImageNet Challenge annually produced many new approaches in image recognition and generated a lot of attention and research funding in this area. The equivalent of the ImageNet Challenge in the field of self-driving is the Darpa Grand Challenge. The first competition was launched in 2004. Self-driving cars had to navigate a 240km route through the Mojave Desert and a prize of one million dollars was offered for the first place. However, none of the participating teams managed to finish the course. Only one year later, the second Darapa Grand Challenge was held with a prize money of 2 million dollars. This time, five teams managed to drive the 212 km route autonomously to the end.

1.2.4 Second Technological Revolution and Big Companies

Visually based self-driving cars became popular, but their performance was largely dependent on the quality of the cameras and sensors used. Around 2006, high-resolution lidar (light detection and ranging imaging) systems were increasingly developed. The resolution of camera systems was also significantly improved. In addition, it was now possible to create precise 3D reconstructions of the surroundings and to recognise and locate obstacles and other cars in these 3D images (See Fig. 6). These improved technologies made vision-based self-driving cars even more powerful and significantly advanced research.

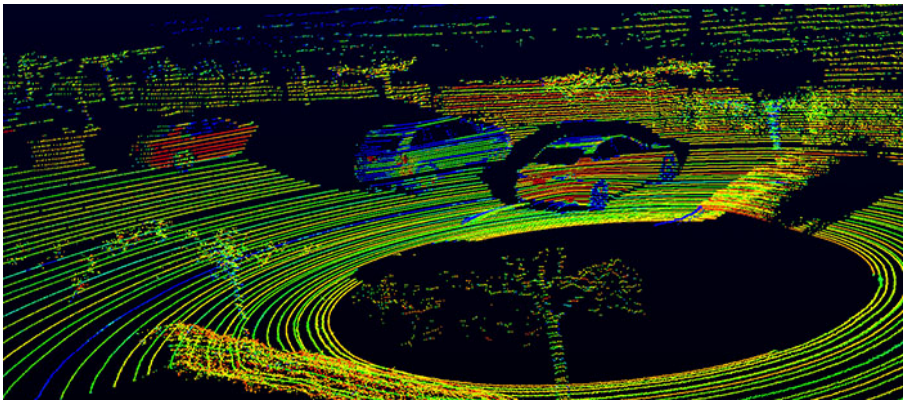


Figure 6: Improved Cameras and Lidar systems made it possible to create accurate 3D reconstructions of the surroundings.

As the development of autonomous driving continued, large tech companies became increasingly aware of the technology and started their own research in this area. In 2009, for example, Google began working on autonomous driving and developed a prototype that was later named "Waymo" (Fig. 7). By 2015, Google had invested $1 billion in development and research in the field of self-driving.

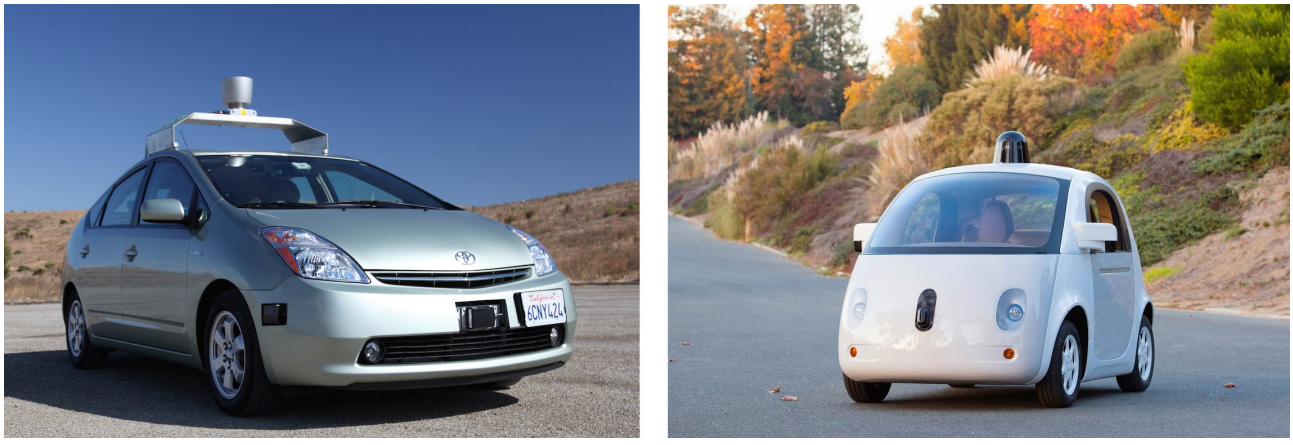


Figure 7: Google's self driving car "Waymo"

In 2010, another milestone was set. In the VisLab Intercontinental Autonomous Challenge (VIAC), an autonomous vehicle drove over 16,000 kilometres from Parma, Italy to Shanghai, China. Here, the self-driving car autonomously followed a guide car driving ahead.

Increasingly, self-driving cars were also tested in small, geofenced areas where they had to operate in real traffic. In contrast to the first Darpa Challenge, for example, in which the participants drove on an empty desert road, challenges such as traffic lights, roundabouts, other road users and pedestrians were added. An example of these projects is the "Stadtpilot" developed by the Technical University of Braunschweig in 2010 (Fig. 9). Similar projects were also carried out by the FU Berlin and other teams.

Figure 8: "Stadtpilot" developed by Technical University Braunschweig

1.2.5 Third Technological Revolution

Around 2012, the third technological revolution occurred, which significantly advanced the development of self-driving cars. Representation learning and deep learning revolutionised many different areas of research and also brought a significant advantage in the accuracy of self-driving systems. In addition, new benchmarks made it possible to compare different technologies and systems in a uniform manner. Similar to the various challenges, such benchmarks provide an incentive for many research teams to keep developing new, improved approaches.

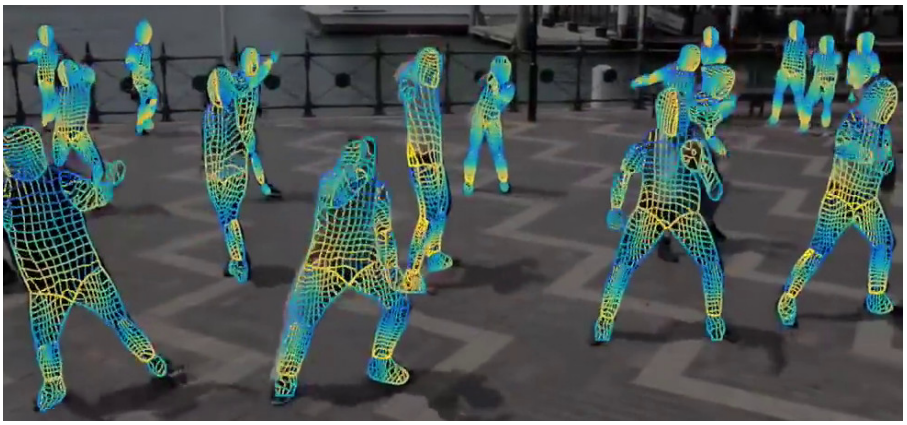


Figure 9: "Dense Human Pose Estimation In The Wild" - One example of new possibilities by using Deep Learning.

The new technological possibilities led to the development of better and better autonomous vehicles, such as the Mercedes S Class in 2014, which was equipped with features such as autonomous steering, lane keeping, autonomous acceleration and braking as well as collision avoidance and had a top speed of up to 200 km/h. The Mercedes S Class achieved a Level 2 Autonomy (Section 1.2.6).

1.2.6 Levels of Autonomy

In 2014, the Society of Automotive Engineers (SAE) published a taxonomy comparing different self-driving technologies and categorising them into five different levels of autonomy. A level 0 system (Driver Only) has no autonomous elements at all and the driver must operate the car himself the entire time. In a Level 1 system (Assisted), either lateral or longitudinal actions are carried out partially autonomously by the vehicle. In a Level 2 system (Partial Automation), both lateral and longitudinal actions are performed by the car itself. The driver can temporarily take his hands off the steering wheel, but must always remain alert and be able to intervene at any time. A Level 3 system (Conditional Automation) has similar capabilities. In addition, however, the

performance of the system is determined at all times, so that it is recognised at an early stage if the computer cannot fully assess a situation. The driver is then warned in time and must take control in these cases. A Level 4 system (High Automation) is even more autonomous. It can assess all situations within a certain use case and react independently. The system is called "Eyes Off" and "Hands Off" because the driver does not have to be attentive or intervene within this use case. However, in situations not covered by the defined use case, the driver still has to actively operate the car. The last and highest level of the scale is Level 5 (Full Automation). As the name suggests, the system is able to act and react completely autonomously in all possible situations. A human driver is no longer required and the occupants of the car do not have to intervene in the system at any time.

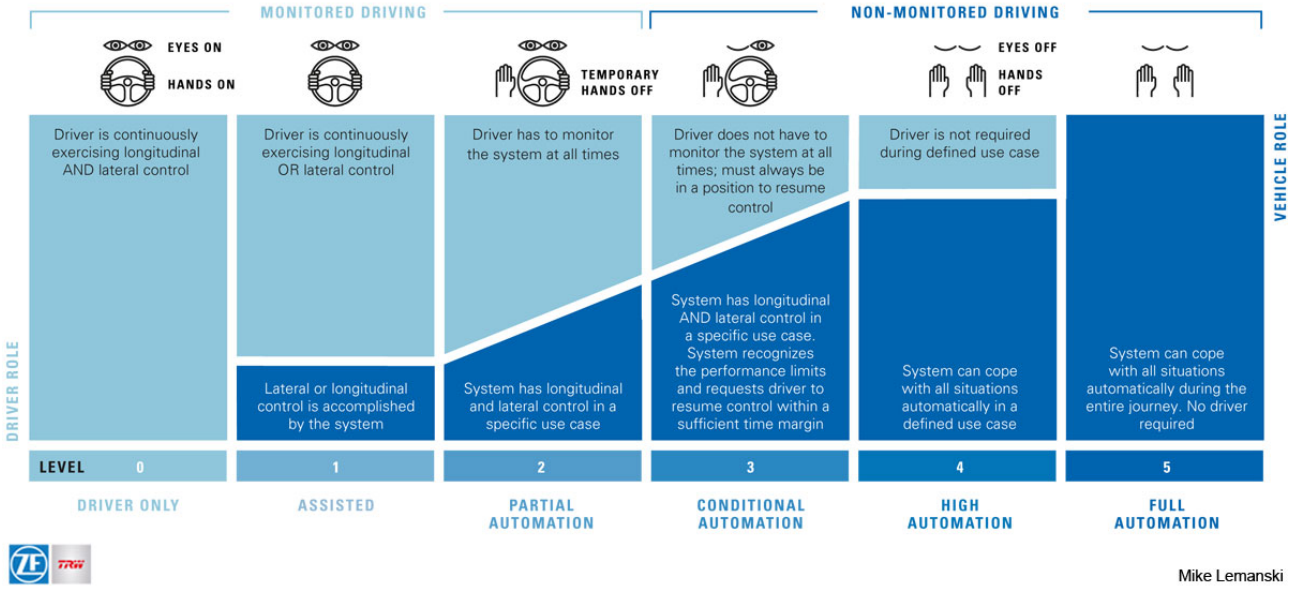


Figure 10: The five levels of autonomy published by the Society of Automotive Engineers (SAE)

1.2.7 Self Driving Industry

The success and progress in the field of autonomous driving in the past years became more and more known and led to the fact that more and more companies started to develop certain components or complete systems for autonomous driving and gradually a complete self-driving industry was formed. Companies like Uber, Otto and Tesla started their own research departments in 2015 and 2016 and hired numerous scientists and engineers. But the development was also marked by setbacks again and again. Two fatal accidents with Tesla Autopilot vehicles occurred, one in 2016 and one in 2018. Accidents like these lowered trust in the technology and repeatedly raised legal questions. Nevertheless, self-driving technology continues to evolve and attract more and more companies.

In most cases, two different business models are being pursued. Large tech companies such as Google, Apple or Uber are pursuing an "autonomous or nothing" strategy that aims to develop fully autonomous vehicles directly and focus primarily on long-term goals. However, this strategy is very risky and costly, which is why only a few companies can pursue it. Most other companies rather follow a model where small technical innovations are introduced and the cars become more and more autonomous step by step. The challenge is to ensure that the driver remains engaged and does not completely rely on the self-driving car while it is still not able to cope with all situations.

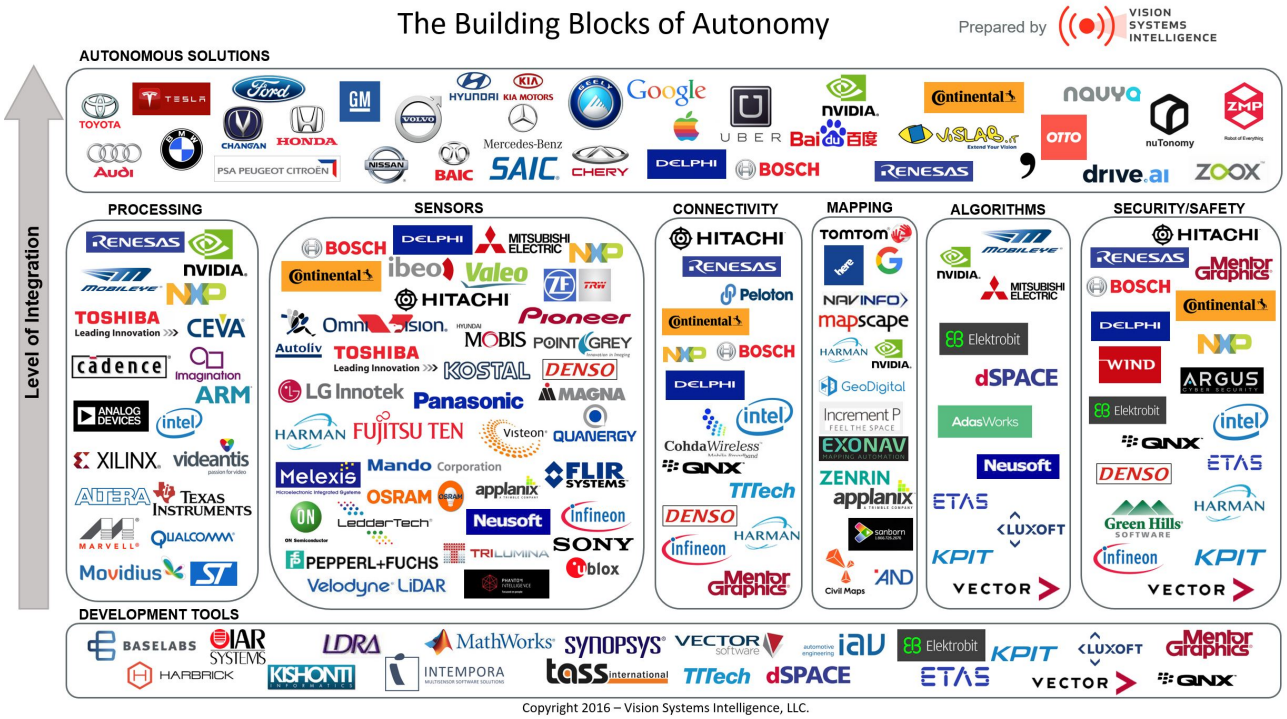


Figure 11: The self-driving industry

1.2.8 Summary

As we have seen, self-driving has a long history and people started thinking about ways to make vehicles autonomous earlier than one might think. Over the past decades, this technology has continued to improve and become more and more reliable, until in 2019, the first vehicle was presented that realizes a level 3 autonomy for highways. In the race for more and more autonomous vehicles, there are now many players, including well-known ones, and Waymo and Tesla seem to be leading the field. But Level 4 and 5 autonomy remain unattained and extremely difficult to implement. In addition, several setbacks have lowered confidence in self-driving and brought ethical and legal issues to the fore. Self-driving is a complex but promising technology and it is expected to become more and more important in the coming decades.

2 Imitation Learning

2.1 Approaches to Self-Driving

An autonomous driving vehicle can sense its environment and can drive safely without human input. A simple autonomous driving stack maps high dimensional sensory input data via a mapping function to low dimensional actions such as steering or gas and braking. The input data can be collected from different kinds of sensors like LiDAR, radar or video captures. The goal in autonomous driving is to find a robust mapping function that maps the input reliably to the output. This section describes the three dominant paradigms for this problem: **Modular Pipelines**, **End-to-End Learning** and **Direct Perception**.

2.1.1 Modular Pipelines

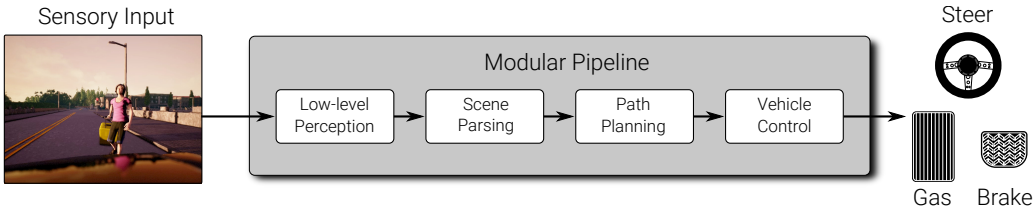


Figure 12: **Modular Pipeline**

A modular pipeline decomposes the entire self-driving stack into independent components. This approach is among others used by the winners of the DARPA Challenge [20, 26] and the industry leaders like Waymo, Uber, Tesla and Zoox. The modular pipeline paradigm allows for easy parallel development due to the small independent components. Furthermore, a modular pipeline is better interpretable since the interfaces of the modules are well known and specified. However, such an approach doesn't allow End-to-End training which means each module is trained individually. This leads to modules learning behaviour that will not benefit the downstream task of driving. Additionally, localization and planning relay heavily on HD-maps, highly detailed maps which are accurate down to the centimeter level, containing details like road lanes, road markings, traffic signs, and barriers. These maps require a lot of human labor to create and maintain up to date.
Fig. 12 shows a particular example of a modular pipeline.

1. At first a Low-level Perception module is responsible for localizing the car and detecting buildings, cars and driving lanes.
2. In the second stage the perceived information must be consolidated into one robust scene representation in the Scene Parsing module.
3. A Path Planning module can use this representation to plan a path on the local lane- as well as the global-level.
4. Once a path has been planed it will be executed by the Vehicle Controller such that the distance and speed will be kept close to the trajectory determined by the Path Planning stage.

It is to note that in practice modular pipelines end up larger and more complex as the shown example.

2.1.2 End-to-End Learning

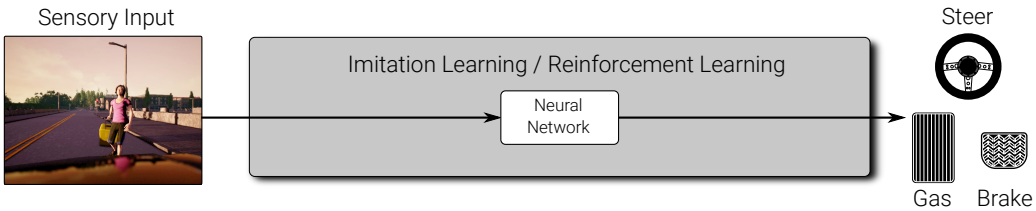


Figure 13: **End-to-End Learning**

An alternative approach to the Modular Pipeline is End-to-End Learning. In End-to-End Learning the mapping from sensory input data to driving commands is solved in one step. Typically, this task is performed by a neural network. The neural network can be trained using different algorithms, the two most important

ones being imitation learning and reinforcement learning. This approach was used early on by researchers like Pomerleau et al. and their ALVINN-vehicle [22]. The advantages of this paradigm are that the model is optimized directly for driving. Furthermore data annotations are cheap, since a camera can be attached to a car and sensors to the steering mechanisms to collect data automatically. On the other hand these models do not generalize as well as Modular Pipelines and are hard to interpret.

2.1.3 Direct Perception

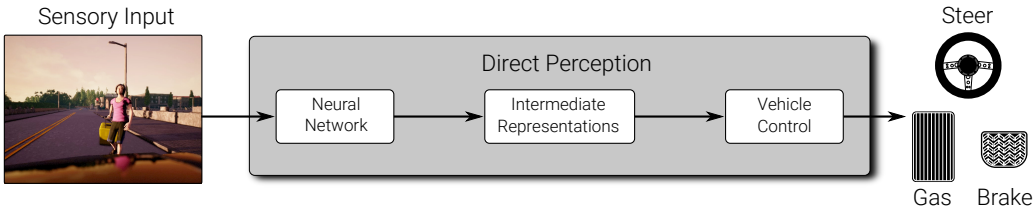


Figure 14: **Direct Perception**

Direct Perception represents a hybrid between Modular Pipelines and End-to-End Learning where an intermediate representation e.g. a semantic segmentation is created by a neural network. This representation is then processed by a second neural network or classical vehicle controller to produce the vehicle control. This approach is among others used by Chen et al. [8], who considered highway driving and used as an intermediate representation a set of distance measures to other cars and lane marks. These distances are then used by the vehicle controller to decide on steering commands. The advantages of Direct Perception are, that the representation can be very compact and interpretable. Unfortunately, it is not always easy to train the vehicle controller jointly. Furthermore, a suitable representation has to be found, introducing a human decision and error into the model.

2.2 Deep Learning

This section will provide a short recapitulation on Deep Learning since it is a prerequisite for this lecture. For a deeper dive into these topics the Deep Learning lectures from Prof. Andreas Geiger on https://youtu.be/OCHbm88xUGU are recommended.
Supervised Learning, of which Imitation Learning is a subset, estimates a vector of parameters $\mathbf{w}$ from a training data set $(x_i, y_i)_{i=1}^N$. Subsequently, such a trained model $f_\mathbf{w}$ can be used to make novel predictions $y = f_\mathbf{w}(x)$ which is also known as inference. The input data x can have many forms e.g. an image and the output y could be a vehicle control in the case of self-driving cars.

2.2.1 Classification

Classification refers to the task of analyzing a set of pre-classified data to learn a model that can be used to classify unseen data into one of several predefined classes. The simplest linear classifier is the logistic function σ that takes as an input a numerical 2-dimensional input vector $x \in \mathbb{R}^2$ and produces an output $y \in (0, 1)$.

$$\hat{y} = \sigma\left(\mathbf{w}^\top \mathbf{x} + w_0\right) \quad \sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

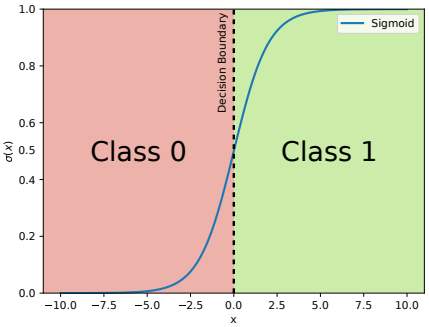
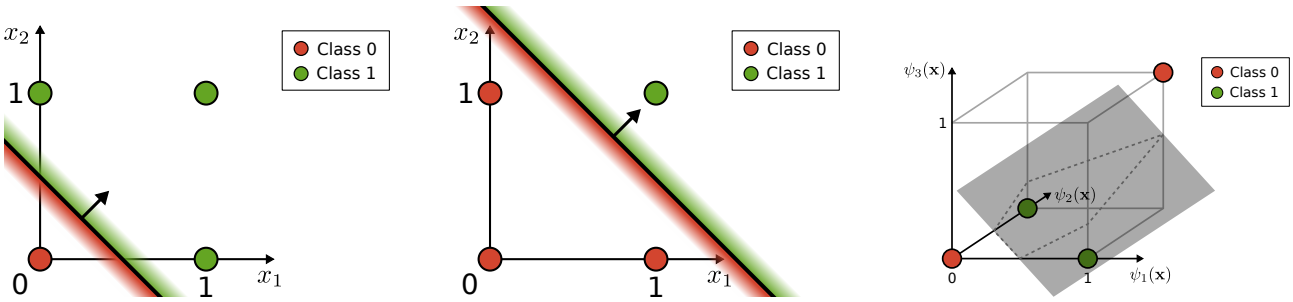


Figure 15: **Logistic Regression** with a plotted sigmoid and a decision boundary for $\omega_0 = 0$ at 0.0

By using a decision boundary (see Fig. 15) at $\mathbf{w}^\top \mathbf{x} + \omega_0 = 0$, all inputs x for which $\mathbf{w}^\top \mathbf{x} > -\omega_0$ are assigned to



(a) OR (b) AND (c) XOR

Figure 16: **Logical functions classifier** to demonstrate how a linear classifier can can find perfect decision boundaries to non linear data

class **1** and otherwise to class **0**. This simple classifier can solve the logical OR and AND operator for the two dimensional inputs x_1 and x_2. For the OR operation an $-\omega_0 > 0.5$ can be chosen to define a decision boundary (see Fig. 16a). Similarly, a decision boundary for the AND operation is given by $-\omega_0 > 0.5$. However for the XOR operator no single decision boundary can classify the two classes correctly. This problem can be solved by lifting the 2 dimensional feature space to higher dimensions. For example by considering the product of x_1 and x_2

$$\mathbf{w}^\top \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{pmatrix}}_{\psi(\mathbf{x})} > -w_0 \tag{2}$$

as illustrated in Figure Fig. 16c. In these 3 dimension a linear decision boundary can be found that separates the two classes. In other words, non linear features allow linear classifiers to solve non-linear classification problems.
The right choice of representation, like the choice between polar and cartesian coordinate systems, can also simplify the separation of data. Finding the right representation had to be done manually until the 2000s. Nowadays the optimal representation can be learned by a deep neural network from the data itself.

In order to linearly classify the XOR operator one can use a combination of OR and NAND decision boundaries like in Figure 17b and expressed in equation (3)

$$\begin{aligned} &\text{XOR}\left(x_1, x_2\right) = \\ &\text{AND}\left(\text{OR}\left(x_1, x_2\right), \text{NAND}\left(x_1, x_2\right)\right) \end{aligned} \tag{3}$$

It is easy to show that the XOR operation of x_1 and x_2 is equal to the AND operation of the OR and the NAND operation of the two arguments.

Equation (3) can also be written as a program of logistic regressors

$$\begin{aligned} h_1 &= \sigma\left(\mathbf{w}_{OR}^\top \mathbf{x} + w_{OR}\right) \\ h_2 &= \sigma\left(\mathbf{w}_{NAND}^\top \mathbf{x} + w_{NAND}\right) \\ \hat{y} &= \sigma\left(\mathbf{w}_{AND}^\top \mathbf{h} + w_{AND}\right) \end{aligned} \tag{4}$$

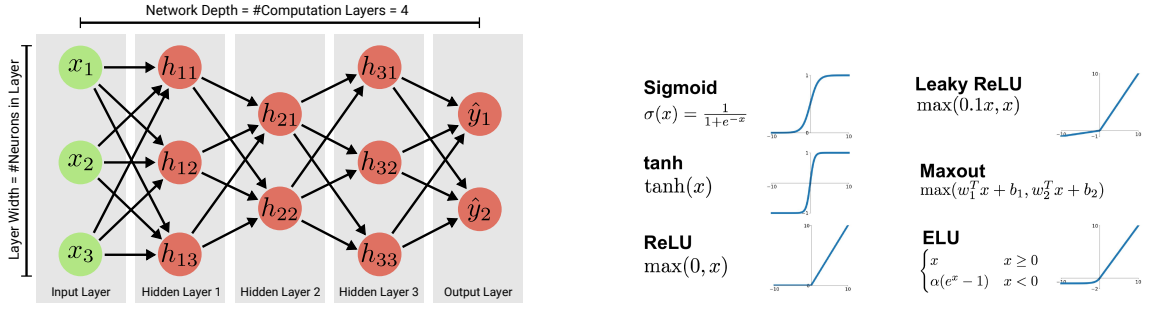
where a hidden feature value h_1 and h_2 is obtained from the OR and NAND operation. The output of h_1 and h_2 are transformed with the AND operator to classify x_1 and x_2 regarding to XOR. This program is visualized in Fig. 17a. Note that $h(x)$ is a non-linear feature of x due to the logistic function σ. $h(x)$ is also called a hidden layer.

2.2.2 Multi-Layer Perceptrons (MLP)

MLPs are feedforward neural networks. They have no feedback connections such as recurrent neural networks do. They represent a mapping function we already described above (here with respect to Fig. 18a): $f_{\mathbf{w}}(\mathbf{x}) = f(\mathbf{x}) = \hat{\mathbf{y}}(\mathbf{h}_3(\mathbf{h}_2(\mathbf{h}_1(\mathbf{x}))))$ with input data x, composing several non-linear functions, where $\mathbf{h}_i(.)$ and $\hat{\mathbf{y}}_i(.)$ are called hidden and output layer, respectively.
Each layer comprises multiple neurons, implemented as affine transformations $(\mathbf{A}_i * \mathbf{x} + \mathbf{b}_i)$ followed by non-linear activation funtions g: $\mathbf{h}_i = g(\mathbf{A}_i * \mathbf{h}_{i-1} + \mathbf{b}_i)$, where $\mathbf{A}_i$ and $\mathbf{b}_i$ are learnable weights. Note, that the activation function is applied element-wise and must be non-linear (e.g. ReLU, Leaky ReLU, tanh, sigmoid, maxout, ELU) to learn non-linear mappings (see Fig. 18b). Each neuron in each layer is fully connected to all neurons in previous layer (see "→" in Fig. 18a). The overall length of the chain (#(hidden layers) + 1) is the

(a) Decision boundary visualization (b) Perceptron program

Figure 17: **Logical XOR classifier** with multiple perceptrons



(a) MLP (b) Activation functions

Figure 18: Network architecture of a MLP and different activation functions

depth of the model. Hence the name deep learning. The number of neurons in each layer denotes the layer width. The wider and deeper the model, the more complex problems can be solved.

Output and Loss Function

The hidden representation of the data $\mathbf{x}$, calculated through numerous applications of affine transformations and non-linearities, is then used to compute the output $\hat{\mathbf{y}}$. For regression, the output of the last hidden layer is directly returned, for classification a sigmoid or softmax non-linear function is applied to get the probabilities of the single classes. For training, a loss function $\mathcal{L}$ is then used to compare ground truth $\mathbf{y}$ and prediction of the network $\hat{\mathbf{y}}$. Most commonly, for regression l_1 or l_2 loss is used. For classification (binary) cross-entropy loss.

Activation Functions

Each hidden layer has a non-linear activation function $g(\cdot)$, which is applied most often in an element-wise fashion to its inputs. These functions must be non-linear to enable the network to learn non-linear mappings overall. Popular activation functions can be seen in Fig. 18b. Often the optimal activation function needs to be found empirically.

2.2.3 Convolutional Neural Networks

Convolutional neural networks (ConvNets/ CNNs) are particularly useful for processing images and as such important for self-driving. Multi Layered Perceptrons do not scale well to high dimensional inputs. For example, a megapixel input image needs a lot of connections and weights which are very hard to learn and require a lot of memory to store. The key ideas of ConvNets are to exploit only sparse interactions, to use heavily parameter sharing. That way the number of parameters can be decreased significantly.

Network Architecture

CNNs are representing data in 3 dimensions (width, height, depth). In Fig. 19 you can see an example of a CNN used to classify images into certain categories. It consists of convolutions, non-linearities as well as pooling operations. For classification purposes, a fully connected layer and a softmax activation function are added. Those components will be explained in more detail below.

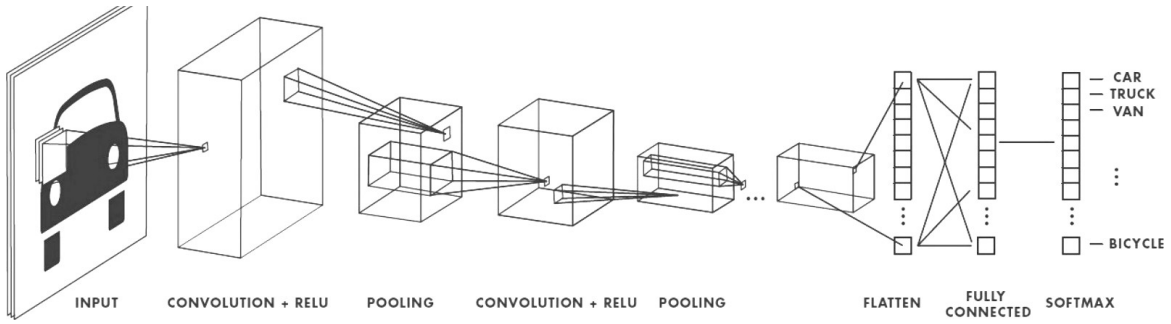


Figure 19: **CNN Architecture.** An illustration of a convolutional neural network in a classification setting.

Convolutions. The main component of these networks are called convolutional operations. Those are using kernels, also called filters, to create feature maps - a feature representation of the image. The convolution kernel slides along the input matrix and applies the convolution operation reducing the input to its essential features (feature maps). During training, the network determines what features are important for solving the given problem, for example, to categorize an image.

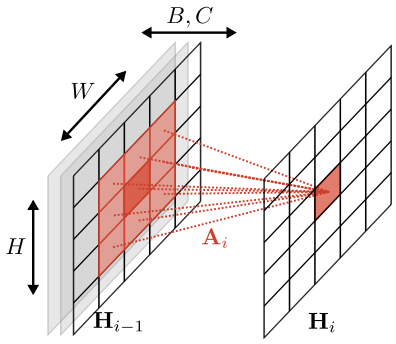


Figure 20: **Convolutional Operation.** An illustration of the convolutional operation in a CNN.

Fig. 20 shows a convolutional operation A_i for layer i. We have a 3 dimensional input $\mathbf{H}_{i-1}$ (height H, width W, depth B) and a 3 dimensional kernel (height K, width K, depth C) where $C = B$. The kernel of size 3 consists of learnable parameters. For better understanding, using an input of depth 1 (e.g. grayscale image), 9 pixels of the input is used to calculate one pixel of the feature map $\mathbf{H}_i$. If padding (explained below) is not applied, this leads to downsampling.

Downsampling reduces the spatial dimension of the input. This means the height and width are reduced after the convolutional operation. This equivalently leads to an increased respective field defined as the region in the input that a particular pixel in the output is able to observe. By using multiple convolutional operations in each layer, the broadened receptive field means that bigger structures can be abstracted in the later feature representations of the network. The usage of convolutional layers in a convolutional neural network therefore mirrors the structure of the human visual cortex, where a series of layers process an incoming image and identify progressively more complex features.

Padding. Downsampling can be avoided by using padding. This adds a boundary of an appropriate size with zeros around the input. By doing so, the height and width of the input will not change after applying the convolutional operation.
Fig. 21 shows the use of padding of the input prior to applying the filter in the convolutional operation. The padding operation is often set as "valid" or "same" padding where "valid" padding actually applies no padding at all, meaning that the feature maps shrink by $k-1$ pixels after each convolution with a $k \times k$ kernel. "Same" padding means that the feature maps keep the same size (as seen in Fig. 21).

Pooling also reduces the spatial dimensions of the input. It has no parameters and uses typically max, min or avg - pooling. Modern architectures are rather using strided convolutions instead of pooling operations. By using pooling or strided convolutions the receptive field size grows faster than only using convolutional operations.

Last Layer (fully connected). By simply using convolutions, pooling operations and non-linearities, a network learns to distinguish certain structures in the data. However, just having a different representation of

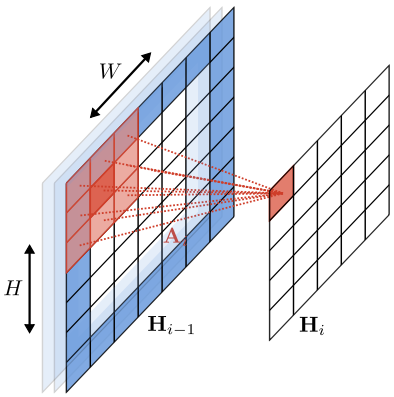


Figure 21: **Padding.** An illustration of using padding of the input prior the convolutional operation.

the initial input is not enough. The final task is to solve a problem, for example, classifying images (see Fig. 19). This is typically done by flattening the last feature maps and using a fully connected dense layer. This often increases the number of parameters. To mitigate this effect, all pixels within each feature map are averaged prior to flattening using global average pooling.

Difference to Fully Connected Networks. As seen in Fig. 22, a fully connected network connects all inputs and each weight is used exactly once. A CNN is utilizing sparse interactions, parameter sharing and is therefore able to have an equivariant representation.

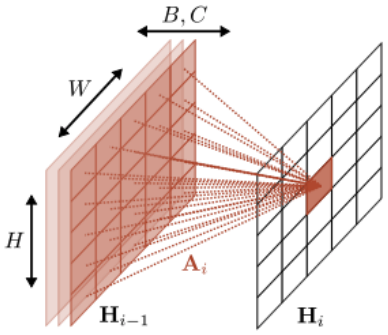


Figure 22: **Fully Connected Network.** Connection of inputs in a fully connected network.

Sparse Interactions. is implemented by using kernels or feature detectors smaller than the input image. Thus, a CNN is "sparse" as only the local "patch" of pixels is connected instead of using all pixels as an input (see Fig. 22 in comparison to Fig. 20). Computing the output requires fewer operations, making it more efficient.

Parameter/Weight Sharing. The key advantage of CNNs is weight/parameter sharing. If one feature detector is useful to compute one spatial position then it can be used to compute a different one. As the parameters are shared, it reduces the number of parameters to be learnt and hence the computational needs. The number of parameters/weights in a convolutional layer can be calculated as follows:

$$\#Weights = C_{out} \times (K \times K \times C_{in} + 1) \tag{5}$$

where $\#Weights$ are the number of parameters/weights, K the height and width of the kernel (filter), C_{out} the number of filters applied (the resulting number of feature maps (depth of the output)) and C_{in} the depth of the input. There is also a bias term for each feature map. In contrast, the number of parameters/weights of a fully connected layer would be as follows:

$$\#Weights = W \times H \times C_{out} \times (W \times H \times C_{in} + 1) \tag{6}$$

where $\#Weights$ are the number of parameters/weights, W and H the width and height of the input, C_{in} the depth of the input and C_{out} the number of neurons in the hidden layer. There is also a bias term for each connection to the hidden layer.

Equivariant Representation. Natural images have many statistical properties that are invariant to translation: A photo of a cat remains a photo of a cat if it is translated one pixel to the right. A CNN is able to take this into account. A shift in the input also translates the computed feature maps. That means we can find a cat with some "cat detector" independent of where the cat appears in the image.

2.2.4 Optimization

In order to optimize neural networks, gradient based optimization techniques are needed since the loss function of the highly non linear models is not convex. We want to find the optimal parameters $\mathbf{w}^*$ that minimize the loss $\mathcal{L}$ over the set $\mathcal{X}$.

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \mathcal{L}(\mathcal{X}, \mathbf{w}) \tag{7}$$

These optimal parameters can be estimated with a gradient update loop

$$\begin{aligned} \mathbf{w}^0 &= \mathbf{w}^{\text{init}} \\ \mathbf{w}^{t+1} &= \mathbf{w}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}\left(\mathcal{X}, \mathbf{w}^t\right) \end{aligned} \tag{8}$$

where η is the learning rate.
The gradients of the loss function with respect to the parameters $\mathbf{w}$ can be efficiently computed with Backpropagation. Therefore the activation values are computed by forward traversing the computation graph. Then the derivatives are calculated by tracing the computation graph backwards while utilizing the stored activation values and the chain rule.

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathcal{X}, \mathbf{w}) = \sum_{i=1}^{N} \underbrace{\nabla_{\mathbf{w}} \mathcal{L}\left(\mathbf{y}_i, \mathbf{x}_i, \mathbf{w}\right)}_{\text{Backpropagation}} \tag{9}$$

Typically, a model has more than 1 million parameters $\dim(\mathbf{w}) \geq 1$ million and needs to be trained on millions of training points $N \geq 1$ million. Therefore the computation becomes extremely expensive and might not fit into memory.

Stochastic Gradient Descent. A solution to this issue is stochastic gradient descent. The idea is to express the total loss over the whole training set as an expectation value

$$\frac{1}{N} \sum_i \mathcal{L}_i\left(\mathbf{w}^t\right) = \mathbb{E}_{i \sim \mathcal{U}\{1,N\}}\left[\mathcal{L}_i\left(\mathbf{w}^t\right)\right] \tag{10}$$

where N is the size of the training set and i indicates the training element. This expectation can be approximated by a smaller subset $B \ll N$. B is known as the minibatch size.

$$\mathbb{E}_{i \sim \mathcal{U}\{1,N\}}\left[\mathcal{L}_i\left(\mathbf{w}^t\right)\right] \approx \frac{1}{B} \sum_b \mathcal{L}_b\left(\mathbf{w}^t\right) \tag{11}$$

Thus the gradient can also be approximated by the minibatch gradient.

$$\frac{1}{N} \sum_i \nabla_{\mathbf{w}} \mathcal{L}_i\left(\mathbf{w}^t\right) \approx \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b\left(\mathbf{w}^t\right) \tag{12}$$

The stochastic gradient algorithm is given by:

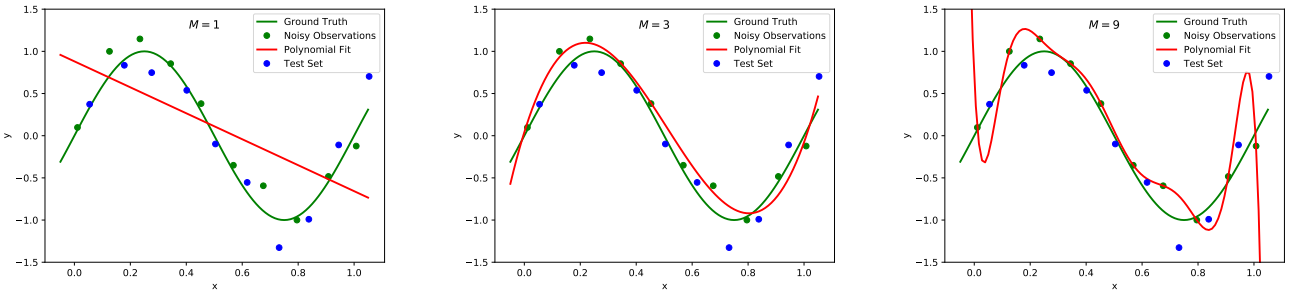
1. Initialize weights $\mathbf{w}^0$, pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw random minibatch $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_B, \mathbf{y}_B)\} \subseteq \mathcal{X}$ (with $B \ll N$)
3. For all minibatch elements $b \in \{1, \ldots, B\}$ do:
   (a) Forward propagate $\mathbf{x}_b$ through network to calculate prediction $\hat{\mathbf{y}}_b$
   (b) Backpropagate to obtain batch element gradient $\nabla_{\mathbf{w}} \mathcal{L}_b\left(\mathbf{w}^t\right) \equiv \nabla_{\mathbf{w}} \mathcal{L}\left(\hat{\mathbf{y}}_b, \mathbf{y}_b, \mathbf{w}^t\right)$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b\left(\mathbf{w}^t\right)$
5. If validation error decreases, go to step 2 , otherwise stop

Many variants of gradient descent exists. One of the most common choices today is Adam which combines momentum and gradient scaling. It is also important to set the learning rate appropriately. A fixed learning rate is typically too slow in the beginning and to fast at the end. Therefore the learning rate will usually be decreased during training, e.g. by an exponential decay $\eta_t = \eta \alpha^t$ or a by decreasing the rate every K iterations $\eta \leftarrow 0.5\eta$

2.2.5 Regularization

Before diving into regularization, model capacities are explained to see when regularization is applied.

Model Capacity
Capacity can be classified into three groups (see Fig. 23).



(a) Capacity to low (b) Capacity about right (c) Capacity to high

Figure 23: **Different Model Capacities.** Showing model capacities on a regression example.

On the left, the model underfits and is not able to achieve a low training error. On the right the model overfits, meaning the generalization error on the test set is very high. In the middle, the model capacity is just right (generalizing well).

Regularization Techniques
To make a model generalize better (right → middle), i.e., minimize its error on a separate test dataset, various regularization-techniques can be applied.

Early Stopping and Parameter Penalties. In early stopping, one monitors the validation loss or accuracy (separate validation set besides training set) and stops the training when the validation loss/accuracy is no longer improving. This can be seen on the left plot in Fig. 24. Here you can see the trajectory taken by the SGD algorithm (dashed line). The optimal set of parameters (w_1 and w_2) for the training set is w^*. However, this set is not optimal for the validation set. The optimal set of parameters for the validation is $\widetilde{w}$.

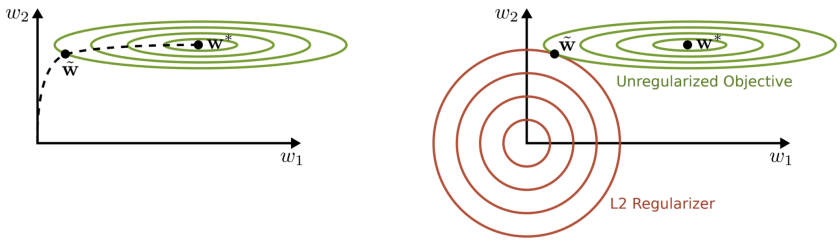


Figure 24: **Early Stopping / L2 Regularization.** Early Stopping (left) and L2 regularizer (right).

The L2-Regularization (right plot in Fig. 24) adds a L_2 penalty term to the loss function $\mathcal{L}$, preventing weights from getting too big in value. This shifts the optimal set from w^* to $\widetilde{w}$.

Dropout. Dropout essentially adds noise to the training process and is a computationally inexpensive but powerful way method of regularization. Dropout approximates training a large number of neural networks with different architectures in parallel (ensemble). Each update to a layer during training is performed with a different view of the configured layer.
An example model architecture can be seen on the left (Fig. 25). During training, neurons can be turned of with probability μ (right plot), typically $\mu = 0.5$. Therefore in each training iteration, the model changes randomly, which creates an ensemble "on the fly" from a single network with shared parameters.

Data Augmentation. The best way towards better generalization is to train with more data, but data is often limited. By using data augmentation, fake data is created from the existing data and added to the training set. Even simple operations like translation or adding per-pixel noise often already greatly improves generalization. However, it is extremely important that the new data preserves the semantics to make data augmentation work at all.

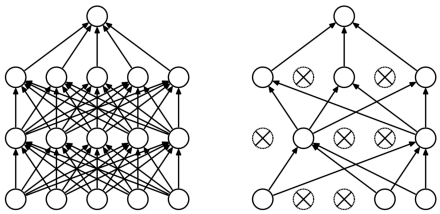


Figure 25: **Dropout.** Neural Network (left) and a dropout example (right).

2.3 Imitation Learning

2.3.1 Formal Illustration

In general, imitation learning is useful when it is easier for an expert to demonstrate the desired behaviour rather than to specify a reward function.

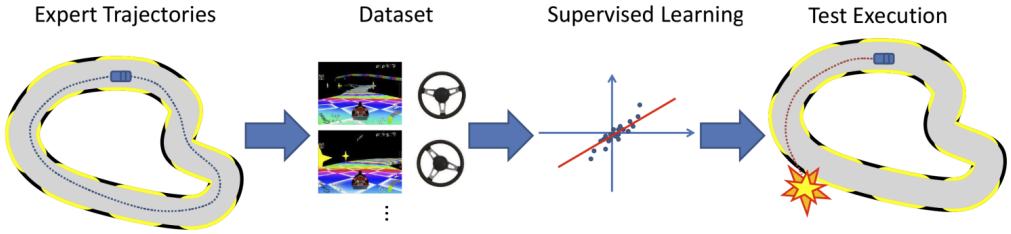


Figure 26: **Imitation Learning.** General concept of imitation learning.

As seen in Fig. 26, imitation learning is based on expert's demonstrations (which are also known as trajectories). A dataset of the experts demonstrations is constructed (such as the direction, car speed, usage of break, video recordings). The taken actions of the demonstrators are denoted as:

$$a^* \in \mathcal{A} \tag{13}$$

where $\mathcal{A}$ is the set of all possible actions one can take. Those may be discrete or continuous (turn angle, speed etc.). a^* is seen as the optimal actions one can take. Based on the situation or state the car is in, the demonstrator shows the best policy (action to take in a certain situation):

$$\pi^* : \mathcal{S} \rightarrow \mathcal{A} \tag{14}$$

where $\mathcal{S}$ is the set of all possible states, which are mostly only partially observed (e.g. recordings, game screen). Being equipped with the optimal actions a^* and optimal policy π^*, we are then trying to create a mapping function for our agent:

$$\pi_\theta : \mathcal{S} \rightarrow \mathcal{A} \tag{15}$$

given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, which maps a state to a concrete action. By doing so, the agent will land in some new state $s + 1$. We can either model/simulate the state dynamically via a state distribution:

$$P(s_{i+1}|s_i, a_i) \tag{16}$$

or have a deterministic mapping function T:

$$s_{i+1} = T(s_i, a_i) \tag{17}$$

Rollout. For now, assume that we already have a learned policy π_θ with parameters θ. The learned policy can be rolled out (tested). Given the initial state s_0 sequentially execute the policy leads to the next action:

$$a_i = \pi_\theta(s_i) \tag{18}$$

and either sample $s_{i+1} \sim P(s_{i+1}|s_i, a_i)$ or calculate $s_{i+1} = T(s_i, a_i)$ depending on what simulator you choose. This yields the trajectory of the agent:

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{19}$$

General Imitation Learning. In a general setting, the state distribution $P(s|\pi_\theta)$ depends on the rollout of the current policy π_θ. $P(s|\pi_\theta)$ can be described as all states reachable by the policy π_θ. We can now optimize θ by calculating:

$$\underset{\theta}{\operatorname{argmin}} \, \mathbb{E}_{s \sim P(s|\pi_\theta)}[\mathcal{L}(\pi^*(s), \pi_\theta(s))] \tag{20}$$

Here, the expectation is calculated over all states that are reachable by π_θ. We get an interdependency between states being sampled from a distribution dependent on θ and the θ itself being optimized.

2.3.2 Behavior Cloning

Behavior Cloning is a simpler variant that uses a provided state distribution P^* by the expert. The state s and the expert's action a^* are derived from P^*, thus reducing it to a supervised learning problem, makes it simpler to train:

$$\underset{\theta}{\operatorname{argmin}} \underbrace{\mathbb{E}_{(s^*,a^*)\sim P^*}\left[\mathcal{L}(a^*, \pi_\theta(s^*))\right]}_{\sum_{i=1}^{N} \mathcal{L}(a_i^*, \pi_\theta(s_i^*))}$$

Since we now use the data within a standard supervised learning setting we need to make the IID assumption. IID means that all data points are sampled independently and identically distributed. This assumption is not true: The state distribution depends on the parameters of the current policy which in turn depends on the state distribution. In other words, the next state that is observed is not an arbitrary one, but it depends (it is not independent) on the previous state, the action that the policy takes given the previous state and the state dynamics model that generates the new state. In practice, this implies that a very large amount of relevant states are not present in the dataset - if the policy does the slightest mistake (policies are never perfect), the agent will deviate from the expert trajectory and observe a new state that is outside the training distribution. Given that unknown state, the agent will deviate further as it does not know what to do and finally collide. It is therefore important to include state-action pairs into the training that are outside the original training distribution / different from what the expert has observed. If we would know all reachable states, we could sample an IID dataset, but in practice we only have the states that the expert observed while driving which is a very small subset of all reachable states.

Data Aggregation (DAgger):
DAgger is a method to solve the train-test distribution mismatch. It starts with a fixed dataset that is provided by an expert. A policy is trained with this dataset and rolled out into an environment, which leads to a new on-policy dataset. For this new dataset an expert is queried to label it. The new dataset is then aggregated to the original one. This sequence is illustrated in Fig. 27. This aggregation is not done inside the training loop but once in a while for a few iterations. A problem with DAgger is that it overfits easily to the main mode of demonstrations. It can not handle unbalanced datasets very well.

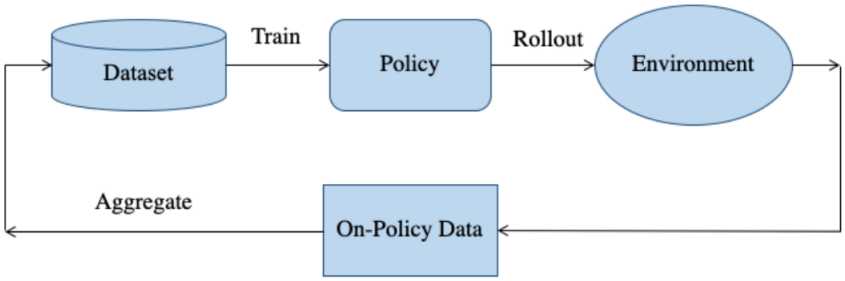


Figure 27: Functioning of DAgger [23]

DAgger with Critical States and Replay Buffer:
To improve DAgger we can add a critical state subsampling and a replay buffer, shown in Fig. 28. With this improvement, DAgger can be used for self-driving tasks. Critical states are sampled from the on-policy data. The sampled data is chosen carefully based on the utility they provide to the learned policy in terms of driving behavior. The replay buffer progressively focuses on the high uncertainty regions of the policy's state distribution. It progressively adds the data to the dataset, thus avoiding abrupt changes in the dataset.

2.3.3 Applications of Imitation Learning

In this section some practical applications of imitation learning are presented.

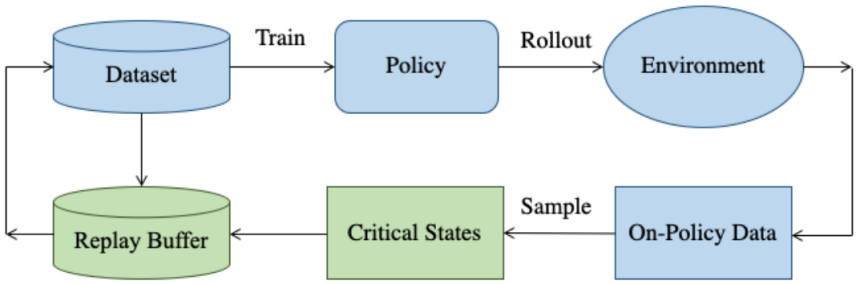


Figure 28: Functioning of DAgger with Critical States and Replay Buffer [23]

ALVINN
An Autonomous Land Vehicle in a Neural Network (ALVINN) [22]. Proposed in 1988, using a very small neural network with only 3 layer (36k parameters) to map road images to turn radius with 45 direction output units (possible decisions). It was trained on simulated road images (still common practice today) and tested on unlined paths, lined city streets and interstate highways.

PilotNet
Proposed by Nvidia [7] as an end-to-end learning approach for self-driving cars. Like in ALVINN, a real vehicle was used and instrumented with 3 camera angles for data augmentation (providing more data) that can be used to avoid the train/test distribution missmatch (to get into states where you haven't been before) instead of, for example, using DAgger. Applying different camera angles, one has to correct the trajectory accordingly and adjust for the shift and rotation as well as providing the adjusted steering command based on which camera angle one is using as an input to the loss function (see Fig. 29).

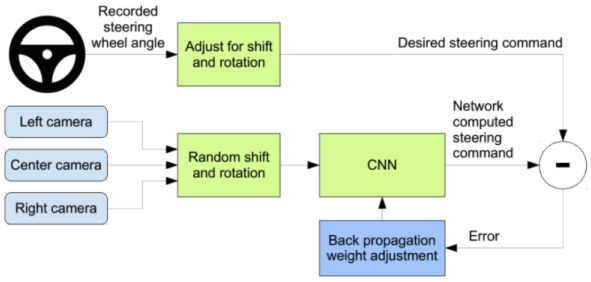


Figure 29: **Pilotnet.** Graphical illustration of the idea behind PilotNet.

In terms of network architecture, a CNN, inspired by AlexNet, is used with 250k parameters, consisting of a normalization layer, 5 convolutional and 3 fully connected layers, trained on 72 hours of driving.

VisualBackProp
As we have a monolithic approach to self-driving in imitation learning, it is hard to understand how decision are made inside the neural network (often seen as a black box). VisualBackProp [6] tries to highlight the decision making process. The idea behind VisualBackProp is to find salient image regions that lead to high activations. In other words, what region of the seen image is being used to form a decision by the network. This is done by forward passing the images through the network and iteratively scale-up the activations (feature maps): Feature maps are extracted at certain points in the network, scaled and pointwise multiplicated with the previous feature map. The final visualization mask should indicate where the network concentrates on.
To check if this visualization mask is of any use, the salient objects are shifted in the original image to see if it has any effect on the predicted turn radius. Based on the paper, the salient object given by the visualization mask actually seem to be important for the decision process.

2.4 Conditional Imitation Learning

In normal driving scenarios, a vehicle that is trained end-to-end to imitate an expert can not make the decision to take a specific turn at an upcoming intersection without any additional input [12]. In conditional imitation learning, in addition to the state s_t, a conditional signal c_t (for example: turn left) is also given as input to the system. This results in an unique action a_t that can be performed by the vehicle. This process is shown in figure 30. The conditional signal can be obtained by GPS for example.

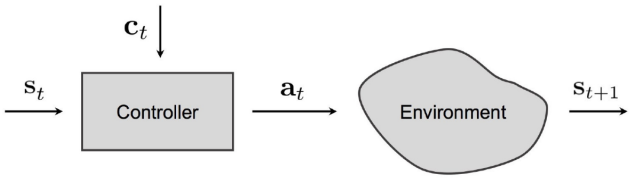


Figure 30: Conditional Imitation Learning process [12]

2.4.1 Comparison to Bevhavior Cloning

Conditional imitation learning is extending behavior cloning (the task is still reduced to a supervised learning problem). By providing a conditional signal, some changes to the notation have to be made:

The training dataset $\mathcal{D}$ (expert's trajectory) used in behavior cloning consisted of states s_i^* and an optimal actions a_i^*.

$$\mathcal{D} = \{(a_i^*, s_i^*)_{i=1}^N\} \tag{21}$$

and is extended by the conditional policy in conditional imitation learning:

$$\mathcal{D} = \{(a_i^*, s_i^*, c_i^*)_{i=1}^N\} \tag{22}$$

The objective function $\mathcal{L}$ from behavior cloning

$$\underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(a_i^*, \pi_\theta(s_i^*)) \tag{23}$$

extends to

$$\underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(a_i^*, \pi_\theta(s_i^*, c_i^*)) \tag{24}$$

Therefore in behavior cloning we are making the assumption, that

$$\exists f(\cdot) : a_i = f(s_i) \tag{25}$$

there exists a function, when inputted a state, it outputs a particular action. However, for example at an intersection, multiple actions can be valid given a certain input state s. This assumption is thus often violated in practice. In conditional imitation learning this problem is addressed:

$$\exists f(\cdot, \cdot) : a_i = f(s_i, c_i) \tag{26}$$

which reduces the number of possible actions and is therefore a better (more robust) assumption as the action is more uniquely defined (can still be violated though).

2.4.2 Network Architecture

For training a policy, two network architectures are proposed [12]. The first architecture (see Fig. 31 left) has 3 inputs: The image i, measurements m (e.g. speed of the car) and the conditional signal c. The image is encoded by an image encoder - CNN network - to get compact representations $I(i)$. The measurements and the conditional signal are processed by a small fully connected network to get $M(m)$ and $C(c)$. Those 3 processed inputs are then concatenated to one representation j. The representation j is then processed by another network to predict the action a.
The other proposed network architecture is slightly different (see Fig. 31 right). Here we are only processing the image i and measurements m to get image and measurement features $I(i)$ as well as $M(m)$. The concatenated representation are then processed by one of a certain number of subnetworks. Which subnetwork is used is decided by the command c, which acts as a switch between the specialized submodules.
In addition to the architecture-difference, the authors of this proposal also introduced noise for data augmentation. They added noise to the steering wheel (added noise into the trajectories) and had the expert drive back. Collecting only 12 minutes of data already helped with the standard issue of behavior cloning (train/test distribution mismatch).

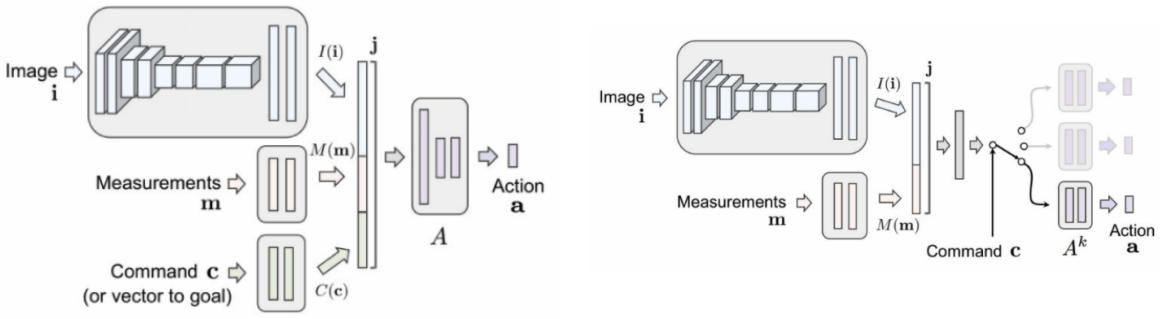


Figure 31: Two network architectures for Conditional Imitation Learning proposed by Codevilla et al. [12]

2.4.3 Applications of Conditional Imitation Learning

Exploring the limitations of Behavioural Cloning [13]. The authors are trying to deal with the causal confusion problem in imitation learning, where the true cause of the action is confused with the wrong cause. For example, when the traffic light turns green and the car stands still, the vehicle does not recognize the green light as a reason to accelerate as the correlation betweeen standing still in one frame and the next is very high. On factor they are proposing (to alleviate this issue) is, in addition to predicting the vehicle control, to also predict the speed of the car.

Neural Attention Fields for End-to-End Autonomous Driving [9]. Combines implicit representations and attention. Attention is important as the input to self-driving system is very high dimensional. One needs to very efficiently narrow down whats important for the actual driving situation. In this approach, a MLP iteratively compresses the high-dimensional input into a compact representation based on attention. These attention maps are querying the inputs based on what the model believes to be important at the current iteration to make the next decision. Hereby, one get rid of a lot of unnecessary input and thus reducing the dimension of the data. The model then predicts waypoints and auxiliary semantics in birds-eye-view space, which aids the learning process.

3 Direct Perception

3.1 Introduction to Direct Perception

So far we have seen two approaches to Self-driving:

- **Modular Pipeline**, which is composed of different modules, e.g. Low-level Perception and Path Planning. Advantages are, that it is easier to develop because it is modular and the individual models are interpretable. Also it is easier to inspect a modular pipeline e.g. in case of an failure. Disadvantages are the requirement of a lot of expert knowledge for the design of the pipeline which may easy lead to errors. Another disadvantage is, that training the entire system end-to-end is very challenging.
- **End-to-end**, with the two dominant algorithms Reinforcement and Imitation Learning. End-to-end learning benefits from the fact, that we train directly for the task we are interested in. The disadvantages lay in the fact, that those networks do not generalize very well and that they need a lot of data. In opposite to Modular Pipelines they are less interpretable, since they are a full model.

The idea of **Direct Perception** is to have a hybrid model between those two approaches, imitation learning and modular pipelines. Like modular pipelines it is also modular, but with much less modules. It learns to predict from the sensory input to an intermediate, low-dimensional representation (see Fig. 32). This leads to decoupling of perception from planning and control. The benefit of the decoupling is, that on the one hand the model generalizes better and on the other hand it is better interpretable. The approach of direct perception also benefits from allowing to exploit classical controller, e.g. PID-Controllers or final state machine, as well as learned controllers.

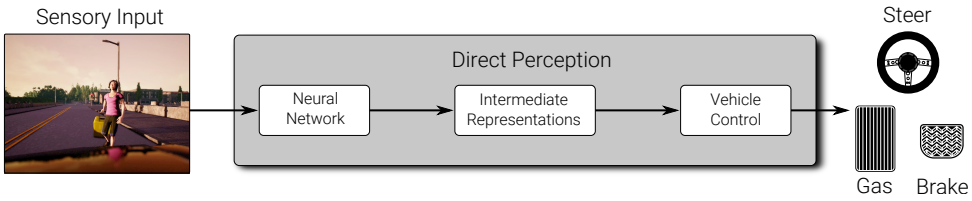


Figure 32: **Direct perception.** An illustration of direct perception. From sensory input over an intermediate representation to control commands.

3.1.1 Example of a direct perception model

In the following we will look at a model, which implements direct perception for autonomous driving [8]. The goal of the paper was to implement a direct perception controller for highway driving and test it in simulation.

Affordances. To reach this goal a set of *affordances* should be predicted. Affordance is a psychological terminology and describes attributes of the environment, which limit the space of possible actions. For the model 13 affordances (e.g. the driving angle relative to the road) and two states (vehicle on the lane-markings or between the lane-markings) were defined. Some affordances depend on the state e.g. if the vehicle is in lane the distance to the markings matter.

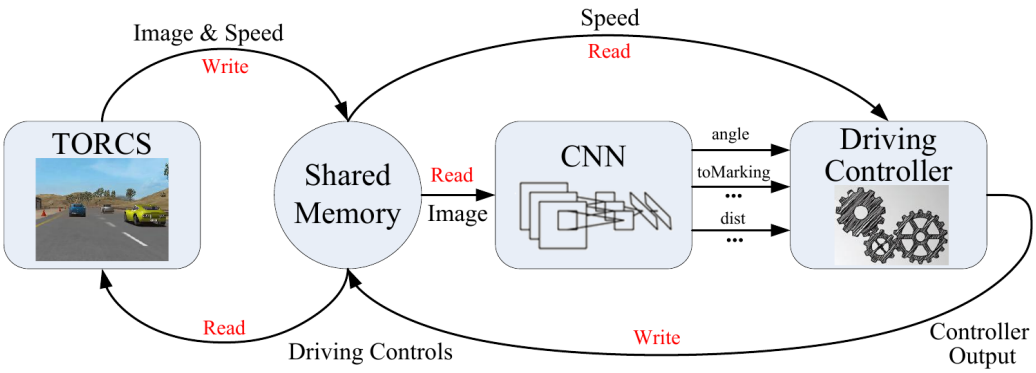


Figure 33: **DeepDriving [8] architecture.** Overview of the DeepDriving architecture, presented in [8]

As shown in Fig. 33 the model is trained and evaluated in the open source car racing simulator TORCS, which allows testing highway-driving. The model and the simulator communicate with an shared memory, where the

image and speed is written by the simulator and the driving controls are written by the model. From the same location, the model reads the input images and the speed and the controller reads the driving commands. The network itself is an AlexNet [17] consisting of five convolutional layers and four fully connected layers with 13 output neurons, one for each affordances to be predicted. The affordance indicators are trained with a ℓ_2 loss. Data - the image and speed as well as the affordances - can be recorded directly from the simulator.

```
while (in autonomous driving mode)
   ConvNet outputs affordance indicators
   check availability of both the left and right lanes
   if (approaching the preceding car in the same lane)
      if (left lane exists and available and lane changing allowable)
         left lane changing decision made
      else if (right lane exists and available and lane changing allowable)
         right lane changing decision made
      else
         slow down decision made
   if (normal driving)
      center_line= center line of current lane
   else if (left/right lane changing)
      center_line= center line of objective lane
   compute steering command
   compute desired_speed
   compute acceleration/brake command based on desired_speed
```

Figure 34: **Controller.** Pseudo code for the driving controller logic.

Controller. The Driving controller takes the affordances, e.g. the steering angle, and provides the controller output for the vehicle, which is put to the shared memory, where the simulator reads it. The affordances logic controller is implemented as a simple state machine (see Fig. 34). Since it is a very simple, hand-engineered controller it leads to very abrupt driving behaviour. While the car is in autonomous driving mode the ConvNet outputs affordance indicators and checks the avaibility of left and right lane. If the vehicle is approaching the preceding car in the same lane the controller checks if there is a left lane available and if lane changing is allowable. In that case the left lane decision is made. In the case, that just the right lane exists, is available and lange changing to the right lane is allowed, the right lange changing decision is made. In any other case the decision making is slowed down. Afterwards it is checked if a lane changing decision was made. In that case the new center line is the center line of objective lane (left or right) else the center line is the center line of the current lane. In the end the resulting steering command, the desired speed and acceleration and brake commands based on the desired speed are computed.

The steering controller is defined by the following equation (27):

$$s = \theta_1(\alpha - d_c/w) \tag{27}$$

The output of the equation is the steering command s, which is directly given to the simulation. The parameter α describes the relative orientation with respect to the tangent of the road. The controller tries to minimize the relative orientation α. The parameter d_c describes the distance to the centerline and the parameter w the road with. Those two parameters tells the controller to steer more, if the vehicle is farer away from the center line. θ_1 is an adjustable parameter.

The speed controller (also called the "optimal velocity car following model") is defined by the following equation (28):

$$v = v_{max}(1 - \exp(-\theta_2 d_p - \theta_3)) \tag{28}$$

The output v is the target velocity. v_{max} is the maximal velocity of the vehicle and d_p describes the distance to the preceeding car. Similar to the steering controller θ_2 and θ_3 are tuneable parameters.

In the evaluation it turns out, that the affordances were predicted correctly and the model is working generally. The way of driving is kind of abrupt and maybe not very comfortable. With network visualisation techniques the creators of the model found out, that the model focuses on relevant parts of the scene.

3.2 Conditional Affordance Learning

To transfer direct perception to cities we can use a similar approach as in imitation learning where we pass the high level directional input from GPS together with the video input into a neural network that predicts the affordances which are passed to the controller. The idea of Conditional Affordance Learning was proposed in the paper *Conditional Affordance Learning for Driving in Urban Environments* [24]. They used the CARLA Simulator with the goal to drive from A to B as fast, safely and comfortably as possible. CARLA also counts the

infractions (e.g., driving on wrong lane, driving on sidewalk, running a red light, violating speed limit, colliding with vehicles, hitting other objects) that occur. Therefore the goal is to go from A to B without doing any or with doing as little as possible of those infractions. To reach this goal they utilizes only 6 different affordances (also see Fig. 35):

- Distance to centerline
- Relative angle to road
- Distance to lead vehicle
- Speed signs
- Traffic lights
- Hazard stop

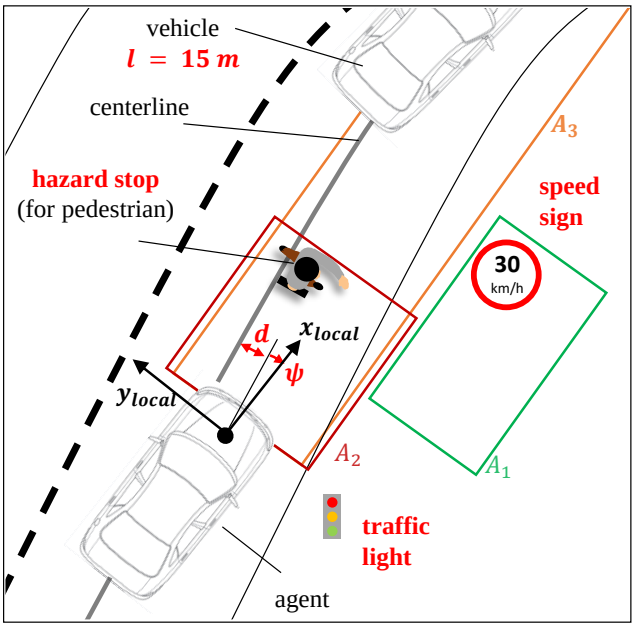


Figure 35: **Illustration of affordances** Illustration of the different affordances defined by the model.

In order to generate a data set that comprises both input images and the ground truth affordances we can run our simulation. Now the question is: How do we obtain these affordances from the simulatior while driving around in the simulation?
The distance to the center line and the relative angle to the road are directly available from the simulator. For the distance to the lead vehicle, the speed signs, the traffic lights and the hazard stops they implemented certain regions of attention in bird's eye view.

The **longitudinal control** is a simple finite-state machine combined with a PID controller. If the hazard stop has been predicted by the perception module we do a full break and we do not care about the other states. Following this, if there is a red light we try to come to a stop in front of it. If neither the hazard stop nor a red light have been detected but we are over the speed limit we try to reduce the speed of the vehicle to follow the speed limit that currently holds. Finally, if none of the above conditions hold, we are either in the car following mode, which means there is a car in front of us and the speed is adjusted to keep a certain distance to it, or we are in the cruising mode, if no vehicle is in front.
As **lateral control** they implemented the *Stanley controller*, which, similar to the controller we have seen before, has a relative steering term, a cross track error term as well as a damping term to avoid oscillations:

$$\delta(t) = \psi(t) + arctan\left(\frac{kx(t)}{u(t)}\right)$$

Perception Stack. They implemented a multi-task model where they used a common backbone such that there is only a single forward pass with multiple shallow heads. The dataset is comprising random driving using the controller operating on the ground truth affordances from the simulator. The loss functions are class-weighted cross-entropy (CWCE) for discrete and mean average error (MAE) for continuous affordances.

Data Collection. The data is collected based on navigation based on the true affordances. At each intersection a random navigational command is provided as input.

Data Augmentation. For data augmentation image flipping can not be used because this would change

right-hand driving to left-hand driving but changing color, contrast and brightness of the images, gaussian blur and noise are used. It is also important to make sure that all possible scenarios occur in training data which is why rear end collisions are provoked.

3.3 Visual Abstractions

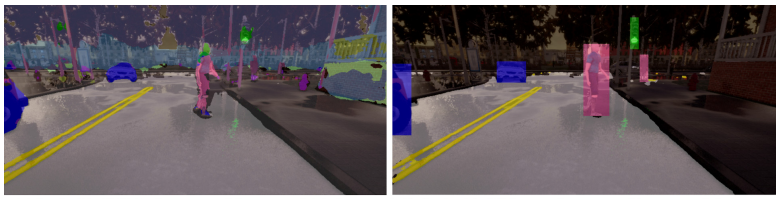
The models we have looked at so far used hand engineered low level affordances as intermediate representations where of course also human error goes into the design. Now the question is: Can we use more generic intermediate representations such as monocular depth estimation, stereo depth estimation or semantic segmentation in order to improve the robustness of self-driving agents. And if we use those representations, how can we efficiently learn the perception stack?
In [28] the authors analyzed various intermediate representations with respect to their ability to improve robust driving behavior. They found out that intermediate representations often improve the results and generalisation. In case of self-driving they found that depth estimation and semantic segmentation provide the larges gains. In particular depth estimation generalizes better in offroad-driving where the geometry matters a lot while semantic segmentation is a useful cue for inner-city driving.

What characterizes a good visual abstraction? A good visual abstraction has to be *invariant*, that means it has to hide irrelevant variations from the policy. Furthermore it should be *universal*, *data efficient* in terms of memory and computation and *label efficient*, which means that little manual effort should be required.

Semantic segmentation as visual abstraction. On the one hand semantic segmentation encodes task-relevant knowledge (e.g., if the road is drivable) and priors (e.g. grouping). It can be processed with a standard 2D convolutional policy network. On the other hand labeling for one cityscapes image requires 90 min, which becomes really expensive, when thousands or millions of images need to be annotated. A possible solution to reduce the effort of labelling is a visual abstraction network.

Visual abstraction model. The visual abstraction network receives a state as an input and yields a visual abstraction ($a_\psi : \mathbf{s} \rightarrow \mathbf{r}$). The control policy takes this abstraction, the navigational command and the velocity and predicts an action ($\pi_\theta : \mathbf{r}, \mathbf{c}, v \rightarrow \mathbf{a}$). Combining both yields: $\mathbf{a} = \pi_\theta(a_\psi(\mathbf{s}))$.
The data sets needed for the training of this model are n_r images annotated with semantic labels $\mathcal{R} = \{\mathbf{s}^i, \mathbf{r}^i\}_{i=1}^{n_s}$ and n_a images annotated with expert actions $\mathcal{A} = \{\mathbf{s}^i, \mathbf{a}^i\}_{i=1}^{n_a}$. Since the n_a is cheaper to obtain, it can be assumed $n_r << n_a$.
During training the visual abstraction network a_ψ is trained with $\mathcal{R}$. This network is then applied to acquire the control data set $\mathcal{C}_\psi = \{a_\psi(\mathbf{s}^i), \mathbf{a}^i\}_{i=1}^{n_a}$. Then, the control policy $\pi_\theta(\cdot)$ is trained using $\mathcal{C}_\psi$.

Analysis of visual abstractions. In [5] the authors analysed the importance of several aspects of the training data. They used a visual abstraction model described in the previous section. The model was evaluated in CARLA 0.8.4 NoCrash benchmark with random start and end location. As metric the percentage of successfully completed episodes were used.
The privileged segmentation forms the ground truth for the semantic labels and therefore the upper bound for analysis. The inferred segmentation are not as perfect as the ground truth and are produced by the visual abstraction model.
All semantic classes sum up to fourteen. As the labelling takes a lot of time in this case the six most relevant classes are chosen. This includes lane markings, vehicles, pedestrians, traffic lights and roads. A hybrid version can also be tried where the model is trained on two stuff and four object classes. The stuff classes include the road and lane markings. The object detection boxes include the vehicle, pedestrians and traffic lights. This reduces the annotation time and lets the model focus on the relevant parts. The Figure 36 shows a summary of the comparison of the models trained with the two different image annotations.



Trained with 6400 finely annotated images and 14 classes
Annotation time ≈ 7500 hours, policy success rate = 50%

Trained with 1600 coarsely annotated images and 6 classes
Annotation time ≈ 50 hours, policy success rate = 58%

Figure 36: **Comparison of two ways to annotate the images**. The hybrid segmentation takes significantly less time to annotate and even leads to a higher policy success rate.

To identify the most relevant classes different models with a different amount of classes were trained. One model with 14 classes, including e.g. vegetation and walls. One model with 7 classes, which just contains classes direct relevant to the traffic e.g. without walls and vegetation but with sidewalks. One model with 6 classes, without sidewalks. And finally one model without lane markings, but with vehicles, pedestrians, green light and red light.
The analysis showed that moving from fourteen to six classes does not hurt the driving performance, but the performance drastically drops, when lane markings are removed.
For all following experiments 6-class representation is used. Overall the hybrid representation matches the standard representation. In difficult scenarios (with a dense traffic) the performance slightly improves. The annotation time for standard segmentation (pixel-wise) takes 300 seconds per image and per class while the hybrid annotation (with bounding boxes) takes around 20 seconds per image and per class. A detailed overview about the result (with 6-class-images is given in Fig. 37)

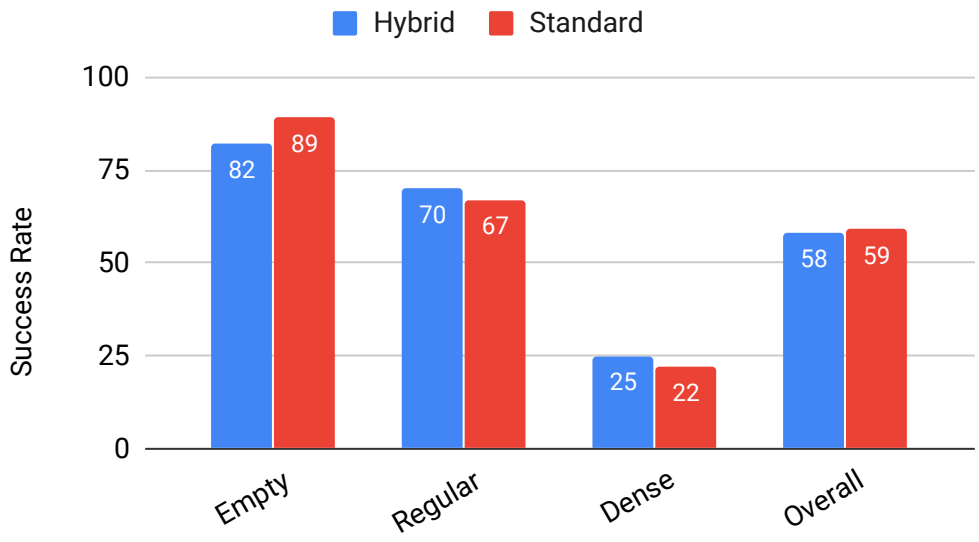


Figure 37: **Different representations.** The bars show the success rates of different traffic densities with hybrid and standard segmentation.

In summary a model trained with 6400 finely annotated images and 14 classes has an annotation time of roundly 7500 hours and a policy success rate of 50% while a model trained with 1600 coarsely annotated images and 6 classes has an annotation time of roundly 50 hours and a policy success rate of 58%.

3.4 Driving Policy Transfer

We have seen how intermediate representations are useful for decoupling perception from planning and control in order to make policy learning easier. But of course not all hardware is equivalent and if you want to transfer a driving policy from a vehicle that is learned in the simulator to a vehicle that is operating in the real world it might also be useful to decouple control from planning further.
This is what is investigated in the paper *Driving Policy Transfer via Modularity and Abstraction* [21]. The problem that is considered in this paper is that driving policies learned in simulation often do not transfer well to the real world ad therefor the paper makes an additional step from the direct perception idea towards a modular pipeline. The idea is to encapsulate the driving policy such that it is not directly exposed to raw perceptual input or low-level control (input: semantic segmentation, output: waypoints).

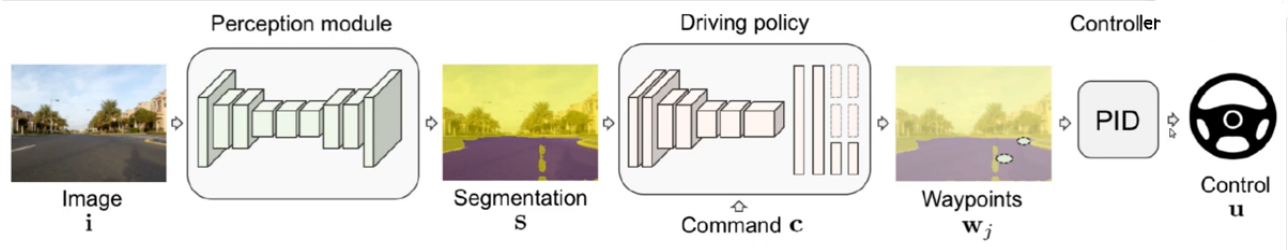


Figure 38: **Driving Policy Transfer:** An illustration of the pipeline proposed in the paper.

As you can see in Figure 38 the input image i is segmented into a segmentation S by the perception module. Afterwards the segmentation and a command $\mathbf{c}$ are input to the driving policy to predict waypoints $\mathbf{w}_j$ that are passed to the PID controller in order to control the vehicle. This approach allows for transferring a driving policy without retraining or finetuning because the only thing that has to be replaced for applying that policy to a new vehicle is the controller.

The representation that is used takes a semantic segmentation (per pixel "road" vs "non-road") as input and outputs 2 waypoints (distance to vehicle, relative angle wrt. vehicle heading) where one is sufficient for steering and the second one is needed for breaking before turns.

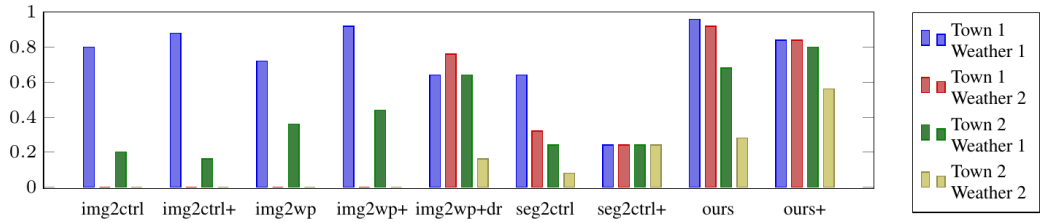


Figure 39: **Success Rate over 25 Navigation Trials**

Results. The Figure 39 shows the results of an ablation study where different variants of the model are compared. One model goes from images to control, one from image to waypoints, one from semantic segmentation to control and the full model called "ours" going from image to semantic segmentation to waypoints to control. The plus added to the descriptions describes an added data augmentation. Conditional imitation learning is used as the driving policy and a PID controller is utilised for lateral (steering) and longitudinal (acceleration, braking) control. It becomes obvious that the full model "ours" yields the best results over the different towns and weather conditions showing that it generalises well.

3.5 Online vs. Offline Evaluation

In this section we discuss the evaluation of self-driving agents in particular, we are going to have a look at online vs. offline evaluations proposed in the paper *On Offline Evaluation of Vision-based Driving Models* [11]. Figure 40 illustrates the difference between offline evaluation *(a)* and online evaluation *(b)*.

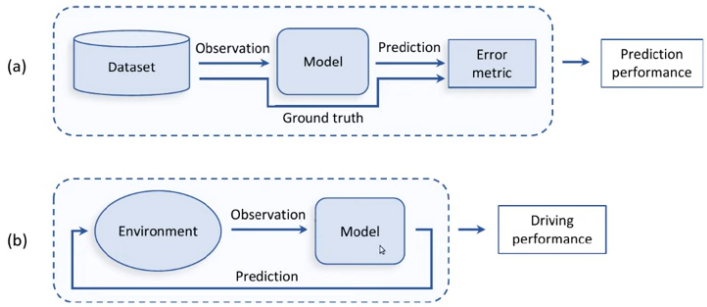


Figure 40: **Online vs. Offline Evaluation**

Online Metrics. In online evaluation the environment generates an observation that is input to the model which makes a prediction that influences the environment and so on. Then we can measure how successful it is based on the model driving in the real world. For the online evaluation the following metrics can be used: The success rate which is the percentage of successfully completed routes. The average completion which is the average fraction of the covered distance to the goal. And the kilometres per infraction which describes the average driven distance between two infractions. But because that is expensive and dangerous people often report offline evaluation metrics in research papers.

Offline Metrics. Offline evaluation metrics can be computed based on pre-recorded validation datasets and are therefore cheap and easy. As you can see in *(a)*, we have a pre-recorded dataset which gives us an observation that we input in our model but it also has the ground truth action and we can simply measure the error

between the prediction of the model and the ground truth. The offline metrics employed in this case are the following: The squared and absolute error of the steering error, the speed-weighted error of the steering angle, the cumulative speed-weighted absolute error, the quantized classification error and the thresholded relative error of the steering angle.

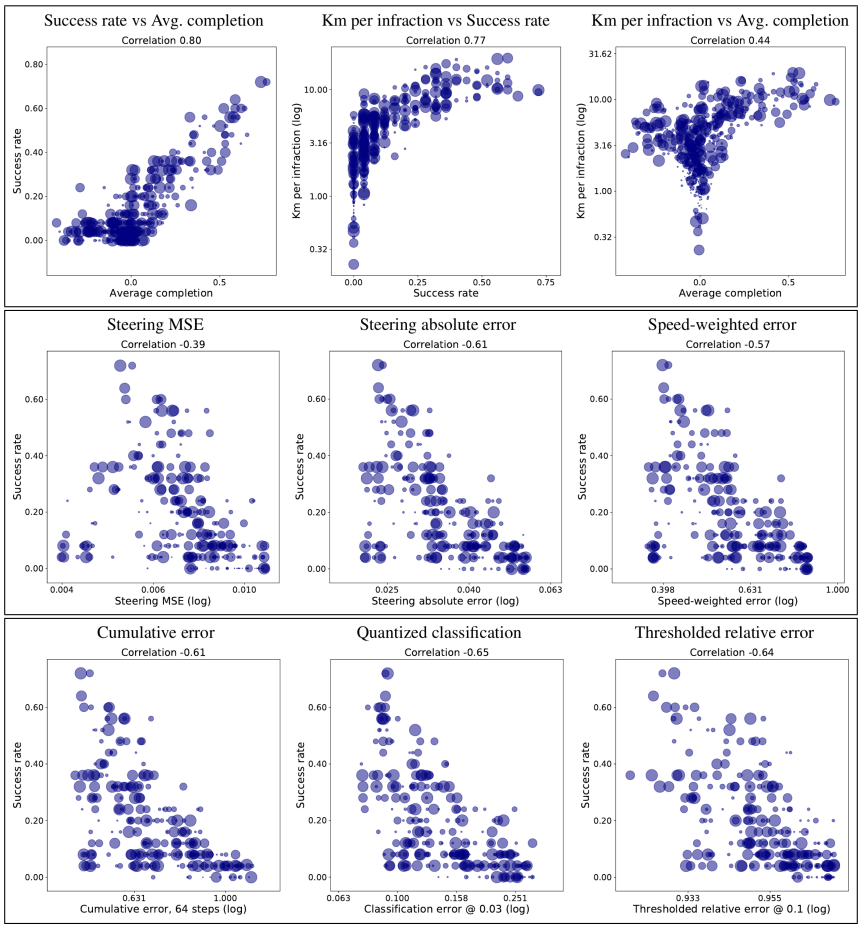


Figure 41: **Comparing the online metrics with themselves (first) and to the offline metrics (other).**

Results. The Figure 41 shows the generalisation performance of 45 different models with varying data set sizes, augmentation, architecture etc.. The radius of the circles shows the training iteration.
As can be seen in the first plot, the online metrics correlate well with each other while only average completion does not correlate well with kilometres per infraction.
The other plots show the correlation of online with offline metrics. It becomes clear that those do not correlate well. Using the absolute steering error improves over the steering MSE, the speed-weighted error however does not lead to an improved corrleation (second). The same holds true for the cumulative error. The quantized metrics are the most robust metrics and do lead to the best correlation results but are still not good (third).

Case Study. A case study was conducted to give further insight into why online and offline metrics do not correlate well. Two models are tested. One model is trained with a single camera and l_2 loss (this is the bad model). The other model is trained with three cameras, data augmentation and l_1 loss (this is a good model). The Figure 42 (top) shows the performance of both models through offline metrics. While model one shows very large errors which might indicate crashes their average performance is similar. The bottom image of Figure 42 shows the performance of both models through online metrics. It becomes obvious that model one (left) crashes every time while model two (right) can finish the routes.

3.6 Summary

Direct perception predicts intermediate representations. As intermediate representations low-dimensional affordances (from psychology) or classic computer vision representations (e.g. semantic segmentation, depth) can be used. Decoupling perception from planning and control offers some benefits and a hybrid model between

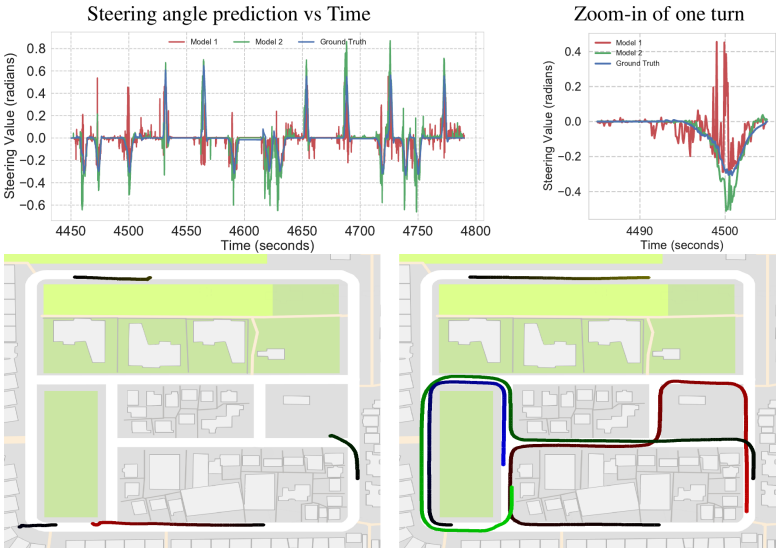


Figure 42: **Evaluation of the Case Study**

imitation learning and modular pipelines. Direct Methods are more interpretable as the representation can be inspected. Effective visual abstractions can be learned using limited supervision. For a better transfer of driving policies planning can also be decoupled from control e.g. from simulation to the real world. One other important point is, that offline metrics are not necessarily indicative of online driving performance and one has to be careful in conclude from offline to online scenarios.

4 Reinforcement Learning

4.1 Markov Decision Processes

In the supervised learning setup (e.g. imitation learning), a large number of expert demonstrations are needed, as well as loss functions that indicate how close we are to the expert behaviour (e.g. per-frame loss). These only have a short-term effect which does not anticipate the future. Now we move away from this approach and instead attempt to build models based on the laws that are of actual importance to us, using the principles of exploration and exploitation while interacting with an environment. These type of loss functions are much more general, e.g. they may minimize the time to the target location, the number of collisions, risk or maximize comfort, and so on. Different types of learning are summarized in Table 1.

	Supervised Learning	Unsupervised Learning	Reinforcement Learning
Data	set of data-label pairs $\{(x_i, y_i)\}$	dataset $\{(x_i)\}$	agent interacting with environment provides reward signals
Goal	learn mapping $x \mapsto y$	discover underlying structure	choose actions that maximize reward
Examples	classification, regression, imitation/affordance learning	clustering, dimensionality reduction, feature learning	manipulation/control tasks (e.g. control robot arms)

Table 1: Comparison between different types of learning.

The general reinforcement learning setup looks as follows (illustrated in Fig. 43): There is an agent placed within an environment. The agent observes the state s_t at time t of the environment. This state is typically only a very limited part of the global environmental state, encompassing key information. Then it sends its action a_t to the environment, in the self-driving context this might correspond to the agent seeing a picture of the street and steering the car left or right. Afterwards, the agent receives a reward r_t in the form of a numerical value as feedback of how good or bad the previous action choice was. Then the environment transitions to the next state s_{t+1}. The overarching goal is to maximise future reward through the actions taken by the agent. Challenges arise, because

- actions may show effects much later on
- reward is received with a delay
- immediate reward sometimes has to be sacrificed for large long-term rewards.

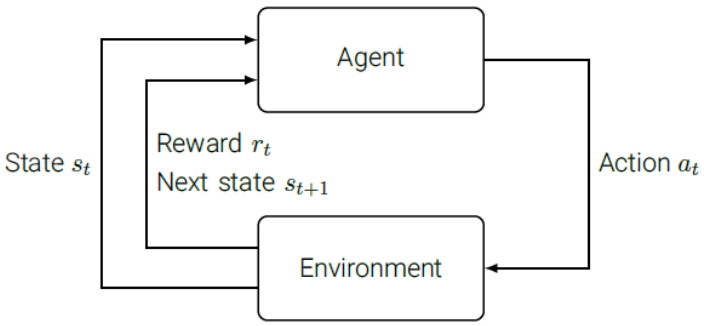


Figure 43: Overview of the basic principle of reinforcement learning.

4.1.1 Examples

Here are some examples for typical reinforcement learning problems:

- Given a cart with an attached pole attached by a hinge to the center of the cart, take action by going left or right to balance the pole into an upright position (reward case). States provide information about the angle, angular velocity, position and velocity of the cart via four values. Fig. 44a illustrates the setup.

- Given a robot with numerous joints, apply torques to said joints to move robot forward. The agent is rewarded if the robot is both upright and moving forward.
- An agent has to learn how to play Atari games, where the goal is to maximize the score based on arrow-key presses which are (negatively) rewarded whenever the score (de-/)increases.
- In the game Go, knowing the state of the board, place next pieces and try to win the game, see Fig. 44b.
- In self-driving the agent steers and accelerates based on images as input is supposed to follow the street (Fig. 44c).

(a) cart pole (b) Go (c) self-driving

Figure 44: Illustrations of examples for reinforcement learning problems.

4.1.2 Markov Decision Processes

Now that an intuition for these kinds of problems have been established, lets formalize the approach mathematically. A **Markov Decision Process (MDP)** consists of a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$. $\mathcal{S}$ is the set of states (e.g., in the case of Go: all possible board configurations), $\mathcal{A}$ the set of possible actions, $\mathcal{R}(r_t|s_t, a_t)$ the distribution of the current reward given a (state, action) pair and $P(s_{t+1}|s_t, a_t)$ the distribution over the next state given a (state, action) pair. γ is a discount factor which increasingly diminishes the the degree to which future rewards are taken into account for the present action choice. MDPs have the so-called **Markov property**: Knowing the sequence of past states adds nothing to the present; in other words, the future is independent of the past given the present: $P(s_{t+1}|s_t) = P(s_{t+1}|s_1, ..., s_t)$.

To reiterate the reinforcement learning loop in the context of MDPs: At the starting time $t = 0$, the initial state $s_0 \sim P(s_0)$ is sampled. Then, the following steps are repeated until end of the rollout of the policy: The agent selects and action a_t according to a policy π and then the environment responds by sampling the corresponding reward $r_t \sim \mathcal{R}(r_t|s_t, a_t)$ and next state $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ and hands these over to the agent.

4.1.3 Policy

A **policy** π is a function from $\mathcal{S}$ to $\mathcal{A}$ which provides the next action based on a given state, and the goal is to find the best policy. In a discrete state and action space this corresponds to a look-up table or a neural network with some parameters. One can distinguish between deterministic policies $a = \pi(s)$ with a direct mapping and stochastic policies $\pi(a|s) = P(a_t = a|s_t = s)$ with a distribution of probabilites over the actions for a given state. Even though MDP policies depend on the current state, past observations may be included, e.g. in the form of a buffer of the ten last images.

In imitation learning, the policy was learned from expert data, which made it a supervised learning problem. In reinforcement learning on the other hand, policies are learning through trial-and-error only; the quantity that is maximized in this case is the expected future reward. On the basis of interactions with the environment, the agent discovers good actions and can then improve the policy.

4.1.4 Exploration vs. Exploitation

In order to discover those desirable actions, the state-action space need to be explored, since they are not provided through any kind of preexisting knowledge base. The first of two fundamental tasks of RL is thus **exploration**, which has the aim of gathering the reward signals of as many state-action pairs as possible. The better the agent knows the space, the more optimal its performance can get. During this process however, total reward has to be sacrificed, because many of the actions that haven't been tried yet will not yield a worthwhile

reward. If real-life robots are used as agents, the exploration phase can additionally become dangerous, if the agent simply tries out everything to find out what will happen. The second fundamental task of RL is **exploitation**, is to make the most use of what has been learned in exploration, i.e. choose the best action in a given state out of all actions that have been tried yet. This however disregards the yet unexplored areas, which might contain better rewards.

These two tasks complement each other, but they also form a trade-off. Because of this, exploration and exploitation need to be balanced. This can be achieved via ϵ-greedy exploration. For this all possible actions with non-zero probabilities are tried out. ϵ is the probability by which an action is chosen at random and with a probability of 1-ϵ the best action (greedy action) discovered so far is chosen. ϵ is usually decreasing over timet to promote exploration at first and then move towards exploitation as the known action-reward pairs increase in number.

4.2 Bellman Optimality and Q-Learning

4.2.1 Value functions

The most fundamental algorithm in reinforcement learning is **Q-Learning**. First, we have to understand what value functions are, how to deal with future reward and what the Bellman optimality principle entail. **Value functions** assign values to states in order to be able to assess how good or bad a present state is based on rewards received in the future. The **state-value function** V^π at state s_t is the expected cumulative discounted reward ($r_t \sim \mathcal{R}(r_t|s_t, a_t)$) when following policy π from state s_t:

$$V^\pi(s_t) = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t, \pi] = \mathbb{E}\left[\sum_{k\geq 0} \gamma^k r_{t+k} \middle| s_t, \pi\right] \tag{29}$$

In other words, given a policy we have trained, the state-value function tells us what the expected future reward with discount factor γ is for an initial state s_t. γ is multiplied with each new future reward and taken to the exponent of the number of time steps into the future (i.e. γ^2 for the reward two time steps into the future). The discounting assures that we don't take rewards into account now that are too far off into the future equally as much as present rewards. The hyperparameter γ lies between 0 and 1 and controls how much weight we want to give to the future, i.e., the agents far- or short-sightedness. This also avoids infinite returns if the Markov process contains cycles. Theoretically, in the calculation of the expectation, we can take the limit of all future rewards, in practice however we only consider a fixed time horizon. The reason for taking the expectation is due to the stochasticity, which may arise due to the policy, the reward distribution or state transition distribution.

The **action-value function** Q^π at state s_t and action a_t is the expected cumulative discounted reward when taking action a_t in state s_t and then following the policy π:

$$Q^\pi(s_t, a_t) = \mathbb{E}\left[\sum_{k\geq 0} \gamma^k r_{t+k} \middle| s_t, a_t, \pi\right] \tag{30}$$

The action-value function differs from the state-value function only by additionally conditioning on the action. The discount factor γ works the same way as for the state-value function.

4.2.2 Optimal Value Functions and Policy

The **optimal state-value function** $V^*(s_t)$ (the star often denotes optimal functions) is the best $V^\pi(s_t)$ over all policies π:

$$V^*(s_t) = \max_\pi V^\pi(s_t) \tag{31}$$

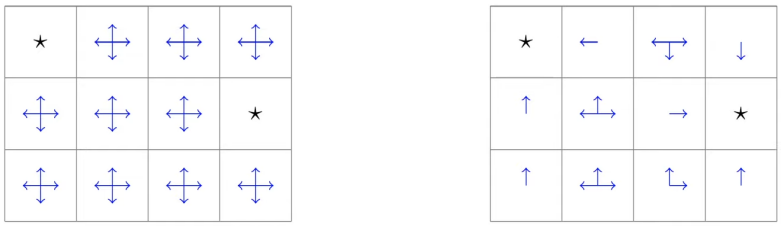
Intuitively, it tells us the maximal value we can obtain. Similarly the **optimal action-value function** $Q^*(s_t, a_t)$ is the best $Q^\pi(s_t, a_t)$ over all policies π:

$$Q^*(s_t, a_t) = \max_\pi Q^\pi(s_t, a_t) \tag{32}$$

It tells us what we can achieve in the best case if start at s_t and have conducted a_t. These optimal functions tell us what the best possible performance in the MDP would be. In most cases finding them is computationally intractable, since we would have to search the space of all possible policies. Assuming we had Q^*, we would immediately have access to the optimal policy, since

$$\pi^*(s_t) = \operatorname*{argmax}_{a' \in \mathcal{A}} Q^*(s_t, a') \tag{33}$$

Since we can not search over all possible policies, determining Q^* is hard. Looking at a very simple example may reveal how such an optimal function can look like. For this purpose we will take a look at a gridworld example. The environment consists of a discrete grid with goal positions marked by a *. The agent is positioned somewhere in the grid and has to reach the goal positions in as few steps as possible by moving left, right, up or down. Each transition has a penalty/negative reward of -1. The two extremes with regards to the policy are the random policy on the one hand, which, as the name suggests, simply randomly chooses an action and the optimal policy on the other hand, which takes the shortest possible path/paths from its current position to the goal position (Fig. 45).



Random Policy Optimal Policy

Figure 45: Left: Illustration of the random policy, right: illustration of the optimal policy.

4.2.3 Bellman Optimality Equation

In general, it is not as easy as doing a complete search for the optimal behaviour in the environment we are in as in the simple example of the grid world. In practice, we need a different approach to find the optimal policy. The **Bellman Optimality Equation** (BOE) by Richard Ernest Bellmann from 1953 will help us with that. The BOE states that:

$$
\begin{aligned}
Q^*(s_t, a_t) &= \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t, a_t\right] && (34)\\
&= \mathbb{E}\left[\sum_{k\geq 0} \gamma^k r_{t+k} | s_t, a_t\right] && (35)\\
&= \mathbb{E}\left[r_t + \gamma \sum_{k\geq 0} \gamma^k r_{t+k+1} | s_t, a_t\right] && (36)\\
&= \mathbb{E}\left[r_t + \gamma V^*(s_{t+1}) | s_t, a_t\right] && (37)\\
&= \mathbb{E}\left[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t, a_t\right] && (38)\\
& && (39)
\end{aligned}
$$

First we summarize the first expression inside the expectation as a sum. The first reward can be pulled out, because $\gamma^0 = 1$. The sum we're left with is the exact definition of the optimal state value function V^* since we are inside the definition of the optimal action value function. This V^* can then finally be reformulated as the optimal action value function maximized over all possible actions. So, basically what the BOE does in the end, is to reformulate the first expectation over all future time steps recursively, by first returning the reward of the current time step and taking γ times the maximum over all actions of the Q^* function of the next step. Due to the non-linearity of the max-operator no closed form solution is available for the BOE. Nevertheless, iterative methods, such as Q-Learning, can still be used to approximate Q^*. If we can solve the BOE, the optimal expected long term reward is locally and immediately available for each state action pair via Q^* and V^*. For V^*, we need a one-step-ahead search to get the optimal action. Q^* however cashes these one-step-ahead searches and this means, we do not have to do any rollouts anymore - we can simply choose the optimal action without having to look into the future, because it is encoded in Q^*.

4.2.4 Q-Learning

One algorithm to iteratively solve for Q^* is called **Q-learning**, in which and **update sequence** Q_1, Q_2, ... is constructed, using a learning rate α. This is how this is done:

$$Q_{i+1}(s_t, a_t) \leftarrow (1-\alpha)Q_i(s_t, a_t) + \alpha(r_t + \gamma \max_{a' \in \mathcal{A}} Q_i(s_{t+1}, a')) \tag{40}$$

$$= Q_i(s_t, a_t) + \alpha \underbrace{(\underbrace{r_t + \gamma \max_{a' \in \mathcal{A}} Q_i(s_{t+1}, a')}_{\text{target}} - \underbrace{Q_i(s_t, a_t)}_{\text{prediction}})}_{\text{temporal difference (TD) error}} \tag{41}$$

What this amounts to is the agent exploring the environment and updating a big table iteratively until we have the optimal table. On the left we have the old value for Q_i and add the difference between the target reward and the current prediction (referred to as the temporal difference (TD) error), weighed by α. As i goes to infinity, Q_i converges to Q^*. The policy is learned implicitly via the Q table. The way this is implemented is straight forward:

- Initialize Q table and initial state s_0 randomly
- Repeat:
  - Observe state s_t, choose action a_t according to ϵ-greedy strategy
  - Observe reward r_t and next state s_{t+1}
  - Compute TD error: $r_t + \gamma \max_{a' \in \mathcal{A}} Q_i(s_{t+1}, a') - Q_i(s_t, a_t)$
  - Update Q-table

Q-Learning is an **off-policy** algorithm, because the updated policy is different from the behavior policy. An issue that arises with this Q-table setup is scalability, because not all problems have discrete state-action spaces or simply too many states (e.g., Go, self-driving). A way to handle this issue is using function approximators in the form of neural networks for example to represent $Q(s, a)$.

4.3 Deep Q-Learning

A downside of the basic Q-Learning version we looked at is that it is only applicable to simple, discrete and low-dimensional state spaces. If we want to expand this approach to higher dimensional state spaces, we need to modify it. Instead of using a table, we now let a deep neural network with weights θ as a function approximator to estimate Q, $Q(s, a; \theta) \approx Q^*(s, a)$. There are two possible versions: The neural network can either take a state-action pair and output the corresponding Q value or it can only take as input a state and output the Q value of all the different actions. The former tends to be impractical, because we have to maximize over the actions and then would have to search for all possible actions - the second version is more convenient in that regard.

The training of the network comprises a forward and backward pass. In the forward pass, we use as loss function the mean-squared error of the Q values:

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(\underbrace{r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta)}_{\text{target}} - \underbrace{Q(s_t, a_t; \theta)}_{\text{prediction}}\right)^2\right] \tag{42}$$

The target consists of the current reward plus the expected future reward and the prediction aims to minimise the discrepancy between the two.

In the backward pass the gradient update of the parameters θ of the Q function then looks like this, which can be optimized via stochastic gradient descent:

$$\nabla_\theta \mathcal{L}(\theta) = \nabla_\theta \, \mathbb{E}\left[\left(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) - Q(s_t, a_t; \theta)\right)^2\right] \tag{43}$$

This optimization problem is very instable however and does not converge in practice. Two overcome this problem two things are needed.

4.3.1 Experience Replay

The first thing to help with the optimization of the Q network is called **experience replay**. We often use mini-batches to speed up training, but here the consecutive samples make learning inefficient, because they are strongly correlated. What experience replay does is to store the agents experiences at each time-step. This is done by continual updates to a **replay memory** D with new experiences made up of tuples $e_t = (s_t, a_t, r_t, s_{t+1})$. Then, during training, samples are drawn uniformly at random from this replay memory, which breaks the correlation between the samples and improves data efficiency, since samples can be used multiple times. In practice a circular replay memory is used, where old experiences (the oldest or random ones) are slowly deleted as new experiences are added - the memory size is thus fixed.

4.3.2 Fixed Q Targets

The second thing to help with the optimization of the Q network is fixing the Q targets. The problem we are facing is that the targets are non-stationary, we have to hunt a moving target, which changes with each change to our policy. This can easily lead to an oscillating or diverging behaviour. The process is stabilized by by using fixed Q targets. This is accomplished by introducing a second Q network with parameters θ^- called the target network, which is a copy of the regular Q network and only updated (by cloning Q) every C steps. This effectively reduces oscillation of the policy by adding a delay:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim U(D)}\left[\left(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta)\right)^2\right] \tag{44}$$

4.3.3 Putting it together

Now, when we put everything together, deep Q-Learning looks something like this: First, we take an action a_t according to the ϵ-greedy policy. We then store the transition e_t in the replay memory D. Next, a random mini-batch of transitions is sampled from D and we compute the Q targets using the old parameters θ^-. Finally we optimize the Q targets and Q network predictions via the following formula using stochastic gradien descent:

$$\mathcal{L}(\theta) = \mathbb{E}_{s_t,a_t,r_t,s_{t+1}\sim D}\left[\left(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta)\right)^2\right] \tag{45}$$

What this algorithm is capable of is demonstrated in a case study of human level control, where deep reinforcement algorithms were able to learn Atari Games such as the breakout game where bricks are successively removed by hitting them with a moving ball (this increases the score) which bounces off a bar which can be moved left or right by the agent. The optimal strategy it finds is creating a small hole that ball can move through such that the ball now bounces between the ceiling and bricks, removing them very fastly.

4.3.4 Shortcomings

Despite the above mentioned improvements, deep Q-Learning suffers from several shortcomings:

- The training times are notoriously long, since the rewards occur very sparsely, but the neural networks require many gradient updates
- Due to the uniform sampling from the replay buffer, all transitions are treated as equally important, even though certain transitions may be much more informative and crucial than others, rendering them underrepresented.
- The exploration strategy is simplistic.
- Fully-observable states are required.
- There can only be a discrete set of actions.

To address some of these downsides, many people have come up with improvements to the original version of the algorithm.

4.3.5 Deep Deterministic Policy Gradients

Deep Deterministic Policy Gradients (DDPG) have been invented to counteract the issue of discrete action spaces. The problem is that in order to find a continuous action, optimization at every time-step is needed. We tackle this by using two networks: One is called the **actor** (illustrated on the left of Fig. 46), which follows a deterministic policy μ and the network with parameters θ^μ maps from a state to an action. The other is called the **critic** (illustrated on the right of Fig. 46), which takes a state and the deterministic action from the actor $a = \mu(s, \theta^\mu)$. What we are interested in is the actor, but the critic serves as an auxiliary model to obtain the actor. The critic, parametrized by θ^Q then outputs a Q value for the state-action pair.

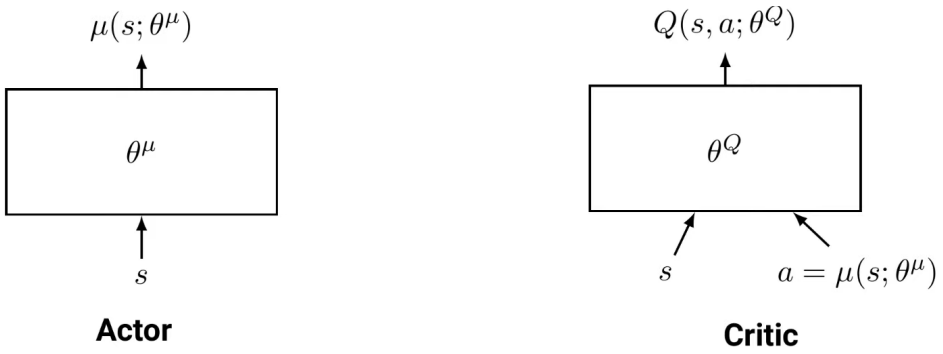


Figure 46: Left: Illustration of the actor network, right: illustration of the critic network.

In more detail, the actor estimates the deterministic policy μ in the direction that most improves Q. Next, we apply the chain rule to the expected return, which turns out to be the gradient of the policy μ:

$$\nabla_{\theta^\mu} \mathbb{E}_{s_t,a_t,r_t,s_{t+1}\sim D}\left[Q(s_t, \mu(s_t;\theta^\mu);\theta^Q)\right] = \mathbb{E}\left[\nabla_{a_t} Q(s_t, a_t;\theta^Q)\, \nabla_{\theta^\mu}\mu(s_t;\theta^\mu)\right] \tag{46}$$

The critic estimates can be learned via the BOE introduced in 4.2.3:

$$\nabla_{\theta^Q} \mathbb{E}_{s_t,a_t,r_t,s_{t+1}\sim D}\left[\left(r_t + \gamma Q(s_{t+1}, \mu(s_{t+1};\theta^{\mu-});\theta^{Q-}) - Q(s_t, a_t;\theta^Q)\right)^2\right] \tag{47}$$

We circumvent the need to maximize over actions, because this is already taken care of by μ. So now we can work on a continuous action space.

Both experience replay and target networks can be used to stabilize training. The target networks may use soft target updates, where the weights are not just cloned, but rather slowly adapted via:

$$\theta^{Q-} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q-} \tag{48}$$
$$\theta^{\mu-} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu-} \tag{49}$$

with $0 < \tau \ll 1$ controlling the trade-off between speed and stability of learning. Exploration is accomplished by adding noise to the policy: $\mu(s;\theta^\mu) + \mathcal{N}$

4.3.6 Prioritized experience replay

Another improvement one may add is **prioritized experience replay**, where important transitions are replayed more often. The priority δ is determined by the magnitude of TD error:

$$\delta = \left|r_t + \gamma \max_{a'} Q(s_{t+1}, a';\theta^{Q-}) - Q(s_t, a_t;\theta^Q)\right| \tag{50}$$

Thus, instances with a high TD error are replayed more frequently with the hopes of being able to minimize these large errors. Intuitively transitions with high TD errors indicate how unexpected by the agent the transition was. This serves to speed-up learning by a factor of two. Now at the end, we will take a look at some other approaches deep RL has to offer.

4.3.7 Asynchronous Deep Reinforcement Learning

The idea in **asynchronous deep reinforcement learning** is to let multiple agents interact with their own environment copies and collect experiences. The agents can have different exploration policies in order to maximize experience diversity. All these models jointly update a shared global model. This has a similar effect as experience replay, because it decorrelates samples and it also speeds up training, due to the parallel training of multiple agents.

4.3.8 Bootstrapped DQN

To increase the efficiency of exploration, **bootstrapping** can be used. Multiple bootstrapped heads approximate a distribution over Q values. With each new training epoch, one of the heads Q_k is randomly selected; at the end of training a single ensemble policy is build by combining all heads. This adds robustness compared with the approach of a single Q function.

4.3.9 Double Q-Learning

In **double Q-Learning**, the Q function is decoupled for the selection and evaluation of the actions. This avoids overestimating Q and stabilizes training. The previous target is replaced with:

$$DoubleDQN \quad : \quad r_t + \gamma Q(s_{t+1}, \underset{a'}{\text{argmax}}\ Q(s_{t+1}, a'; \theta); \theta^-) \tag{51}$$

So we are not taking the maximum over the actions anymore, but evaluating Q with parameters θ^- and taking the action that maximizes the Q function for parameters θ. The weights θ parametrize the online network used to determine the greedy policy. The weights θ^- parametrize the target network, which determines the corresponding action value.

4.3.10 Deep Recurrent Q-Learning

Yet another approach adds recurrency to DQNs to be able to better handle partial observability of states. Here, one of the fully connected layers may simply be replaced with an LSTM layer, which can memorize observations from early on.

4.3.11 Faulty Rewards

One important aspect to keep in mind for reinforcement problems is that the choice of the reward functions has a very big impact on learning. This means, they have to be defined in a sensible manner to achieve good results. Certain desired behaviours may be obvious, in the context of self-driving this might include for example safety, efficiency and comfort. Still, other reward-worthy behaviours may be less obvious and it is also not trivial how to balance the different rewards.

5 Vehicle Dynamics

While it is possible to control a vehicle by mapping directly from images, it is often much easier to design a functional controller, if we understand the underlying system well. This includes concepts such as vehicle kinematics and dynamics. This chapter is mostly based on parts of the lecture "Vehicle Dynamics & Control" by Prof. Schildbach from the University of Lübeck.

First, we are going to motivate the importance of vehicle dynamics and introduce some geometric and physical concepts. The second part covers the kinematic bicycle model, which is the most basic vehicle model. Next, tires are introduced to the model, which changes much of the simple behavior we have seen before. We conclude by introducing the dynamic bicycle model, which also takes kinetic concepts like dynamics, forces and friction into consideration.

5.1 Introduction to Vehicle Dynamics

When modeling vehicle dynamics, one has to distinguish between kinematics and kinetics. Kinematics describes the motion of points and bodies. It considers position, velocity and acceleration among others. Examples include celestial bodies, particle systems, robotic arms and the human skeleton. The simple kinematic bicycle model introduced in Section 5.2 is purely based on kinematics. Kinetics, on the other hand, describes causes of motion as well as effects of forces and moments. It will become important for the tire models and the dynamic bicycle model. Much of kinetics is based on Newton's laws, e.g. the famous equation

$$F = ma \tag{52}$$

5.1.1 Holonomic and Non-Holonomic Constraints

Constraints on the configuration space are an important concept for vehicle dynamics. They are called holonomic constraints. Assume a particle in three dimensions $(x, y, z) \in \mathbb{R}^3$. A simple example is constraining the particles to the x/y plane by setting the z-coordinate to zero. This can be expressed as a function of the configuration of the particle being equal to zero:

$$z = 0 \tag{53}$$

$$\Leftrightarrow \quad f(x, y, z) = 0 \quad \text{with} \quad f(x, y, z) = z \tag{54}$$

Constraints of the form $f(x, y, z) = 0$ are called holonomic constraints. They constrain the configuration space, in this case to the x/y plane. However, the system can move freely in that space. This means that the controllable degrees of freedom equal the total degrees of freedom, which is two in this case.

Non-Holonomic constraints are constraints on the velocity. Assume for example a vehicle that is parameterized by $(x, y, \psi) \in \mathbb{R}^2 \times [0, 2\pi]$. The first two variables are the x/y coordinates relative to the zero point of the two-dimensional coordinate system and the origin of the vehicle is set as fixed point inside of it, such as the center of the rear axle. The third variable is the orientation of the vehicle. It is the angle between the forward direction of the vehicle and the x-axis of the coordinate system.

Given such a model, the 2D vehicle velocity can be computed based on the velocities along the x- and y-directions and the steering angle. The velocities along the axes are equal to the respective first time derivatives.

$$\dot{x} = v \cos(\psi) \tag{55}$$

$$\dot{y} = v \sin(\psi) \tag{56}$$

$$\tag{57}$$

By combining both equations, we derive the following constraint on the velocity:

$$\dot{x} \sin(\psi) - \dot{y} \cos(\psi) = 0 \tag{58}$$

This non-holonomic constraint cannot be expressed in the form $f(x, y, \psi) = 0$. The car cannot freely move in any direction as for example no sliding sideway motions are allowed. Only movements along the vehicle's forward axis in combination with the steering angle are allowed. This is illustrated in Fig. 47 on the right. There are two controllable degrees of freedom, namely the velocity v and the steering ψ. This is less than the total degrees of freedom, which is three, because there is a dependency between them. The constraint restricts the velocity space, but not the configuration space, because every point is accessible, just not by every movement.

To summarize holonomic systems constrain configuration space. We can freely move in any direction. The controllable degrees of freedom equal to total degrees of freedom and the constraints can be described

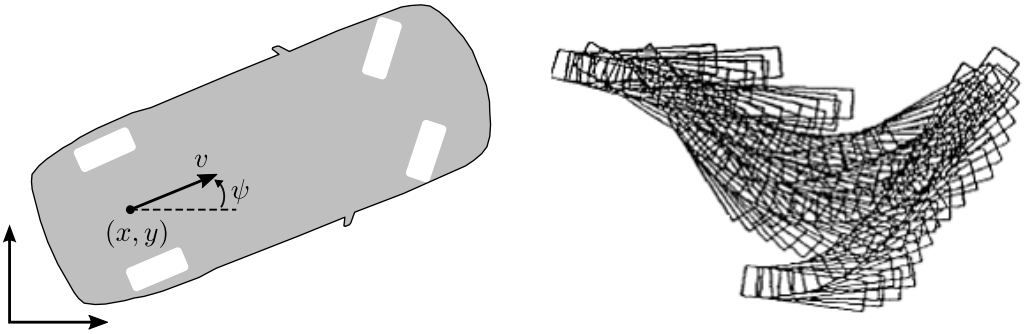


Figure 47: **Non-holonomic Vehicle** An example of a non-holonomic constraint based on the x- and y-coordinates as well as the steering angle.

by $f(x_1, \ldots, x_N) = 0$. In contrast, in non-holonomic systems that constrain velocity space we cannot freely move in any direction. The controllable degrees of freedom are less than total degrees of freedom. Holonomic constraints cannot be described by an equation $f(x_1, \ldots, x_N) = 0$. A typical example is the car seen above, whose movements are constrained by the velocity and steering angle.

In praxis, many system can be subject to both holonomic and non-holonomic constraints. Consider for example a robot, that can only move on a plane, which is a holonomic constraint. Its exact movements are constrained by the angles of its joints, which is a non-holonomic constraint.

5.1.2 Coordinate Systems

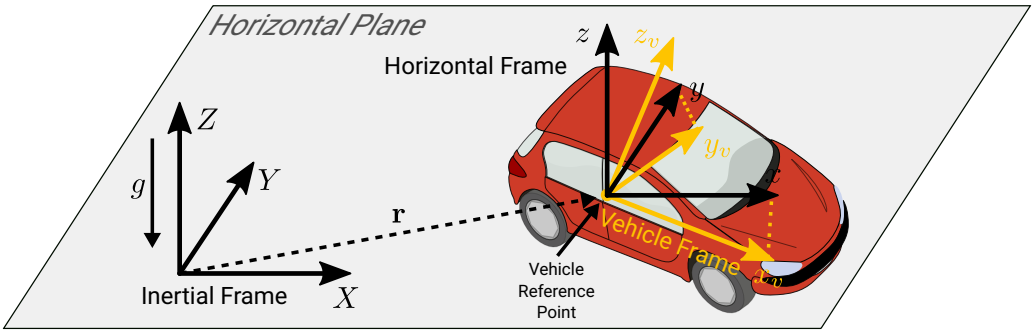


Figure 48: **Coordinate Systems** Relationship between the inertial, vehicle and horizontal frames in a coordinate system.

In coordinate systems we consider three different coordinate system as shown in Fig. 48. The first one, called the inertial frame, is fixed to the earth with a vertical Z-axis and an X/Y horizontal plane. It is also called the world coordinate system.

The second on is the vehicle frame. It is attached to the vehicle at a fixed reference point. Reference points might be the middle point of the rear axis or the vehicle's center of mass. x_v points towards the front, y_v to the side and z_v to the top of the vehicle. This is specified in ISO 8855.

Finally, the horizontal frame also has its origin at the vehicle reference point just like the vehicle frame. However, its x-
and y-axes are projections of the x_v- and y_v-axes onto the X/Y horizontal plane. This removes any rotation outside of the X/Y plane.

5.1.3 Kinematics

We start introducing kinematics by first considering a simple point P in three dimensional space as subject. Its position $\mathbf{r}_P(t) \in \mathbb{R}^3$ time $t \in \mathbb{R}$ is given by three coordinates. The velocity and acceleration are the first and second derivatives of the position $\mathbf{r}_P(t)$. This is illustrated in Fig. 49.

$$\mathbf{r}_P(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix} \qquad \mathbf{v}_P(t) = \dot{\mathbf{r}}_P(t) = \begin{pmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{z}(t) \end{pmatrix} \qquad \mathbf{a}_P(t) = \ddot{\mathbf{r}}_P(t) = \begin{pmatrix} \ddot{x}(t) \\ \ddot{y}(t) \\ \ddot{z}(t) \end{pmatrix}$$

The simple point based kinematics model can be expanded to more complex objects. A rigid body refers to a collection of infinitely many infinitesimally small mass points which are rigidly connected, i.e., their relative

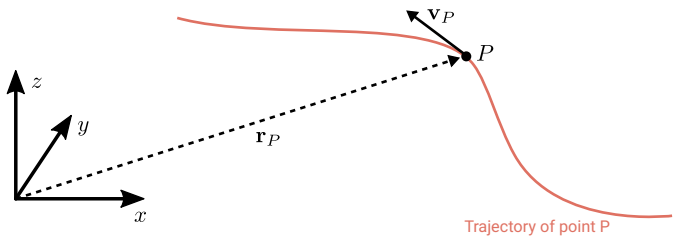


Figure 49: **Kinematics of a Point** The model's velocity and acceleration are given by the first and second time derivative of the point's position. This position is given as the trajectory in red.

position remains unchanged over time. It's motion can be compactly described by the motion of an arbitrary reference point C of the body plus the relative motion of all other points P with respect to C. The reference point is usually chosen to be the same as the center of the horizontal and vehicle frames.

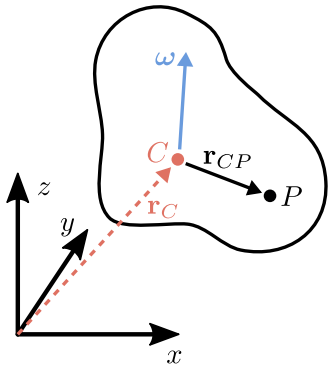


Figure 50: **Kinematics of a Rigid Body** The kinematics of points of a rigid body based on a reference point, velocity and angular velocity.

This concept is further visualized in Fig. 50. Let C an arbitrary reference point fixed to the rigid body and P an arbitrary point on the rigid body, whose movements we want to model. We call the angular velocity of our rigid body $\boldsymbol{\omega}$. The position of P depends on the position of the reference point in the inertial frame and the position of P in the vehicle frame relative to C:

$$\mathbf{r}_P = \mathbf{r}_C + \mathbf{r}_{CP} \tag{59}$$

Similarly, the point's velocity is given by the reference point's velocity modified by the movement of P regarding the angular velocity in the vehicle frame:

$$\mathbf{v}_P = \mathbf{v}_C + \boldsymbol{\omega} \times \mathbf{r}_{CP} \tag{60}$$

Due to rigidity, points P can only rotate wrt. C. Thus a rigid body has six three degrees of freedom given by the position of C and another three dimensions given by the rotations of P along the three coordinate axes. This makes a total of six degrees of freedom.

At each time instance $t \in \mathbb{R}$, there exists a particular reference point O for which $\mathbf{v}_O(t) = 0$. This is called the instantaneous center of rotation. Each point P of the rigid body performs a pure rotation about O:

$$\mathbf{v}_P = \mathbf{v}_O + \boldsymbol{\omega} \times \mathbf{r}_{OP} = \boldsymbol{\omega} \times \mathbf{r}_{OP} \tag{61}$$

We will illustrate this concept with two examples shown in Fig. 51. The first example considers a turning wheel, which is completely lifted off the ground and therefore fixed. The wheel does not move in x or y direction and the angular velocity vector $\boldsymbol{\omega}$ points into the x/y plane. Obviously, the instantaneous center of rotation is given by the center of the wheel. The velocity of any point P on the turning wheel with radius R is given as:

$$v_P = \omega R \tag{62}$$

The farther away of the center of rotation, the faster the velocity gets. At O itself, the velocity is zero. The velocity increases linearly with the distance to the center point.

The second example considers a rolling wheel. Since we are considering a purely kinematic and not dynamic or kinetic model, the wheel is rolling on the ground without slip. The ground is fixed in the x/y plane. Since

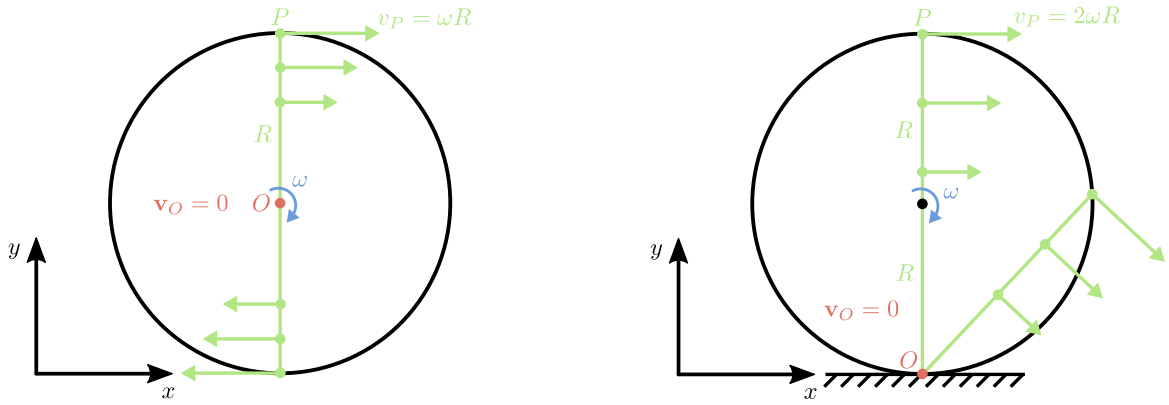


Figure 51: **Instantaneous Center of Rotation** Examples of instantaneous centers of rotation based on a rolling and turning wheel.

the ground is fixed, only the wheel can move. The angular velocity vector ω points into x/y plane. In this case, the velocity of point P with radius R is given as:

$$v_P = 2\omega R \tag{63}$$

The doubling of the speed given the angular velocity compared to the first example can be explained by the instantaneous center of rotation. Since the rotation is not relative to the center of the wheel, but to the contact point, the radius is doubled. However, this is only true for this very moment of time t. Any so tiny moment later, the contact point moves along with the wheel on the ground and changes.

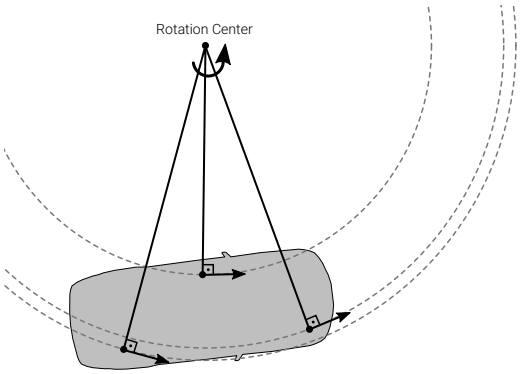


Figure 52: **Rigid Motion** Each point of the vehicle moves along a circular trajectory relative to an instantaneous center of rotation.

5.2 Kinematic Bicycle Model

The kinematic bicycle is both the most prominent and simple vehicle model. It makes many simplifying assumptions among which the absence of wheel slip is the most important one. This means that wheels cannot move sideways, but the orientation of the wheels is always in the velocity direction of wheels. This can only be realistically applied to situations with small speeds. At high speeds this model is less reliable.

The vehicle is a rigid body as introduced in the last section. For each motion it rotates around a rotation center as illustrated in Fig. 52. This means that different points on the rigid body move along different circular trajectories. The velocity vectors of these points are tangential to the circular trajectories and as such orthogonal to the line to the center of rotation.

To reduce the complexity, the four wheels of a real vehicle are approximated with two imaginary wheels at the centers of the axles. This gave the kinematic bicycle model its name. While it is an approximation that fails in cases where we want to control each wheel individually, it works fairly well in a general modeling sense in most situations.

The full kinematic bicycle model is illustrated in Fig. 53. It makes two strong assumptions: first of all, it assumes there is only planar motion and no roll or pitch. The only rotation is controlled by the heading angle ψ. It is the vehicle's direction as angle between the vehicle frame and horizontal frame. Second and more

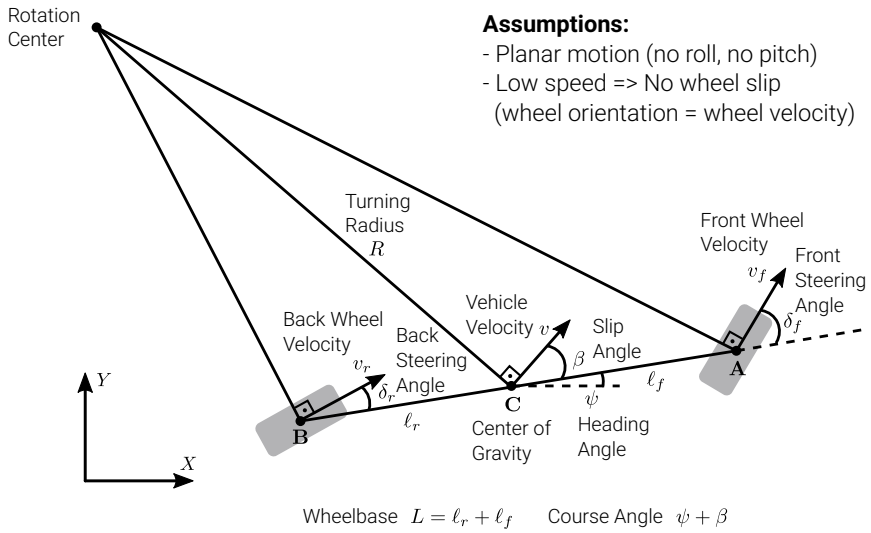


Figure 53: **Kinematic Bicycle Model** The kinematic bicycle model with two imaginary wheels.

important, it assumes there is no wheel slip. This is only a valid assumption at low speed. This means the orientation of the wheels is into the direction of the velocity of the points A and B at the center of the wheels. These velocities are denoted as v_f and v_r.

Since we assume there is no wheel slip, the velocity of the rear and front wheels are in the direction of the wheels. That means that the wheels are also orthogonal to the lines connecting the rotation center and the center of the wheels. However, the vehicle itself can indeed have a slip angle called β. In this case, the direction of the velocity at the center of gravity v does not point in the direction of the vehicle, but just as the wheels, it moves at a vector tangential to the circular trajectory relative to the rotation center.

The slip angle depends on the front and back steering angles δ_f and δ_r. These angles are relative to the vehicle orientation. For large steering angles, the vehicle turns tightly and has a large slip angle, but small turning radius. If the vehicle drives in an almost straight line, the steering and slip angles are small and the turning radius tends towards infinity. The course of the velocity vector v relative to the horizontal frame is given as the sum of heading and slip angle. Finally, there are the distances between the centers of the wheels and the center of gravity. They are called l_f and l_r and together, they form the wheelbase L.

We want to derive the motion equations of the kinematic bicycle model to reason how the vehicle moves over time. This means we have to learn the time derivatives of the position relative to the x- and y-axis of the inertial frame and the time derivative of the heading angle, which is also called the angular velocity.

The velocity relative to x y can easily be computed by using the sine and cosine formulas in the rectangular triangle between the x-axis and the vector of the vehicular velocity. Unfortunately, the formula of the angular velocity is fairly complicated.

$$\dot{X} = v\cos(\psi + \beta) \tag{64}$$

$$\dot{Y} = v\sin(\psi + \beta) \tag{65}$$

$$\dot{\psi} = \frac{v\cos(\beta)}{\ell_f + \ell_r}\left(\tan(\delta_f) - \tan(\delta_r)\right) \tag{66}$$

$$\beta = \tan^{-1}\left(\frac{\ell_f \tan(\delta_r) + \ell_r \tan(\delta_f)}{\ell_f + \ell_r}\right) \tag{67}$$

Instead, we consider a simpler version of the kinematic bicycle model with a fixed rear wheel. While the ability to move all wheels allows more dynamic maneuvers and to counter some drift, most cars on the market only allow front steering. Since the new model only has one steering angle at the front, we rename δ_f to just δ.

The velocity in x and y directions stay the same, but the angular velocity is much easier to derive. In the left red triangle in Fig. 54, the front steering angle δ also appears at the rotation center, because of the orthogonality of the wheel velocities. Therefore, we can derive the tangent of δ in the rectangular triangle as the division between the wheel base and the distance of the rear wheel to the center of rotation:

$$\tan\delta = \frac{l_f + l_r}{R'} \tag{68}$$

Rearranging gives the following formula:

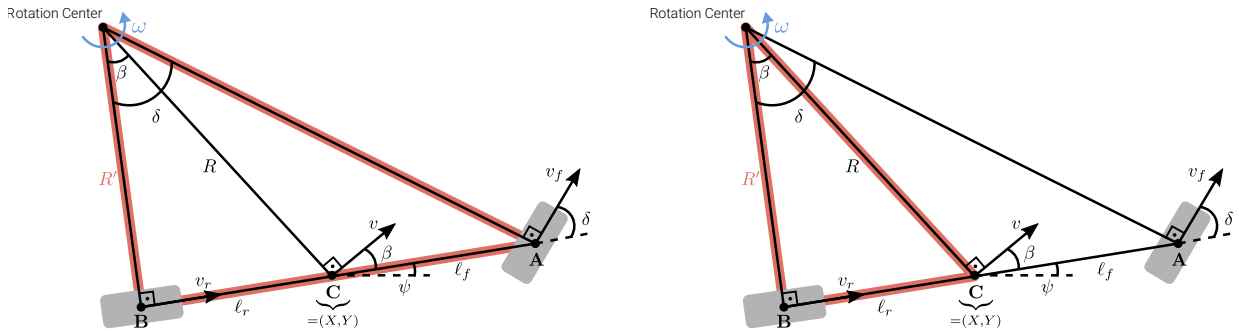


Figure 54: **Front Steering Only** Trigonometric derivation of the angular velocity in the kinematic bicycle model with front steering only.

$$\frac{1}{R'} = \frac{\tan\delta}{l_f + l_r} \tag{69}$$

Next, we consider the rectangular triangle given by the center of rotation, center of gravity and rear wheel. Because of orthogonality, the angle slip angle β also appears at the rotation center. Therefore, we can compute the tangent of delta based on the rear length and distance to the rotation center:

$$\tan\beta = \frac{l_r}{R'} = \frac{l_r \tan\delta}{l_f + l_r} \tag{70}$$

By plugging in the upper formula for $\frac{1}{R'}$ into the last equation and solving for the slip angle, we get the formula for the slip angle β. We continue on the right triangle by computing a formula for $\frac{1}{R}$ based on the cosine:

$$\cos\beta = \frac{R'}{R} \tag{71}$$

$$\Rightarrow \quad \frac{1}{R} = \frac{\cos\beta}{R'} \tag{72}$$

The circular velocity is simply the vehicle velocity divided by the turning radius. Plugging in the two results from above, we get the following tractable equation:

$$\dot{\psi} = \omega = \frac{v}{R} \tag{73}$$

$$= \frac{v\cos(\beta)}{R'} \tag{74}$$

$$= \frac{v\cos(\beta)}{l_f + l_r}\tan(\delta) \tag{75}$$

$$\tag{76}$$

This gives the following set of motion equations for the kinematic bicycle model with front steering only. By integrating these motion equations, we can follow the trajectory of the vehicle.

$$\dot{X} = v\cos(\psi + \beta) \tag{77}$$

$$\dot{Y} = v\sin(\psi + \beta) \tag{78}$$

$$\dot{\psi} = \frac{v\cos(\beta)}{\ell_f + \ell_r}\tan(\delta) \tag{79}$$

$$\beta = \tan^{-1}\left(\frac{\ell_r \tan(\delta)}{\ell_f + \ell_r}\right) \tag{80}$$

When we are driving almost straigt, the β and steering angle δ become very small. In those situations β can be ignored in the x and y velocities, $\cos(\beta)$ becomes 1 and $\tan(\delta)$ is roughly linear to δ itself. Therefore, the motion equations can be simplified in the following way:

$$\dot{X} = v\cos(\psi) \tag{81}$$
$$\dot{Y} = v\sin(\psi) \tag{82}$$
$$\dot{\psi} = \frac{v\delta}{\ell_f + \ell_r} \tag{83}$$

For a time discretized version of the model the configuration can be computed as the previous state modified by the direction of the velocity times the length of a time step. This is how the model would be simulated on a computer, which does not allow true continuous behavior.

$$X_{t+1} = X_t + v\cos(\psi)\,\Delta t \tag{84}$$
$$Y_{t+1} = Y_t + v\sin(\psi)\,\Delta t \tag{85}$$
$$\psi_{t+1} = \psi_t + \frac{v\delta}{\ell_f + \ell_r}\,\Delta t \tag{86}$$

5.2.1 Ackermann Steering Geometry

A problem of the kinematic bicycle model is the approximation using only two wheels centered at the axles. In practice, the left and right wheel steering angles are not equal if no wheel slip is allowed. The reason is that they are both different points of the rigid body and are on different circular trajectories around the rotation center. Another problem is that the left and right rear wheels do not have the same speed. This has to corrected with a differential. The combination of admissible steering angles is called the Ackerman steering geometry. If angles are small, then the left/right steering wheel angles can be approximated as follow:

$$\delta_l \approx \tan\left(\frac{L}{R + 0.5B}\right) \approx \frac{L}{R + 0.5B} \tag{87}$$
$$\delta_r \approx \tan\left(\frac{L}{R - 0.5B}\right) \approx \frac{L}{R - 0.5B}, \tag{88}$$

where δ_l and δ_r are the right and left steering angles, L is the wheelbase and B is the track between left and right wheels.

5.3 Tire Models

In real life, the assumption of the kinematic bicycle that there is no wheel slip, is not true for even moderate speeds or terrain with less grip. In fact, wheels are always sliding at least a little bit. In order to understand what slip is and the relationship between slip and forces, we need to understand tires. Tire models describe the lateral and longitudinal forces at the tires. There exist many different tire models at various levels of complexity. For a simple qualitative description we consider the tread block model.

5.3.1 Tread Block Model

We consider a tire model, where the tire is composed of many little tread blocks, hence the name. Each tread block is a little piece of rubber attached to the tire. Three different stages of the forces between the contact patch with its tread blocks and the ground are shown in Fig. 55.

As soon as the wheel is driven externally, the tire tread blocks start deforming and slipping, as shown at the first stage. The tire tread blocks adhere to the ground, deform and slip when loosing contact. They deform, because as soon as we apply a circumferential force to the wheel, the tread blocks start deforming and slipping, when losing contact to the ground. The farther to the back, the larger the force on the treads of the contact patch becomes. Once a tread block leaves the contact patch, it snaps back into its original shape and position. This causes them to slip. When the driving force increases and static friction is exceeded, the blocks slip earlier, as shown in the second force graph. As sliding friction is smaller than static friction, this decreases the transmitted driving force. The last graph shows that if the tire tread blocks start even sliding at the beginning, only sliding friction can be applied.

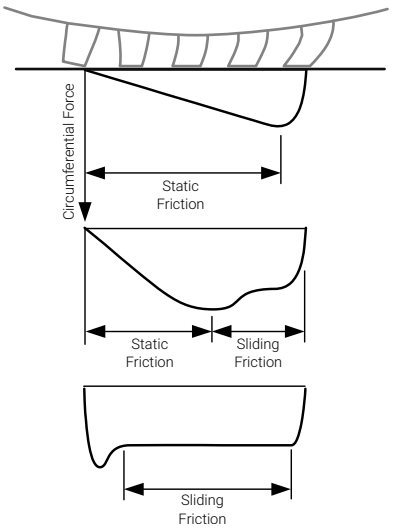


Figure 55: **Tread Block Model** Static and sliding friction at three different stages of the tread block model.

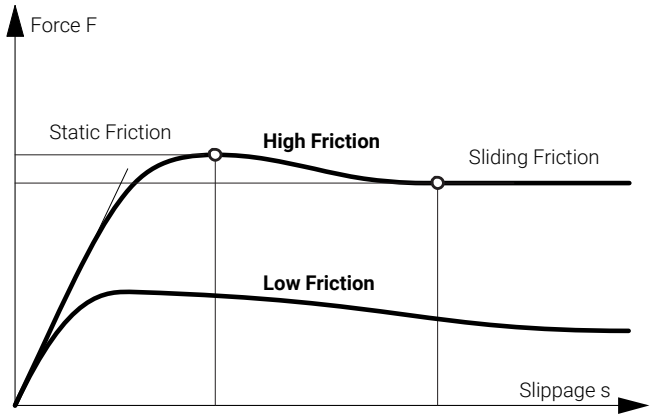


Figure 56: **Force vs. Slip** Plot of the longitudinal force depending on slip for both low and high friction.

Slippage is the difference between surface speed of the wheel and vehicle speed. If the wheel would be in perfect contact with the ground, the slippage would be zero, because the wheel and vehicle speed were identical. That happens, when you are driving with a constant velocity without any forces acting on the vehicle.

The force F grows linearly with the slippage s in the beginning. This is the area where the tread block a linearly deforming and not sliding much. Once we reach static friction and the slippage gets too large, this leads to a reduction of the force applied to the street and we are in a region of sliding friction. Thus, large slippage s leads to a reduction of F because the sliding friction is smaller than the static friction. This sliding can be reduced by systems such as ABS.

Let us consider the force curve for low friction, for example when driving on slippery terrain. The start of the curve does not change as the elasticity of the blocks remains the same. However, the maximum reduces due to the decreased static friction, i.e., the tread blocks start sliding earlier due to a decrease in friction.

So far we only considered the longitudinal force on tires. However, the lateral force is also relevant when steering. The lateral force F_y is analogous to longitudinal force but blocks move laterally now. For small s and α the forces are given by:

$$F_y = c\,s = c \cdot \tan(\alpha) \approx c \cdot \alpha, \tag{89}$$

where v is the wheel velocity, v_x the longitudinal velocity, v_y the lateral velocity and c the cornering stiffness, which is a constant. As can be seen in Fig. 57 the force curve is very similar to the longitudinal case.

The longitudinal and lateral forces can be combined in a circle of forces. Both lateral force F_x and longitudinal force F_y force cannot exceed the maximum friction force $\|F\|_{max}$. It follows that more longitudinal force implies less lateral force. The maximum acceleration can only be reached for straight driving. Also, the circle of forces explains why it is so difficult to steer, when braking hard. Since the wheels start to slip, the sliding friction is reached in longitudinal direction and it is not possible to add any force in lateral direction. Another application of this model is to make statements about maximal possible vehicle accelerations.

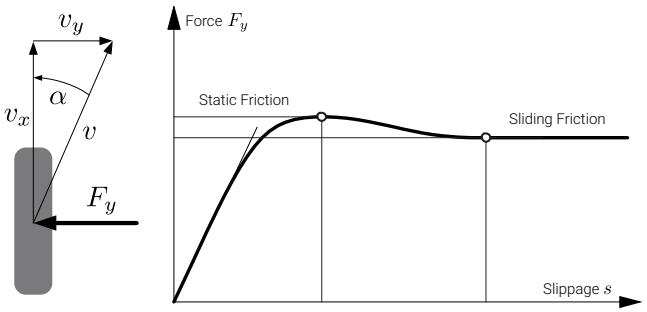


Figure 57: **Lateral** Plot of the lateral force depending on slip.

5.4 Dynamic Bicyle Model

After the introduction of tire models, we are know ready to define the dynamic bicycle model. In this section we focus on a simple version, which only considers lateral wheel slip.

5.4.1 Dynamics of a Rigid Body

Consider a point P with mass m in $\mathbb{R}^3$. Let $\mathbf{r}_P(t) \in \mathbb{R}^3$ be its position in an inertial reference frame. We want to model the translatory motion of this point. Let $\mathbf{v}_P(t)$ denote its velocity and $\mathbf{a}_P(t)$ its acceleration. The linear momentum of P is defined as

$$\mathbf{p}_P(t) = m\mathbf{v}_P(t) \tag{90}$$

By Newton's second law we have

$$\frac{d}{dt}\mathbf{p}_P(t) = m\,\mathbf{a}_P(t) = \mathbf{F}_{net}(t) = \sum_i \mathbf{F}_i(t), \tag{91}$$

where $\mathbf{F}_i(t)$ represents all forces acting on the point mass P.

Instead of modeling points, we want to compute the translatory motion of a rigid body. Consider a rigid body B with mass m in $\mathbb{R}^3$. Let $\mathbf{r}_C(t) \in \mathbb{R}^3$ be the position of its center of gravity C. Let $\mathbf{v}_C(t)$ denote its velocity and $\mathbf{a}_C(t)$ its acceleration. The linear momentum of B is defined as

$$\mathbf{p}_B(t) = m\mathbf{v}_C(t) \tag{92}$$

The center of gravity of a rigid body behaves like a point mass with mass m and as if all forces act on that point.

$$\frac{d}{dt}\mathbf{p}_B(t) = m\,\mathbf{a}_C(t) = \mathbf{F}_{net}(t) = \sum_i \mathbf{F}_i(t) \tag{93}$$

where $\mathbf{F}_i(t)$ represents all forces acting on the rigid body B.

The rotatory motion of a rigid body is more complex and we cannot simply treat it like a point as in the translatory case. For the **rotatory motion,** also the geometric shape of B
and the spatial distribution of its mass is important. Let $\rho(x, y, z)$ be the body's density function. This allows us to compute the body's mass:

$$m = \int_B \rho(x, y, z)\, dx\, dy\, dz = \int_B dm \tag{94}$$

This mass function is also illustrated in Fig. 58. A little, incremental mass element in blue symbolizes a small fraction of the entire body and the mass of such a little spatial extent can be computed using the density function.

The inertia tensor of B is defined as the moments of inertia on the diagonal and the moments of deviation at the other positions:

$$\boldsymbol{\Theta} = \begin{bmatrix} I_x & I_{xy} & I_{xz} \\ I_{yx} & I_y & I_{yz} \\ I_{zx} & I_{zy} & I_z \end{bmatrix} \quad \underbrace{\begin{matrix} I_x = \int_B (y^2 + z^2)\, dm \\ I_y = \int_B (x^2 + z^2)\, dm \\ I_z = \int_B (x^2 + y^2)\, dm \end{matrix}}_{\text{moments of inertia}} \quad \underbrace{\begin{matrix} I_{xy} = I_{yx} = -\int_B xy\, dm \\ I_{xz} = I_{zx} = -\int_B xz\, dm \\ I_{yz} = I_{zy} = -\int_B yz\, dm \end{matrix}}_{\text{moments of deviation}}$$

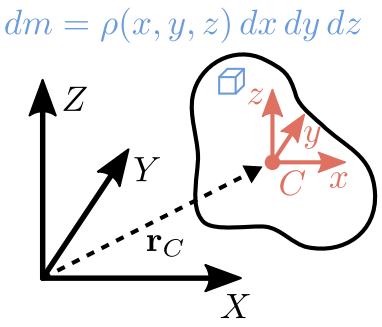


Figure 58: **Mass of Rigid Body** Mass function of a rigid body.

After defining the inertia tensor of B, we can compute its rotatory motion. Let $\boldsymbol{\omega} = (\omega_x\ \omega_y\ \omega_z)^\top$ be the vector of angular velocities. The rigid body rotates around this vector and rotational velocity is equal to the magnitude of this vector. Then the angular momentum $\mathbf{L}_C$ of the rigid body B is given as:

$$\mathbf{L}_C = \boldsymbol{\Theta}\,\boldsymbol{\omega} \tag{95}$$

Similar to the linear motion, we can apply the angular momentum principle. It states that the time derivative of the angular momentum is equal to the inertia tensor times the angular acceleration, which is equal to the net moments of all forces acting on the rigid body:

$$\frac{d}{dt}\mathbf{L}_C(t) = \boldsymbol{\Theta}\,\dot{\boldsymbol{\omega}} = \mathbf{M}_{net}(t) = \sum_i \mathbf{M}_i(t), \tag{96}$$

where $\mathbf{M}_i(t)$ are the moments of all forces acting on B with respect to the center of gravity C. This is illustrated in Fig. 59.

Figure 59: **Angular Velocity and Momentum** The rotational velocity based on a vector of angular velocties and the angular momentum principle.

If the body frame is chosen as a principal axis system for the rigid body, which are the symmetry axes, then the inertia tensor is diagonal. This greatly simplifies the model.

$$\boldsymbol{\Theta} = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix}$$

For the planar motion of a rigid body in the x/y-plane two of the angular velocities and their respective momenta are zero:

$$\omega_x = \omega_y = 0 \quad \text{and} \quad M_x = M_y = 0 \tag{97}$$

Hence the angular momentum has a much simpler form and becomes $L_z = I_z\,\omega_z(t)$ and the angular momentum principle yields the following relationship:

$$I_z\,\dot{\omega}_z = \sum_i M_i \tag{98}$$

5.4.2 Dynamic Bicycle Model

Now that we have introduced the dynamics of a rigid body, we are able to present the dynamic bicycle model. While it is similar to the kinematic bicycle model, it also considers wheel slip and momenta. As such, it is much more realistic in a variety of driving situations. We only consider a model with a front steering and without a back steering angle as before. So the back wheel is aligned with the vehicle orientation.

Yet again, we have to make several assumptions. First, the vehicle's motion is restricted to the X/Y plane. The vehicle is considered as a rigid body. Only lateral tire forces, generated by a linear tire model, are considered. The steering angle δ is small:

$$\sin\delta \approx \delta \tag{99}$$
$$\tan\delta \approx \delta \tag{100}$$
$$\cos\delta \approx 1 \tag{101}$$

Finally, the assumption is that the longitudinal velocity v_x is constant. The entire model is illustrated in Fig. 60.

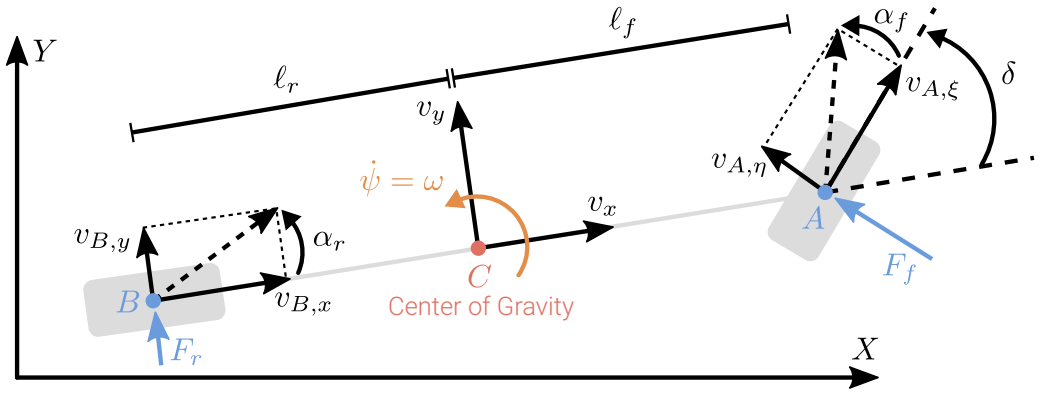


Figure 60: **Dynamic Bicycle Model** Illustration of the simple version of the dynamic bicycle model.

First, we consider the lateral or linear dynamics. Similar to before, we can compute the force in lateral direction using Newton's second law:

$$ma_y = \sum_i F_{y,i} = F_r + F_f \cos\delta \approx F_r + F_f \tag{102}$$

In this case, the force of the front wheel has to be adjusted based on the steering angle. Since the back wheel is not steering, this is not necessary for F_r. And based on our assumption that the steering angle is small, $\cos\delta \approx 1$. Similar, the lateral acceleration can also be defined via the time derivative in y direction and modified derivative in x direction based on the rotation defined by ω.

$$a_y = \dot{v}_y + \omega\, v_x, \tag{103}$$

where $\omega\, v_x$ is called centripetal acceleration. The time derivatives can now be plugged into the formula based on Newton's second law:

$$m(\dot{v}_y + \omega v_x) = F_r + F_f \tag{104}$$

The second equation we define is for the yaw dynamics. It is the body's rotational component.

$$I_z\, \dot{\omega} = \sum_i M_i = -l_r F_r + l_f F_f \underbrace{\cos\delta}_{\approx 1} \tag{105}$$
$$\Rightarrow \quad I_z\, \dot{\omega} = -l_r F_r + l_f F_f \tag{106}$$

Since the force at the back wheel acts against the rotation around the center of gravity, this momentum has to be subtracted. The force on the front, on the other hand, acts in the same direction as the angular velocity.

Next, we consider the tire forces again. From the previous section, we know that at the beginning of the curve, the forces are linear. They are linear, because they are the product of a cornering stiffness factor c_r and rear wheel slip angle α_r.

$$F_r = -c_r \alpha_r \approx -c_r \tan(\alpha_r) = -c_r \frac{v_{B,y}}{v_{B,x}} \tag{107}$$

$$v_{B,x} = v_x \quad v_{B,y} = v_y - \omega l_r \tag{108}$$

The velocity in x direction was assumed to be constant. However, the one in y direction has to be modified based on the rotational component. The equations for the front wheel are very similar, but the direction of the rotation has always be taken into consideration.

$$F_f = -c_f \alpha_f \approx -c_f \tan(\alpha_f) = -c_f \frac{v_{A,\eta}}{v_{A,\xi}} \tag{109}$$

$$v_{A,x} = v_x \quad v_{A,y} = v_y + \omega l_f \tag{110}$$

$$\tag{111}$$

However, since we are no interested in the velocities $v_{A,x}$ and $v_{A,y}$, we have to transform them into a coordinate system centered on the direction of the front tire. Therefore we apply a rotation based on the steering angle δ. Based on our assumption the steering angle is small, the equation simplifies yet again:

$$v_{A,\xi} = v_{A,x} \underbrace{\cos(\delta)}_{\approx 1} + v_{A,y} \underbrace{\sin(\delta)}_{\approx \delta} \tag{112}$$

$$v_{A,\eta} = -v_{A,x} \underbrace{\sin(\delta)}_{\approx \delta} + v_{A,y} \underbrace{\cos(\delta)}_{\approx 1} \tag{113}$$

This transformation is important, because it allows us to compute a close form equation of the forced on the front and rear wheels. To get these, we simply plug the equations of the velocities on the wheels in longitudinal and latitudinal direction into the formulas of the forces to get the forces acting on the tires:

$$\begin{aligned} F_r &= -c_r \frac{v_{B,y}}{v_{B,x}} = -c_r \frac{v_y - \omega l_r}{v_x} \\ F_f &= -c_f \frac{v_{A,\eta}}{v_{A,\xi}} = -c_f \frac{-v_x \delta + v_y + \omega l_f}{v_x + (v_y + \omega l_f)\delta} \approx c_f \delta - c_f \frac{v_y + \omega l_f}{v_x} \end{aligned}$$

The last approximation is possible, because v_x is much bigger than $(v_y + \omega l_f)\delta$.

Now that we have equations for the forces on the tires, we can plug them into the formulas of the force via Newton's and the angular momentum:

$$\begin{aligned} m(\dot{v}_y + \omega v_x) &= \underbrace{-c_r \frac{v_y - \omega l_r}{v_x}}_{=F_r} + \underbrace{c_f \delta - c_f \frac{v_y + \omega l_f}{v_x}}_{=F_f} \\ I_z \dot{\omega} &= -l_r \underbrace{\left(-c_r \frac{v_y - \omega l_r}{v_x}\right)}_{F_r} + l_f \underbrace{\left(c_f \delta - c_f \frac{v_y + \omega l_f}{v_x}\right)}_{=F_f} \end{aligned}$$

We can rewrite these two equations into a nicer form called the state space representation, where the state is the velocity in y direction, the heading angle of the vehicle and its angular velocity.

$$\begin{bmatrix} \dot{v}_y \\ \dot{\psi} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} -\frac{c_r + c_f}{m v_x} & 0 & \frac{c_r l_r - c_f l_f}{m v_x} - v_x \\ 0 & 0 & 1 \\ \frac{l_r c_r - l_f c_f}{I_z v_x} & 0 & -\frac{l_f^2 c_f + l_r^2 c_r}{I_z v_x} \end{bmatrix} \underbrace{\begin{bmatrix} v_y \\ \psi \\ \omega \end{bmatrix}}_{\text{State}} + \begin{bmatrix} \frac{c_f}{m} \\ 0 \\ \frac{c_f}{I_z} l_f \end{bmatrix} \underbrace{\delta}_{\text{Input}}$$

This gives us a relationship of the time derivative of the state vector to the state vector itself plus a vector times the steering angle as input. Now we can simulate the vehicle and we know exactly how the velocity in y direction, the heading angle and the angular velocity changes based on the current state and our inputs. This model can also augmented by the global position, but this causes the model to become a nonlinear state space model.

5.5 Summary

A vehicle can be modeled as a rigid body. It is subject to holonomic and non-holonomic constraints. The bicycle model approximates the vehicle using only two wheels. Additionally, the kinematic bicycle model assumes no wheel slip. This approximation only works at low speed. However, modeling tires requires to consider slip. For tires the sliding friction is always smaller than static friction. Our goal is to operate in the static friction area of the force curve. The circle of forces tells us that latitudinal and longitudinal forces are dependent. Finally, we introduced the dynamic bicycle model, which also takes tire forces and wheel slip into account.

6 Vehicle Control

Lecture 6 is about vehicle control, different controller systems and sophisticated controllers. Driver Assistance Systems, that help for example to simplify steering, accelerating or breaking, are already contained in most vehicles today.

6.1 Introduction

In the following we will look at different controllers and control systems. One of the first controller systems is called the governor, a machine that uses centrifugal force to regulate the speed of machines during the industrial revolution to avoid them shaking apart.

https://www.youtube.com/watch?v=B01LgS8S5C8

6.1.1 Open-Loop Control System

The first control system is the Open-Loop control system. There is no feedback of the controlled variable $y(t)$ and therefore impossible to handle unknown situations like sudden obstacles. Open Loop Controllers are difficult to be efficiently used in practice since it requires detailed knowledge about all influencing factors which is not guaranteed in real life.

The goal is that the controlled variable $y(t)$ (eg. distance to the line markings) reaches the referenced variable $r(t)$ (eg. optimal distance to the line markings). $y(t)$ can be adjusted with the correcting variable $u(t)$ (eg. steering angle) where t defines the time. Disturbances that may occur during the process are defined as $z(t)$ Fig. 61. Not knowing how to handle unknown disturbances leads to drift. In real life driving, it is nearly impossible to keep track of all possible disturbances, making Open-Loop Control insufficient for automated driving.

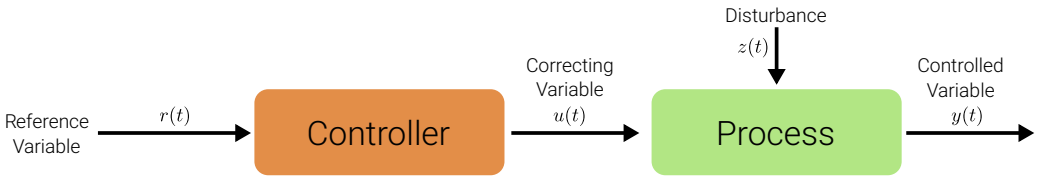


Figure 61: **Open-Loop Control System.** An illustration of the functionality.

6.1.2 Closed-Loop Control System

Closed-Loop Controller have an additional feedback loop and an error $e(t)$ defining the difference of the reference variable $r(t)$ and the controlled variable $y(t)$. The goal is here, to minimize the error between the measurement and the reference. With this inner feedback loop unknown disturbances are now able to be handled by trying to get the error to zero. A small downside is that all measurements have noise which is describe by the measurement noise variable $v(t)$. The noise depends on the quality of the sensor Fig. 62.

In this course, the focus is on Closed-Loop Control, since this is the only sufficient control for self driving. A vehicle needs to be controlled longitudinally (accelerating, braking) AND laterally (steering). There are different Closed-Loop controllers that can be used. In the following lecture there will be an introduction to black box, geometric and optimal controller.

6.2 Black Box Control

Black Box Controller don´t assume knowledge about the process, as the name already suggests. Black Box Controllers only focus on error minimization and can be applied to any task.

6.2.1 Bang-Bang Control

Bang-Bang Controllers are also known as Hysteresis Controllers. Those Controllers abruptly switch between two states. They are often used in household thermostats to regulate water temperature. If the temperature drops below a certain threshold τ_1 the thermostat begins to heat it up u_1 until another temperature threshold is reached τ_2 that causes the thermostat to stop heating u_2. As a result, the water temperature never drops or exceeds certain temperature thresholds. This can be mathematically formulated as:

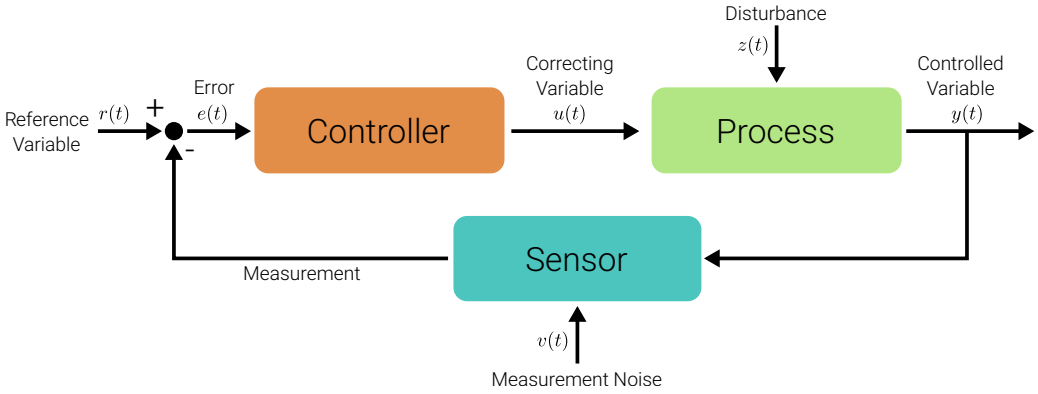


Figure 62: **Closed-Loop Controller.** An illustration of the functionality.

$$u(t) = \begin{Bmatrix} u_1, e(t) \leq \tau_1 \\ u_2, e(t) > \tau_2 \end{Bmatrix}$$

How this would look like in the Closed-Loop System is illustrates in Fig. 63. However, Bang-Bang Controllers are not sufficient for Self Driving since the abruptness of changing actions would lead to a uncomfortable zick-zack driving behaviour.

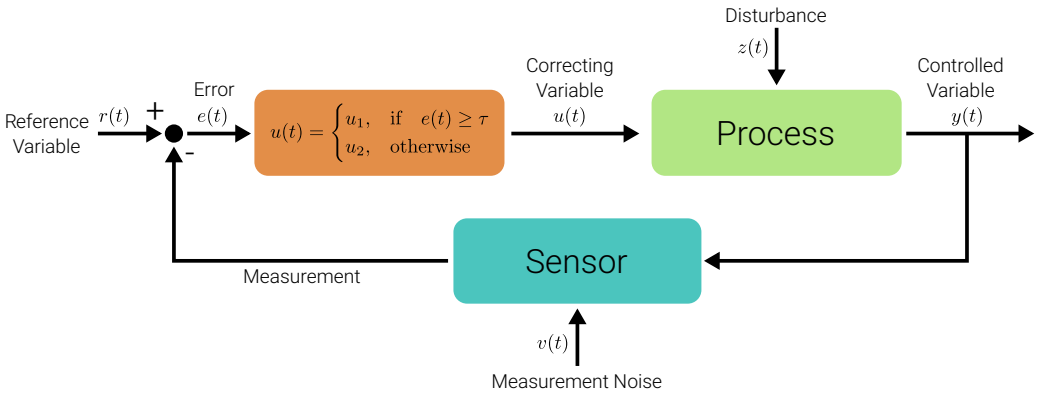


Figure 63: **Bang-Bang Controller.** An illustration of the functionality.

6.2.2 PID Control

A Controller that leads to smoother driving is the PID Controller. PID Controller are the currently most used controller in industrial settings. Similar to the Bang-Bang Controller the PID Controller does not require any knowledge about the plant/process that is controlled. The Controller itself contains of three different components. The **P**roportional Component that controls the correcting variable $u(t)$ proportional to the error multiplied with a gain K_p. The **I**ntegrating component computes the integral of the error over all previous time steps t and multiplies it with a intergral gain K_i. The third component calculates the **D**erivate over time and multiplies is with a differential gain K_d. The outcome of all three components are summed up and defines the correcting variable $u(t)$ Fig. 64.

The mathematical formulation is given as the following:

$$u(t) = K_p e(t) + K_i \int_0^t s(t')dt' + K_d \frac{de(t)}{dt}$$

The most important element is the **P**roportional element, but using it alone causes overshooting and oscillation Fig. 65a. This happens because the proportional control will not immediately stop if it reaches a goal. This delay causes overshooting the goal and having to readjust causes oscillating driving behavior. Adding the **D**erivative component introduces a damping behavior the closer the vehicle gets to the goal Fig. 65b. The **I**ntegral component corrects further residual errors by integrating past error measurements. An example of the effect of the derivative component is illustrated in Fig. 66.

A still open question is how to actually set those parameters. One heuristic approach is the by computer simulation empirically derived called **Ziegler-Nichols**:

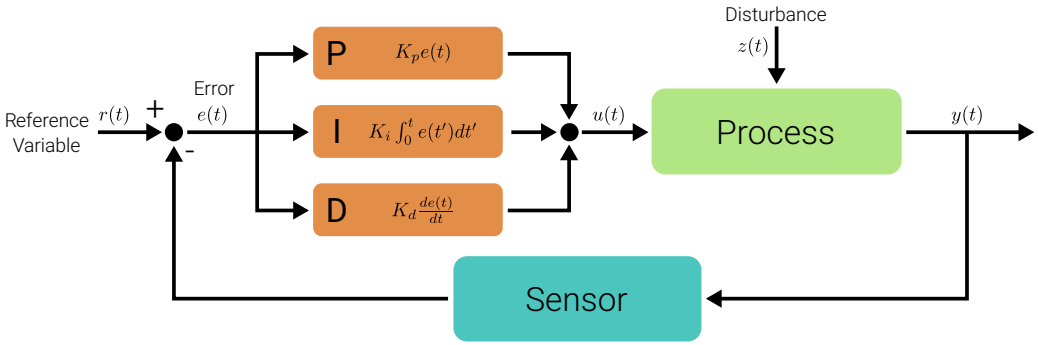
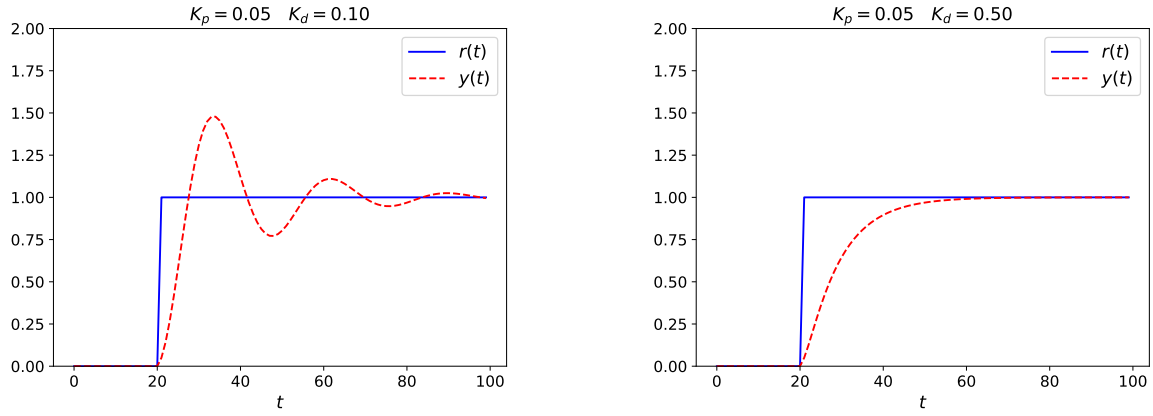


Figure 64: **PID Controller.** An illustration of the functionality.



(a) PID Controll with small effect of the derviative component

(b) PID Controll with larger effect of the derviative component

Figure 65: The Effect of The Derivative Component

- Set $K_i = K_d = 0$
- Increase K_p until the ultimate gain $K_p = K_u$ where the system oscillates
- Measure the oscillation period T_u at K_u
- Set $K_p = 0.6K_u$, $K_i = \frac{1.2K_u}{T_u}$ and $K_d = \frac{3K_uT_u}{40}$

Those heuristics provide a good start, but fine tuning is still necessary to exhibit optimal behavior.

EXAMPLE Longitudinal Vehicle Control

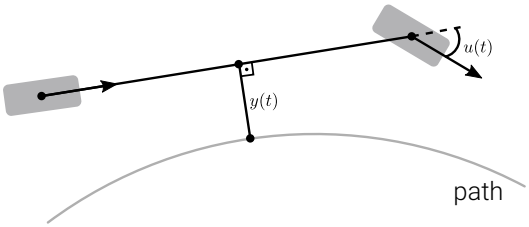
An example of how one can use the PID Control System for longitudinal control is the regulate the vehicle velocity $y(t)$. Let the reference variable $r(t) = v(t)$ be the target velocity and $u(t)$ the gas/brake pedal. The error is calculated by $s(t) = v(t) - y(t)$.

$$v(t) = v_{max}(1 - \exp(-\theta_1 d(t) - \theta_2))$$

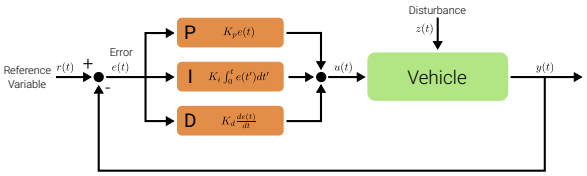
EXAMPLE Lateral Vehicle Control

For Lateral Vehicle Control one can calculate the Cross-Track error. The Cross-Track error describes the distance between the center of gravity of the vehicle and the closest point of the path we want to follow. The goal is that the Cross-Track error becomes zero.

- Reference variable $r(t) = 0$ = no cross track error
- Correcting variable $u(t) = \delta$ = steering angle
- Controlled variable $y(t)$ = cross track error



(a) Cross Track Error



(b) PID Control System

Figure 66: Lateral Control using PID Controller

- Error $e(t) = -y(t) =$ cross track error

For a visual summary of the content so far, check out this video:
https://www.youtube.com/watch?v=4Y7zG48uHRo

EXAMPLE Waypoint-based Vehicle Control

Based on the waypoints we can define two controllers that measure the expected velocity when following the waypoints $w = w_1, ..., w_k$ Fig. 67. The semantic meaning of the waypoints is the position where the vehicle should be at a time certain step in the future. Precisely, if the waypoints are close to the vehicle the velocity should be lower, vice versa the velocity (longitudinal control) should be high if the waypoints are further away:

$$v = \frac{1}{K} \sum_{k=1}^{K} \frac{||w_k - w_{k-1}||_2}{\Delta t}$$

For the lateral control the steering angle δ Fig. 67 can be calculated as the angle between the vehicle heading and the direction towards the look ahead waypoint:

$$\delta = \tan^{-1} \frac{p_y}{p_x}$$

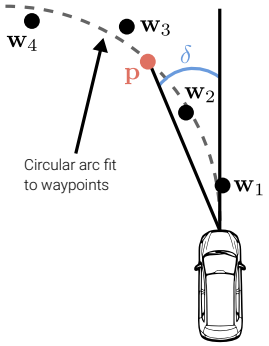


Figure 67: **PID Controller.** An illustration of the functionality.

6.3 Geometric Controller

Getting away from Black Box Controllers and actually using available information about the process. Geometric Controllers make use of that knowledge.

6.3.1 Pure Pursuit Control

The first kind of Geometric Controller we have a closer look at is the Pure Pursuit Controller. Its goal is to track a target point (red point) at look ahead distance d to follow a path. The look ahead distance d is defined as the distance between the target point and the center of the wheels. To get the steering angle δ one can exploit the geometric relationship between the vehicle and the path we want to follow. To achieve this goal the Cross-Track error e gets minimized by following the circular trajectory (light grey circle) around an instantaneous rotation center (black dot). R defines the radius of the trajectory circle. As the vehicle moves, the target point moves

as well. To get the steering angle δ we use the angle α, the angle between the vehicle direction L and the look ahead direction d. Note that α and δ also appear at the top at the rotation center Fig. 68.

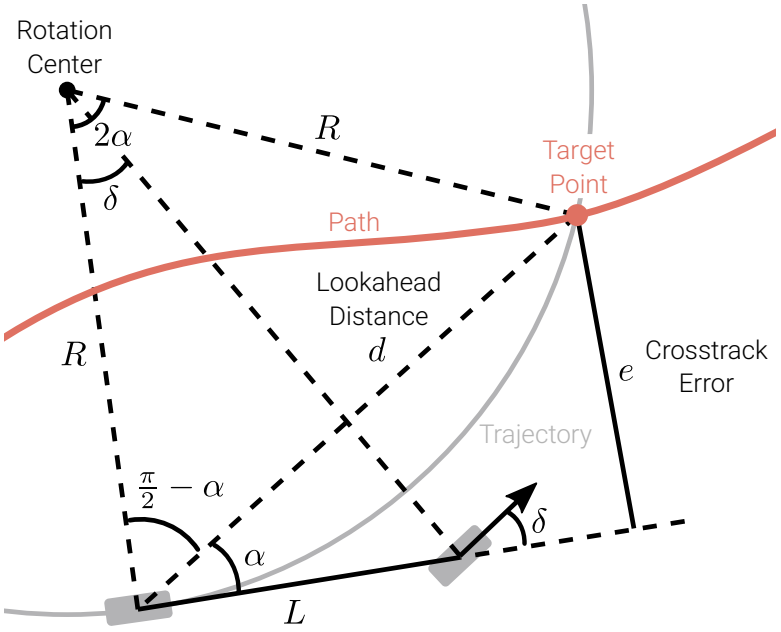
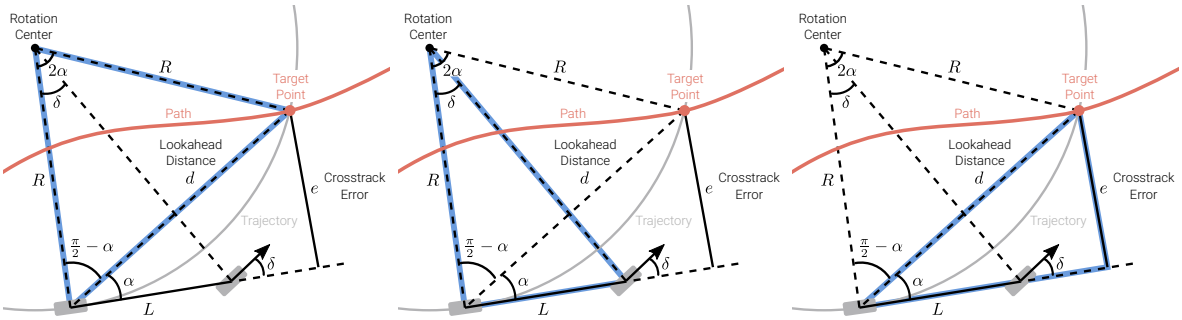


Figure 68: **Pure Pursuit Controller.** Geometrical Features.

To calculate the curvature K of the trajectory we use the law of sines Fig. 69a:

- $\frac{d}{\sin(2\alpha)} = \frac{R}{\sin(\frac{\pi}{2}-\alpha)}$ can be rewritten as:
- $\frac{d}{2\sin\alpha\cos\alpha} = \frac{R}{\cos\alpha}$
- $K = \frac{1}{R} = \frac{2\sin\alpha}{d}$



(a) (b) (c)

Figure 69: **Pure Pursuit Controller.** (a) Calculating the Curvature of the Trajectory using Geometry. (b) Calculating the Steering Angle using Geometry. (c) Calculating the Cross-Track Error using Geometry.

Calculate the steering angle δ using further geometry rules Fig. 69b:

- $\tan(\delta) = \frac{L}{R} = \frac{2L\sin(\alpha)}{d}$ can be rewritten as:
- $\delta = \tan^{-1}(\frac{2L\sin(\alpha)}{d})$
- $\delta \approx \frac{2L\sin(\alpha)}{d}$
- d is often based on vehicle speed v : $d = Kv$ with K being constant

Calculate the Cross-Track Error e using further geometry rules Fig. 69c:

- $\sin\alpha = \frac{e}{d}$

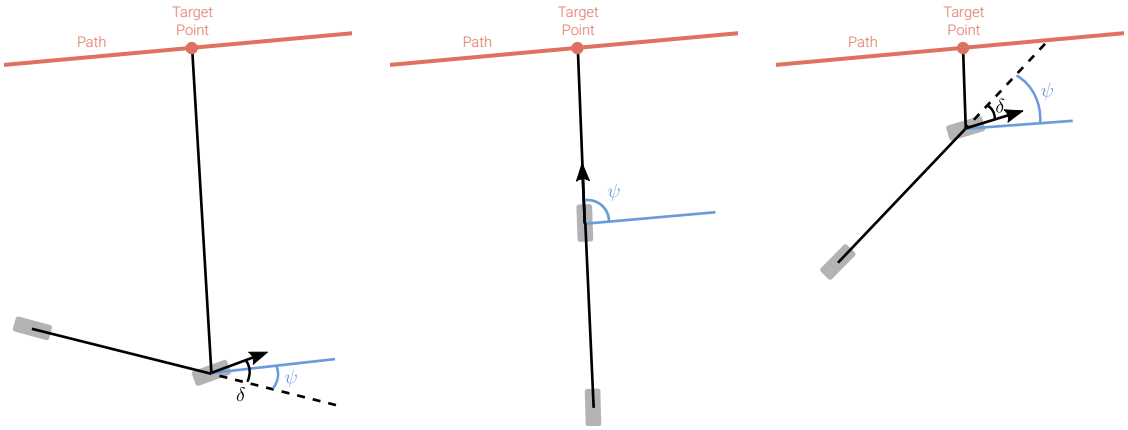


Figure 70: Stanley Control. (a) A case driving parallel to the path. Large cross-track error. (b) A case approaching the path in the perpendicular direction. Strong heading error. (c) A case approaching the path. Small heading and cross-track error.

- $\delta = \tan^{-1}(\frac{2L\sin(\alpha)}{d})$
- $\delta = \tan^{-1}(\frac{2Le}{d^2}) \approx \frac{2L}{d^2}e$
- $d = Kv$ with K being constant

6.3.2 Stanley Control

The Stanley Control was successfully used by the Stanford Racing Team to win the **DARPA Challenge**. The control law used in the challenge is:

$$\delta = \psi - \tan^{-1}(\frac{ke}{v})$$

- v = speed, ψ = heading error, e = Cross-Track Error

ψ directly tries to minimize the heading error. In contrast to the PID Controller the reference point in the vehicle is the front axle and in contrast to the Pure Pursuit Controller it does not have a look ahead direction it just tries to find the closest point on the path to the front wheel. It combines the minimization of the heading and Cross-Track error. It can be shown that the Cross-Track error converges exponentially to 0 independent of v! However, this Control does only work on small velocity without disturbances similar to the others as they do not model tire forces etc.

Fig. 70a shows large Cross-Track and small Heading error, Fig. 70b shows a large Heading error and a large Cross-Track error and Fig. 70c illustrates rather low heading and Cross-Track errors. The Stanley Controller can correct both large heading and Cross-Track errors. Furthermore, Stanley Control is **globally stable**, it is proven that it guides the vehicle back on track independent of initial conditions. Additionally damping terms or curvature information can be included for even better results. However, it does not consider noisy observation, actuator dynamics or tire force effects.

A real life application of Stanley Control in automated driving can be seen here:
https://www.youtube.com/watch?v=M2AcMnfzpNg

6.4 Optimal Control

All Controllers we have seen so far have a common problem: The are not able to look into the future. However, it is often necessary to look into future events in situations like curves, slippery roads or crossings.

6.4.1 Linear Quadratic Regulator (LQR)

Consider a time-continuous linear system

$$\dot{x} = AX + b\delta, x(0) = x_{init}$$

and a quadratic cost function with orthogonal weight matrix Q

$$J = \frac{1}{2}\int_0^\infty x^T(t)Qx(t) + q\delta(t)^2 dt$$

the feedback control $\delta(t)$ that minimizes J is given by:

$$\delta(t) = -k^T(t)x(t)$$

with $k(t) = \frac{1}{q}b^T P(t)$ and $P(t)$ being the solution of the *Ricatti equation*. There were no further information about this equation, since it exceeds the course material. One important note is that there is a closed form solution due to the specific formulation that was done above.

6.4.2 Model Predictive Control

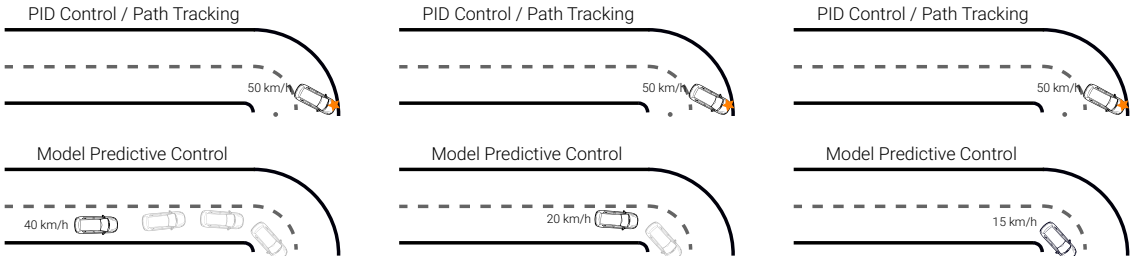
Model Predictive Control is the most expressive controller and generalizes the idea of Linear Quadratic Regulators resulting in a more flexible approach. However, Model Predictive Control gives up on the closed form solution introduced by the Linear Quadratic Regulator which requires test time optimization. Therefore the approach is computationally more expensive.

More precisely, Model Predictive Control generalizes LQR to **non-linear** cost functions and dynamics, which enables to handle straight roads that lead into a turn. It allows for incorporating constraints and receding windows. However, it is more expensive since we need a non-linear optimizer. The optimization problem of Model Predictive Control can be stated formally by:

- argmin $_{\delta_1,\dots,\delta_T} \sum_{t=1}^{T} C_t(\mathbf{x}_t, \delta_t)$ Sum of Costs
- s.t. $x_1 = x_{init}$ Initialization
- $x_{t+1} = f(x_t, \delta_t)$ Dynamics Model
- $\underline{\delta} \leq \delta_t \leq \overline{\delta}$ Constraints

So we unroll the dynamic model T times (like Recurrent Networks in Deep Learning) and apply non-linear optimization to find $\delta_1, ..., \delta_T$. In the following a example is provided why Model Predictive Control is useful at fast velocities. Consider the following scenario: A vehicle drives at 50 kilometers per hour on a straight road that leads to a curve. A PID Controller that follows a waypoint is unable to break and steer fast enough to correctly take the turn. It overshoots and crashes into the other side of the road which is a worst case scenario in automated driving Fig. 71a.

The Model Predictive Control however, looks into the future and can adapt velocity and steering right in time Fig. 71b. It can be observed that the vehicle with Model Predictive Control decelerates and slightly steers to the left so the right turn can be done without crossing the line markings in the middle of the road Fig. 71c. This example illustrates the importance of incorporating the prediction of future time steps into the model, especially for high velocity.



(a) (b) (c)

Figure 71: **Model Predictive Control.** Comparison to PID Control handling a Curve at 50 km/h.

7 Odometry, SLAM, Localisation

In this lecture, we will focus on the individual modules of the modular pipeline regarding the ego-motion of self-driving cars. As self-driving cars operate in a dynamic environments and hence must be aware of their ego-motion and the motion of other traffic participants. Ego-motion, which describes the own motion of the vehicle, includes three different types of motion:

- Visual odometry which refers to the estimation of relative ego-motion from images
- SLAM algorithms which build a map and simultaneously localise in that map (note that both of these tasks need to be tackled jointly)
- Localisation methods which find the global pose, how the vehicle is located with respect to the map, of a vehicle in a given map

All of these three methods are useful things to do for example estimating the ego motion relative to the previous frame is important to integrate trajectory over a longer time horizon to measure how well e.g. our dynamics model predicts and to be able to control the vehicle we need to know how we are doing. Furthermore, the localisation via a map is also important as we might add useful information to the map such as the location of traffic lights or lane markings which could facilitate the detection of these. All three of these methods will be explained in detail in the following sections.

7.1 Visual Odometry

Odometry is the use of sensors to estimate the change in ego-position over time and yields relative motion estimates not global position estimates with respect to a map. It is hence sensitive to error accumulation over time and only precise locally. Odometry information can be obtained using a variety of sensors such as:

- Wheel odometry systems which use wheel encoders to measure wheel rotation
- Inertial measurement units that measure a body's forces
- Visual odometry algorithms that use camera images

The inertial measurement units measure force through accelerometers which typically estimate acceleration linearly but also angularly. To obtain the pose we need to integrate twice which also means that the noise is integrated twice. That results in IMU data not being usable on its own to arrive at precise pose estimates. Therefore, it is often used in combinations with visual or GPS sensors. Furthermore, multiple systems are often combined to complement each other, e.g. VIO. This leads to more robust results for real-world scenarios where one type of sensor would not be sufficient enough, e.g. in a tunnel, there is not enough light for the visual odometry algorithms to work. However, today's focus will be laid solely on visual odometry algorithms.

Visual odometry is concerned with tracking the pose, i.e., position and orientation, of the camera with respect to the environment from its images. As if we are able to track the camera pose and the camera is calibrated we know the relative pose of the camera with respect to the vehicle coordinate system. Therefore, we know the vehicle motion through knowing the camera motion. Visual odometry considers a limited set of recent images for real-time which can lead to error accumulation. However, most times visual odometry only considers the current frame with respect to the previous frame from estimating the relative motion between two images. Then, a sparse local map is often built as a by-product. However, mapping is not the focus of visual odometry. Here, global reconstruction is used from knowing the pose of each individual camera at each frame and combining all local reconstructions from the stereo. This local map of the environment constructed from odometric information is used to estimate the motion. While not being very dense or accurate compared to e.g. SLAM algorithms, it is still required to estimate the pose.

In the following, we will introduce two different kinds of visual odometry, see figure 72, indirect and direct methods. Due to both their advantages and disadvantages, these are often combined in practice.

7.1.1 Indirect Methods

Indirect methods include a feature matching step which is performed by a feature-based visual odometry method. These indirect methods parse the input images through a feature detector to then extract local features and match key points, this is also called an indirection step and can be performed by e.g. SIFT, SURF, BRISK or ORB (some are faster than others). Features should be invariant to perspective and illumination changes. This step detects salient points such as blobs or corners in the images and tries to match these features/points between respective frames, which are typically adjacent, by their similarity to get image correspondences. Typically, for

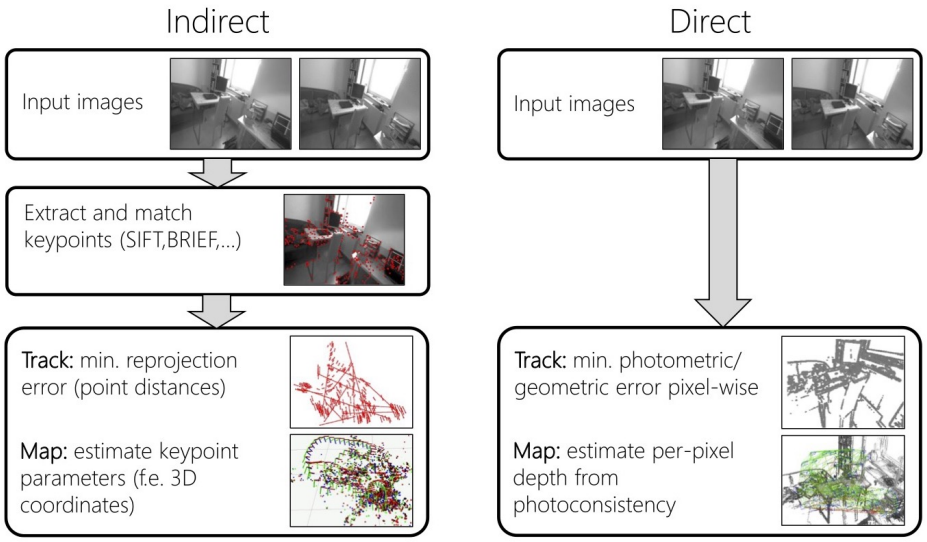


Figure 72: **Indirect vs. Direct Visual Odometry**

self-driving applications we do not need to be as invariant as in general computer vision as the two frames we are comparing are taken very close in time and therefore nothing much has changed. Afterwards, an algorithm is applied that tracks the minimal reprojection error (point distances) and returns a map of estimated keypoint parameters e.g. 3D coordinates. We then get the relative motion estimate. These feature-based methods are faster but in general less accurate because often only a few reliable feature correspondences can be found. However, they are more robust to initialisation and less prone to local minima.

7.2 Image Formation

Generally, 2D points can be written in inhomogeneous coordinates as seen in equation 114 or in homogeneous coordinates as seen in equation 115. In these homogeneous coordinates, a 3D vector is used for a 2D point and allows to define the image projection process that is inherently non-linear as a linear function/mapping. Inhomogeneous vectors can be converted to homogeneous vectors and vice versa using the augmented vector $\bar{x}$ where the last element of a homogeneous vector is equal to 1, see figure 73.

$$x = \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2 \tag{114}$$

$$\tilde{x} = \begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{w} \end{pmatrix} \in \mathbb{P}^2 \tag{115}$$

The goal of perspective projection is to estimate the location of $\mathbf{x}_s$, which describes the point where the image plane is intersected, from the 3D point $\mathbf{x}_c$ relative to the camera coordinate system. Hence the perspective projection of a 3D point $\mathbf{x}_c \in \mathbb{R}^{\not\models}$ to pixel coordinates $\mathbf{x}_s \in \mathbb{R}^{\not\models}$, the screen coordinates in the 2D image plane. The light rays, which are always linear, pass through the camera centre, the pixel $\mathbf{x}_s$ and the point $\mathbf{x}_c$. We can assume that the principal axis, which is orthogonal to the image plane, aligns with the z-axis. The perspective projection can be seen in figure 74. However, we need to note that in this figure the y coordinate is not shown for clarity but behaves similarly.

In perspective projection, 3D points in camera coordinates x_c/y_c are mapped to the image plane by dividing them by their z component z_c and multiplying with the focal length $f_{x/y}$ and can be described by the equation 116.

$$\begin{pmatrix} x_s \\ y_s \end{pmatrix} = \begin{pmatrix} f_x x_c/z_c \\ f_y y_c/z_c \end{pmatrix} \Leftrightarrow \tilde{\mathbf{x}}_s = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{116}$$

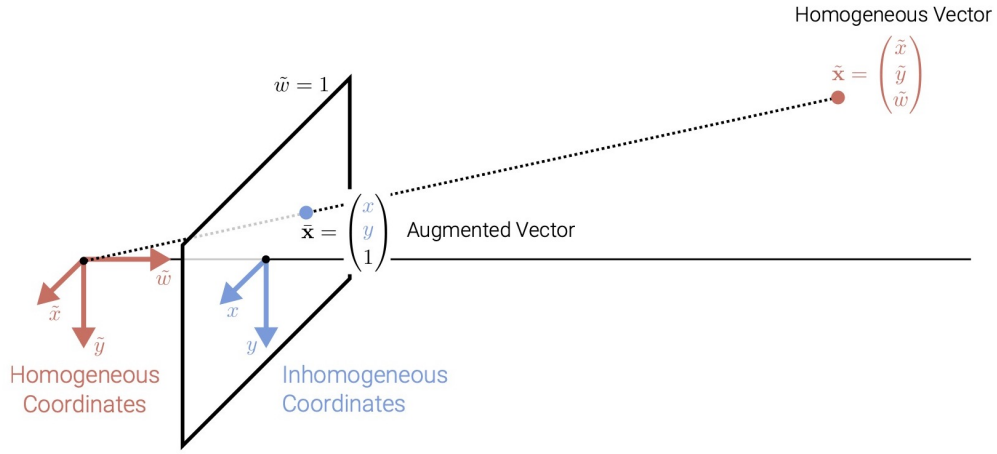


Figure 73: **2D Points** - homogeneous and inhomogeneous vectors

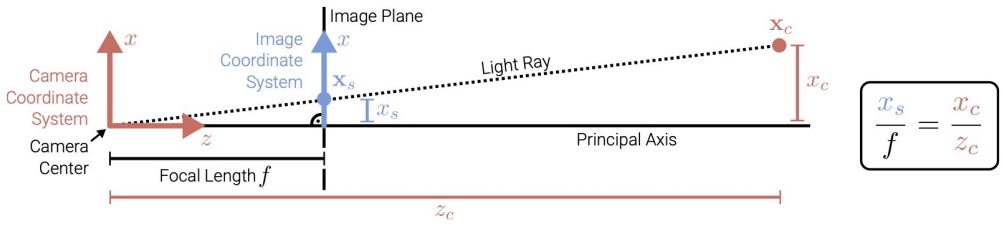


Figure 74: **Perspective Projection**

The part on the right is nonlinear because we divide by z, but becomes linear when converting to the 3D homogeneous vector $\tilde{x}$ consisting of homogeneous coordinates, which implicitly divides by z for us. So far, we have assumed that the centre of the image coordinates system is at the point where the principal axis intersects the image plane but normally we work with positive pixel coordinates where the image coordinate system is set at the corner of the image. Therefore, to ensure positive pixel coordinates, a principally point offset c is usually added which moves the image coordinate system to the corner of the image plane. The complete perspective projection model can then be given by:

$$\begin{pmatrix} x_s \\ y_s \end{pmatrix} = \begin{pmatrix} f_x x_c/z_c + s y_c/z_c + c_x \\ f_y y_c/z_c + c_y \end{pmatrix} \Leftrightarrow \tilde{\mathbf{x}}_s = \begin{bmatrix} f_x & s & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{117}$$

Here, the left 3x3 submatrix of the projection matrix is called the calibration matrix K which parameters are called camera intrinsics. Note that f_x and f_y are independent, allowing for different pixel aspect ratios, and the skew s arises due to the sensor not being mounted perpendicular to the optical axis. Furthermore, whenever a 3D point i not represented in the camera coordinates but in world coordinates, we can chain the transformations if we know the camera pose concerning the world coordinate system which allows us to still project the point in world coordinates onto the image plane. Let K be the calibration matrix (intrinsics) and $[\mathbf{R} \mid \mathbf{t}]$ the camera pose (extrinsic). We then chain both transformations to project a point in world coordinates to the image which can be seen in figure 75 below.

Epipolar geometry forms the basis for most simple visual odometry algorithms and tries to estimate, given two images, the relative motion the camera has undergone between these two images. This assumes that the camera has been calibrated beforehand, which is true for most visual odometry systems, therefore the intrinsic camera parameters are known. Epipolar geometry aims to recover the camera pose and 3D structure from image correspondences. The required relationships are described by the two-view epipolar geometry, see figure 76. We assume the following:

- Let R and t denote the relative pose between two perspective cameras
- A 3D point x is projected to pixel x_1 in image 1 and pixel x_2 in image 2
- The 3D point x and the two camera centres span the epipolar plane

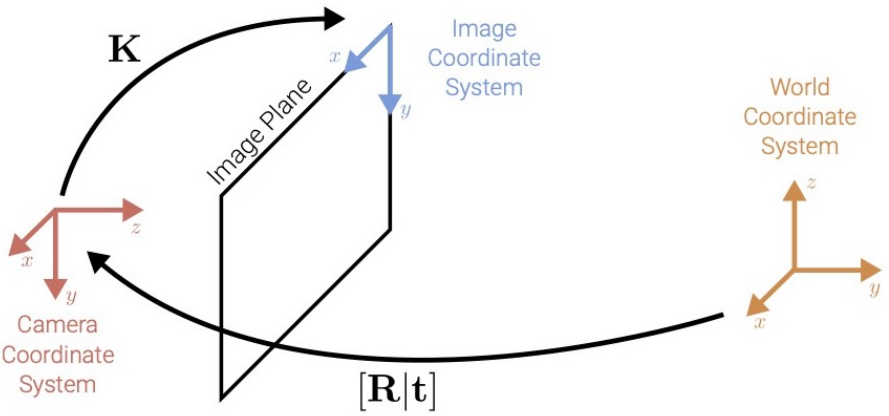


Figure 75: **Chaining transformations**

- The correspondence of pixel x_2 in image 1 must lie on the epipolar line l_1 in image 1
- The correspondence of pixel x_1 in image 2 must lie on the epipolar line l_2 in image 2
- All epipolar lines pass through the epipole

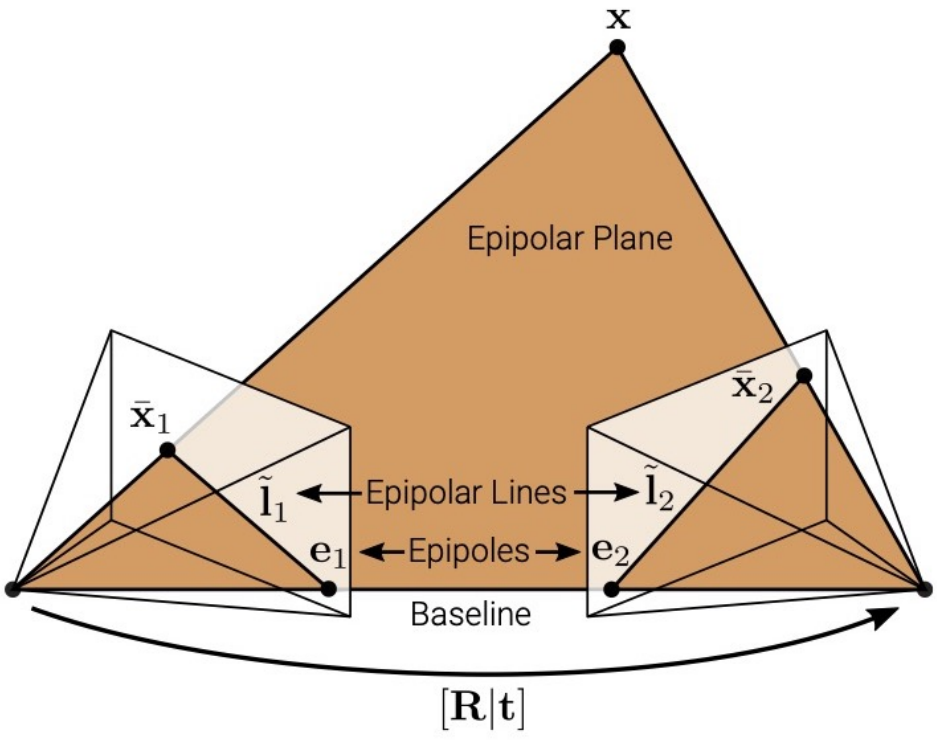


Figure 76: **Epipolar Geometry**

The epipolar constraint, the fundamental property, is given by $\tilde{\mathbf{x}}_2^T \tilde{\mathbf{E}} \tilde{\mathbf{x}}_1 = 0$ with the essential matrix $\tilde{\mathbf{E}} = [\mathbf{t}] \times \mathbf{t}$. The epipolar constraint holds for all correspondences of points and multiplies the homogeneous point on the image plane of camera two with the essential matrix (3 x 3) with the point/pixel in the other image. For the essential matrix, we multiply the translational part $\mathbf{t}$ with the 3 x 3 rotation matrix. This converts the translational vector into a skew-symmetric 3 x 3 matrix that represents the translational vector. Furthermore, we can recover the rotation and translation parts by determining the essential matrix from N image correspondences forming N homogeneous equations in the nine elements of the essential matrix. As the

essential matrix is homogeneous we use singular value decomposition to constrain the scale and decompose the essential matrix into its rotational and translational part which allows us to receive the odometric information we seek.

Additionally, the rigid body motion **(R, t)** can be further refined via non-linear optimisation. Let $\pi(\mathbf{x}_i, \mathbf{R}, \mathbf{t})$ denote the projection of the i'th 3D point onto the 2D image plane and let $(\mathbf{x}_{1,i}, \mathbf{x}_{2,i})$ denote the 2D feature correspondences in the first and second frame. We then want to minimise the reprojection error of all correspondences concerning the motion and the 3D location:

$$\mathbf{R}^*, \mathbf{t}^*(, \mathbf{X}^*) = \underset{\mathbf{R},\mathbf{t}(,\mathbf{X})}{\arg\min} \sum_{i=1}^{N} \|\mathbf{x}_{1,i} - \pi(\mathbf{x}_i; \mathbf{I},\ \mathbf{0})\|_2^2 + \|\mathbf{x}_{2,i} - \pi(\mathbf{x}_i; \mathbf{I},\ \mathbf{0})\|_2^2 \tag{118}$$

Optimising reprojection errors as opposed to algebraic models leads to better results / a better noise model. The equation stated above minimises/optimises for R and t but also allows for X, however, this takes more time but improves the results. We then take the sum of all correspondences of the measured corresponding image 1 subtracted by the predicted correspondence, where the point is assumed to be in the coordinate system of camera 1 so there is no need for extrinsic transformation, adding the same for the second image and camera. Note that, outliers must be considered and removed to not affect the estimate e.g. using RANSAC and triangulation can be used to initialise X and speed up by not optimising X. Also, for monocular visual odometry, the length of the translation vector cannot be determined. But, this ambiguity can be resolved by using stereo images or wheel odometry. These stereo measurements allow for instantaneous triangulation as they know the depth of the point through correspondences between the left and the right camera.

7.2.1 Direct Methods

Direct methods to not have a feature matching step and hence directly align the images concerning each other based on all image pixels, which provides more potential information, by tracking the minimal photometric /geometric error pixel-wise to then map the estimate per-pixel depth from photo consistency between the warped images trying to align two images through consecutive images of such a motion of such an image sequence to find the optimal motion. These methods work using the peer-pixel depth which is measured through sensors (RGB-D, Lidar) or multi-view stereo. As, if we know the per-pixel depth we can simulate an image from a different viewpoint. Therefore, we can optimise the viewpoint such that the simulated image pixels agree with the pixels from the observed image.

$$\mathbf{R}^*, \mathbf{t}^*, \mathbf{D}_1^* = \underset{\mathbf{R},\mathbf{t},\mathbf{D}_1}{\arg\min} \sum_{x \in \Omega} \|\mathbf{I}_1(\mathbf{x}) - \mathbf{I}_2(\pi(\mathbf{x}; \mathbf{R}, \mathbf{t}, \mathbf{D}_1))\|_2^2 \tag{119}$$

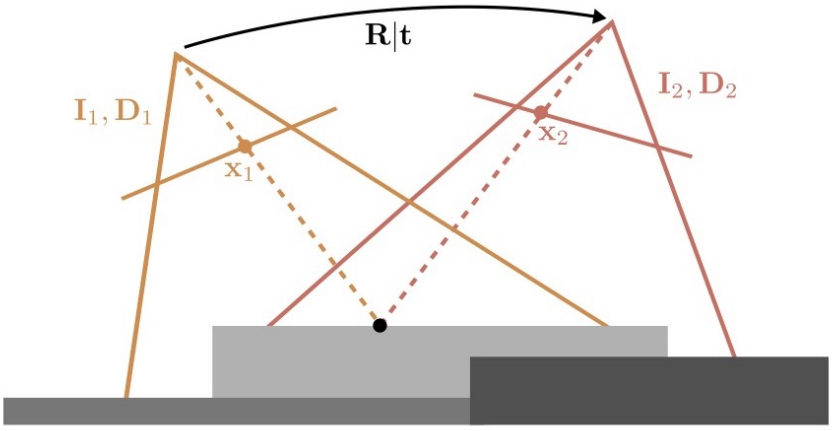


Figure 77: **Direct Visual Odometry** Simulating an image from a different view point [16]

Ideally, image 2, which is warped into the first cameras image plane, is equal to image 1. However, in real-world settings, we try to minimise the photometric error in the image concerning the extrinsic parameters. The function stated above minimises the difference between the two images concerning $\mathbf{R}$, $\mathbf{t}$, and $\mathbf{D}_1$. It takes the sum over all pixels in the image domain of the intensity of image 1 at pixel x subtracted by the intensity of the projection f a 3D point onto the second image plane. Here, the projection function knows x, the rigid

transformation, the extrinsic of the camera and the depth of the point to transform happen. Once we have done this transformation we can query the intensity of that pixel in the second image and warp it into the first image. However, direct visual odometry methods often consider only parts of the image e.g. non-saturated which is sufficient to get a robust estimate of the odometry. Direct methods often lead to more accurate results as they exploit the full image. However, satisfying real-time requirements requires efficient implementations. Furthermore, they suffer from local minima and require accurate initialisation.

7.3 Simultaneous Localisation and Mapping (SLAM)

So far, we have optimised two adjacent frames but did not focus on a globally consistent and accelerated map, which was only produced as a by-product to help estimate the relative motion. Now, we want to be able to optimise over larger windows, ideally the entire history meaning all existing frames. To do so, we need to optimise both poses and the map which can be done by SLAM algorithms. SLAM is known as a chicken-egg problem as localisation requires mapping and vice-versa. Therefore, joint optimisation of poses and maps is necessary. SLAM's key feature is the correct accumulation errors via loop-closure detection which results in a more globally consistent map that can be used for localisation. There exist indirect (feature-based) and direct SLAM methods in any different implementations e.g. EFK SLAM or Bundle adjustment that differ in e.g. the number of poses they update. However, in this lecture, we will solely focus on feature-based SLAM via bundle adjustment.

The goal of feature-based SLAM is to optimize reprojection errors, the distance between observed feature and projected 3D point in an image plane, concerning the camera parameters and the 3D point cloud. Bundle adjustment minimises the reprojection error of all observations as described in equation 120. We assume the following:

- Let $\prod = \{\pi_i\}$ denote the N cameras including their intrinsic and extrinsic parameters.
- Let $\chi_w = \{\mathbf{x}_p^w\}$ with $\{\mathbf{x}_p^w\} \in \mathbb{R}^3$ denote the set of P 3D points in world coordinates.
- Let $\chi_s = \{\mathbf{x}_{ip}^s\}$ with $\{\mathbf{x}_{ip}^s\} \in \mathbb{R}^2$ denote the image (screen) observations in all i cameras.

$$\overset{*}{\prod}, \chi_w^* = \underset{\prod, \chi_w}{\arg\min} \sum_{i=1}^{N} \sum_{p=1}^{P} w_{ip} \left\| \mathbf{x}_{ip}^s - \pi_i(\mathbf{x}_p^w) \right\|_2^2 \tag{120}$$

Here, w_{ip} indicates if point p is observed in image i and $\pi_i(\mathbf{x}_p^w)$ is the 3D-to-2D projection of 3D world point $\mathbf{x}_p^w$ on to the 2D image plane of the i'th camera.

$$\pi_i(\mathbf{x}_p^w) = \begin{pmatrix} \tilde{x}_p^s & \tilde{w}_p^s \\ \tilde{y}_p^s & \tilde{w}_p^s \end{pmatrix} \text{ with } \tilde{\mathbf{x}}_p^s = \mathbf{K}_\text{i}(\mathbf{R}_\text{i}\mathbf{x}_p^w + \mathbf{t}_\text{i}) \tag{121}$$

Note, that the newly introduced world coordinate system could possibly coincidence with one of the camera systems but does not necessarily have to. In figure 78, $\mathbf{K}_\text{i}$ and $[\mathbf{R}_\text{i}|\mathbf{t}_\text{i}]$ are the intrinsic and extrinsic parameters of π_i, respectively. During bundle adjustment, we optimize $\{(\mathbf{K}_\text{i}, \mathbf{R}_\text{i}, \mathbf{t}_\text{i})\}$ and $\{\mathbf{x}_p^w\}$ jointly.

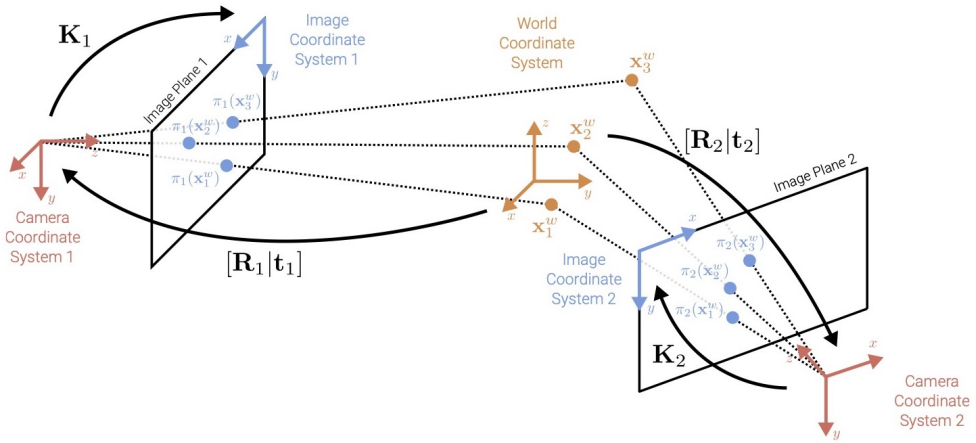


Figure 78: **Bundle Adjustment** an overview of the different coordinate systems

The Hessian is typically sparse as landmarks/poses do not interact with each other which leads to tremendous computational benefits when using sparse optimisers allowing us to exploit sparsity. Furthermore, we can use loop closure detection, shown in figure 79, to correct the drift, the accumulation of the error over time. While revisiting the same place again and again while traversing the environment we can try to match features concerning previous frames and frames very far into the past. So, loop closure detection finds correspondences between the current frame and all previous frames and uses these correspondences as additional constraints during optimization to then reduce the error accumulation.

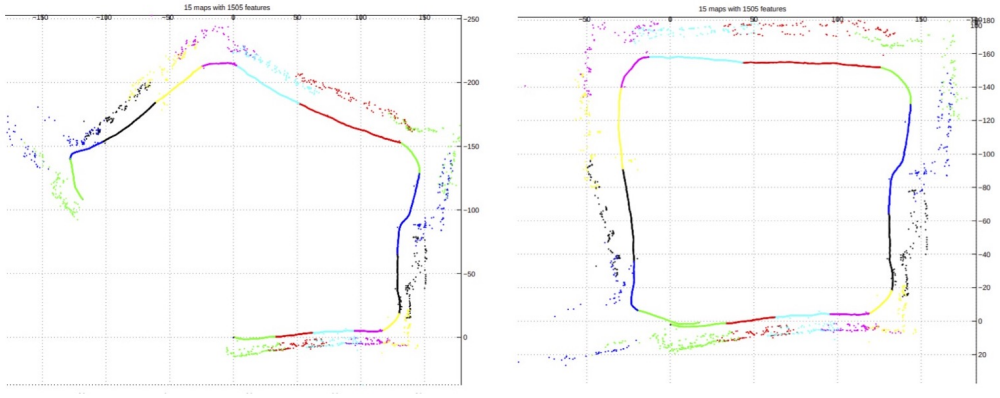


Figure 79: **Loop Closure Detection** left: before optimisation // right: after optimisation [10]

7.4 Localisation

Control concerning a path requires knowledge about the position of the vehicle. Sometimes local knowledge is sufficient e.g. the lateral position on a highway, but often the global location is required e.g. path planning, determining relevant elements that have been marked in a map such as street signs or lane markings. Therefore, several localisation approaches have been introduced which we will look at in further detail in the following.

7.4.1 Satellite Localisation

In the past numerous satellite system have been introduced from a variety of countries that all use infrastructure and can all be used by modern GPS receivers. Generally, the more satellites available the more accurate the poses and the more available the pose estimates. Currently, there are about four to five satellites per orbital plane. However, the satellite visibility depends on the local neighbourhood as the satellites are orbiting the earth and are sometimes occluded by the earth itself. Therefore, it is important to see as many satellites as possible to localise as accurately as possible. Trilateration is defined as the principle of localisation through satellites and aims at determining the location by measuring distances to satellites at known locations using the geometry of circles and spheres. The distance to each satellite is measured as the time delay between sent signal/code and received signal/code and satellites are usually equipped with an onboard atomic clock. Therefore we can assume that:

- Given one satellite, we know which sphere we are on
- Given two satellites, we know which circle we are on
- And given three satellites, we know the exact location

Often, in practice, a fourth satellite is used for time synchronization and differential GPS is used to improve measurements using ground stations at fixed locations. The variance in the estimate is described by the `Geometric Dilution of Precision (GDOP)`, which depends heavily on the constellation of satellites concerning each other as the precision with which your location is estimated depends on this. However, there are a few problems that occur when relying on satellite localisation, such as their availability, as satellites are often not visible in e.g. tunnels or narrow city streets and are dependent on the national organisations in charge and their interests. Also, their accuracy is not precise enough to predict the full pose but only is accurate enough to predict the location not the rotation unless we fuse the satellite information with e.g. information from an IMU. Furthermore, the satellite frequency is not frequent enough to control vehicles as the measurements we are receiving are too sparse. Additionally, the atmospheric variations can potentially lead to errors and multipath effects can lead to wrong signals due to reflections from e.g. facades.

7.4.2 Visual Localisation

Visual records a map/database of known locations with associated features. This map could be a collection of 3D landmarks with feature vectors that describe the appearance of the landmarks. These features can be extracted from images or laser scans as part of the map building. Generally, the mapping should be conducted under similar conditions as during localization to minimize the domain shift concerning the sensor setup and environmental conditions. Visual localisation either localizes only at the image level (topometric localisation) or refines using triangulation or geometric pose estimation and at localization time, tries to retrieve features extracted from the current input image/scan in the map/database. However, the map can be very large, so we need to efficiently retrieve the correct information within a large database in real-time which can be quite challenging. Furthermore, appearance changes can be quite problematic when using image sensors.

`Topometric localisation` is an image-level localisation technique that only localises concerning a frame ID. It does not retrieve the 6D pose, but just a frame index, resulting in no exact 3D pose. Topometric localisation records a sequence with cameras, Lidar and GPS as ground truth and aims to localise a new image (different day/season) concerning the recorded sequence. Topometric localisation is a combination of metric and topological localization. The metric part optimises the pose concerning the feature correspondences and the topological part estimates the observer location qualitatively from a finite set of locations. The map is created using a directed graph with nodes as frames which are created based on the distance threshold. Note that, if the vehicle stands still there are no new frames created as those frames would not provide any new information and are therefore unnecessary.

$$Predict : P(x_t \mid z_{1:t\text{-}1}) = \sum_{x_{t\text{-}1}} P(X_t \mid x_{t\text{-}1}) P(X_{t\text{-}1} \mid x_{1:t\text{-}1}) \tag{122}$$

$$Update : P(x_t \mid z_{1:t}) \propto P(z_t \mid x_t) P(x_t \mid z_{1:t\text{-}1}) \tag{123}$$

Then, we apply Bayesian localisation described in equation 122 and 123 [3], a discrete Bayesian filter with standard filter equations. We want to model the distribution over the location over the graph nodes where each node is a frame in the map. This Bayesian localisation, strictly speaking, does not handle localisation but only tracking. Also, it needs to know the starting position at the start point to track through the route.

`Learning-based localisation` uses neural networks to try to directly regress to the pose and that is exactly what has been done in PoseNet. PoseNet uses a single RGB image as input and outputs a 6 degree of freedom camera pose with a 3D camera position and a 3D camera rotation. Furthermore, it makes use of a 23 layer deep convolutional network based on GoogLeNet and is said to be more robust than feature-based methods but also less accurate than feature-based methods.

`Feature-based localisation` is similar to SLAM methods and requires geometric verification, feature extraction on the query image as well as the generation of a feature-based map.

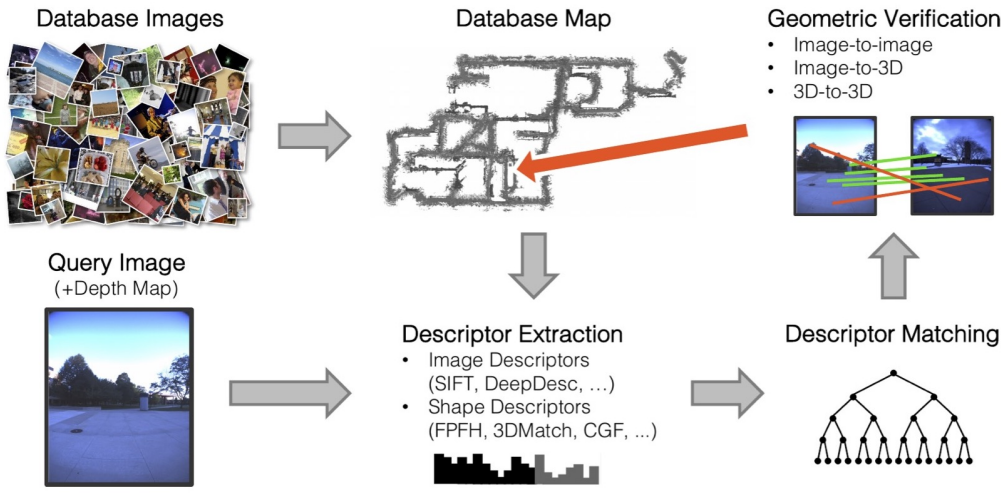


Figure 80: **Feature-based Localisation** an Overview

An overview of the feature-based localisation can be seen in figure 80. During the mapping process, feature-based localisation performs a sparse 3D reconstruction based on sparse feature correspondences (SLAM) and

associates each 3D point with a local image descriptor. Here, the semantic viewpoint invariant feature representations can facilitate matching. During the descriptor matching, feature-based localisation searches for the nearest neighbours of features from the query image in the descriptor space and approximates the search due to the curse of dimensionality. Popular approaches would be e.g. k-d trees or inverted index. During the pose estimation, given n 2D-to-3D correspondences feature-based localisation aims to compute pose R, t such that $x_i = K[R|t]X_i$, the projection error is minimised. Here, K, R, t denote the calibration matrix, rotation matrix, and translation vector. For the pose estimation, very efficient methods exist if the camera calibration is known. During the geometric verification, which tries to figure out which correspondences are correct and which are wrong, algorithms such as RANdom SAmple Consensus (RANSAC) are used to solve the problem of wrong feature matches corrupting pose estimation results by solving minimal problems many times and returning the best solution. RANSAC faces the problem of having more than 80% outliers and solves this quite fast. It takes / samples three observations, which is the minimal amount, randomly from features found in the image and fits the model. It then takes the solution where the majority of the other images align and in the end rejects all outliers which did not project precisely on the 2D observations.

7.4.3 Map-based Localisation

Map-based localisation uses a road map and relative motion estimates from visual odometry to localise in that road map as well. It is not a tracking but a full localisation method that updates the location probability distribution on a map using visual odometry measurements.

7.5 Summary

In conclusion, visual odometry estimates relative ego-motion from images. SLAM algorithms build a map and simultaneously localize in that map. Furthermore, they use loop closure detection to close loops. In general, localization methods find the global pose in a given map. Indirect and direct visual odometry / SLAM methods exist. The indirect methods are usually faster and converge better, but are less accurate. On the other hand, direct methods are slower but lead to more accurate results. Both, direct and indirect methods can be advantageously combined. Also, except for offline mapping, real-time computation is generally required.

8 Road and Lane Detection

In the previous lecture we have seen how we can build maps and also utilize maps through localization in order to, for example, plan the path of a vehicle. However, a lot of information can be sensed locally, using the sensors of the vehicle. This lectures covers that topic, also, with the goal of path planning.

8.1 Introduction

Road and Lane Detection algorithms are available today in a lot of modern vehicles. **State-of-the-Art Commercial Systems** can include some of the following:

- **Lane departure warning** - driver is warned when leaving the current lane,
- **Lane keeping support** - driver is assisted with minimal steering interventions in order to stay in the lane, but it is easy to override this command,
- **Lane keeping/lane departure protection** - car keeps the current lane autonomously[3].

Road and Lane Detection - the goal is to navigate without a detailed global map by sensing "drivable areas" in the vicinity of the car. For this task, there are multiple cues available such as geometric (e.g., some areas are elevated when compared to another), semantic (e.g., the road is gray and has a certain texture), and man-made cues (e.g., lane markings - made to support the driver).
Input for this problem is: image/Lidar/radar. Example input can be seen in Figure 114a.
There is multiple possible **output representations**.

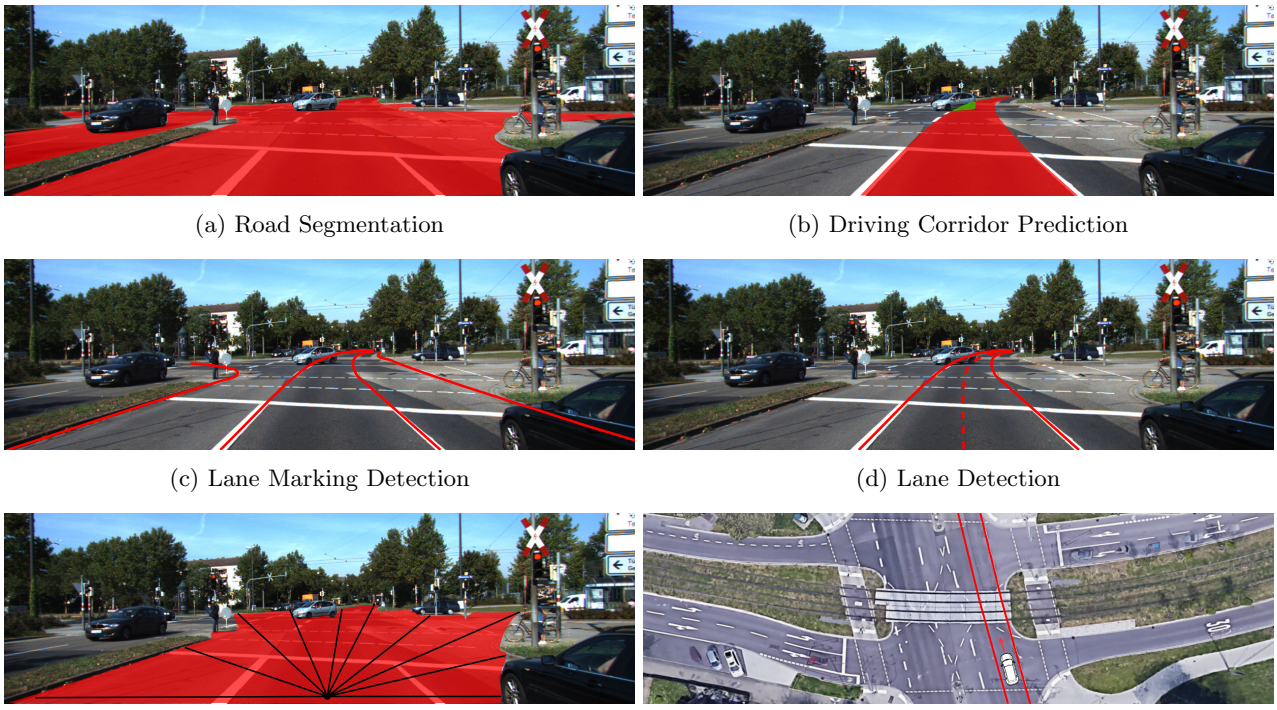
Figure 81: **Example input** - Road and Lane Detection problem.

8.1.1 Representations

- **Road Segmentation** - This is a purely semantic cue ("where is a vehicle allowed to drive?"), with the goal to classify every pixel as either road or non-road pixel. Road segmentation does not consider "reachability", it only does per-pixel segmentation, it doesn't provide information about, for example, geometry or lanes. This can be computed using semantic segmentation algorithms e.g., Deep Neural Networks. We can see an example of road segmentation in Figure 82a.
- **Driving Corridor Prediction** - The goal is to estimate the corridor ahead (pixel mask) within which the vehicle is supposed to drive, and it may or may not consider obstacles. We can see an example of this prediction in Figure 82b. There is usually multiple driving corridors and the relevant one depends on the high-level plan.
- **Lane Marking Detection** - Roads are often subdivided into lanes using lane markings. Lane markings are man-made cue as they are made to help the driver, but they can also be used by the driving assistance system. The goal of the lane marking detection is to detect lane markings and fit parametric model (e.g., spline) to them. Important thing to note is that even the road markings are included in this process. This works well on highways, but in the city, lane markings detected are often not reliable. We can see an example of such a detection in Figure 82c.
- **Lane Detection** - The goal is to detect the lane we are driving in. Lane detection groups two lane markings (left and right from us) into one lane, and yields information about lane width and centerline (dashed). This can all be seen in Figure 82d. Again, relevant lane depends on the high-level plan.

[3]Lane keeping assist: https://www.youtube.com/watch?v=OQkdvi55woA

- **Freespace Estimation** - This is a purely geometric cue ("which places can be directly reached?"). The goal is to estimate places in image that can be reached without collision. It is computed using some geometric information that can come from laser measurements, from Lidar, or stereo information. Freespace Estimation does not require appearance cues such as lane markings, road texture etc. We can see this estimation in Figure 82e.
- **2D vs. 3D Representation** - Estimating quantities in the 2D image domain is not directly useful - it must be mapped into 3D space (or bird's eye view like in Figure 82f) where the vehicle is controlled (and where distances have a meaning). This mapping is possible given an estimate of the road surface (for example, 3D plane).



(a) Road Segmentation (b) Driving Corridor Prediction

(c) Lane Marking Detection (d) Lane Detection

(e) Freespace Estimation (f) Bird's Eye View Representation

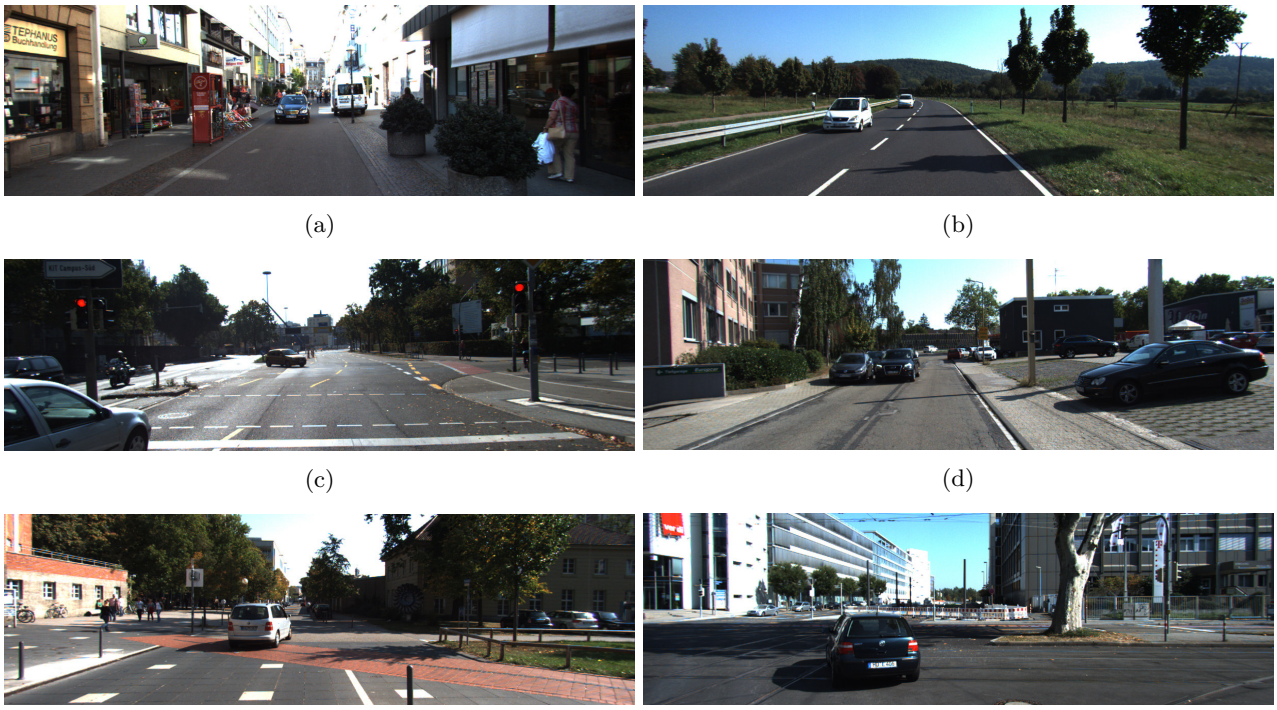
Figure 82: **Example outputs (different representations)** - Road and Lane detection problem. (a) Red color represents road pixels; (b) Red color represents the predicted corridor, while green color represents the obstacle on that corridor; (c) Red color represents the lane marking curves fitted; (d) Red lines: full - lane markings, dashed - centerline; (e) Red color represents the estimation, black lines represent some possible directions; (f) Red lines represent our lane mapped from another representation;

8.1.2 Ambiguities

Sometimes it can be hard to handle the mentioned representations. Some examples of ambiguities are:

- Road Segmentation
  - Sometimes sidewalk also looks like a road (Figure 83a)
  - Some areas are not designated for driving, but are not clearly marked (Figure 83e - poles)
- Lane Marking Detection, Lane Detection, Driving Corridor Detection
  - No lane markings in the image (non-existent, damaged or missing) (Figure 83a)
  - Temporary lane markings during road construction are there but the old road markings are still visible (Figure 83c)
  - Navigation in parking lots is not possible based on lane markings alone (Figure 83d)
  - Scene elements can resemble lane markings, but are not lane markings (Figure 83e)
  - Tram railroad can be easily confused with lane markings (Figure 83f)
- Freespace Estimation

– If the sidewalk or the curbs don't differ much in height when compared to the road, it can be taught to be drivable. (Figures 83a, 83d)
– Is the grass field by the road traversable or not? (Figure 83b)



(a) (b) (c) (d) (e) (f)

Figure 83: **Example images for ambiguities**

8.1.3 Overview

	Advantages	**Disadvantages**
Road Segment.	- Works in unstructred env. - Doesn't rely on accurate geometry	- Semantic ambiguities - Doesn't output path information
Driving Corridor	- More directly relevant to control - Also comprises obstacle inform.	- Depends on high-level plan - Ambiguous/difficult to estimate
Lane Marking	- Relatively easy inference task - Compact parametric models	- Markings missing/deteriorated - Only in structured env. (highway)
Lane	- Directly yields path / centerline - Compact parametric models	- Ambiguous/difficult to estimate - Only in structured env. (highway)
Freespace Est.	- Works in unstructred env. - Doesn't require semantics	- Relies on accurate geometry - Doesn't output path information

8.2 Road Segmentation

Problem: Given an RGB image as an input, the task is to predict a semantic class label for every pixel.
Solution: Deep Convolutional Image Segmentation [**?**][**?**].
We do this by using convolution, pooling and unsampling layers to predict per-pixel class labels. We have encoder-decoder structure where the encoder part includes convolution and pooling layers, and the decoder part includes unsampling and convolution layers. This can be seen in Figure 84. To train the parameters of this network we use the standard empirical risk minimization: we define a loss function (when having labeled images i.e., input-output pairs), and we want to minimize it. The loss function commonly used is cross-entropy loss. [**?**] The results can be seen in Figure 87a.

Another model from the literature [**?**] combines semantic segmentation with conditional random fields (CRF). The general idea is to make semantic segmentation coherent in space and time, where we have a densely connected CRF that spans both space and time. This can be seen in Figure 85. We can also see that the expectation formula contains the unary potential for each pixel (the first sum in the formula - this represents

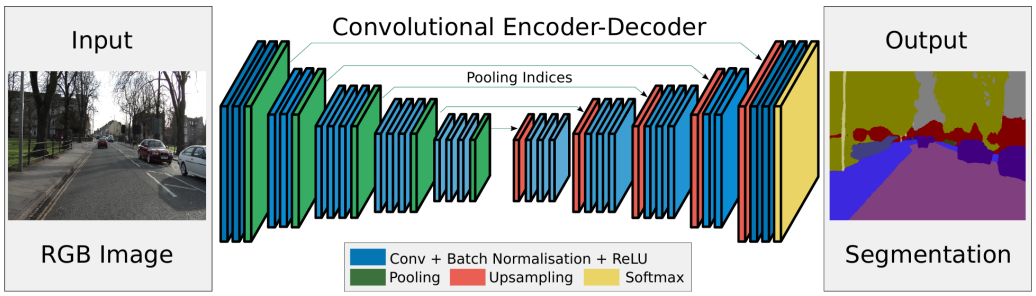


Fig. 2. An illustration of the SegNet architecture. There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to densify the feature map. The final decoder output feature maps are fed to a soft-max classifier for pixel-wise classification.

Figure 84: SegNet architecture for Image segmentation. Taken from [4].

which label should the pixel be labeled with) and we have the pairwise connections (second part of the formula) which intuitively tells us that if two pixels are close in space and/or time, they should probably have the same class label. In this paper, semantic segmentation is formulated as an optimization problem over multiple frames and there are temporal and spatial smoothness constraints via fully connected CRF model. The results can be seen in Figure 87b.

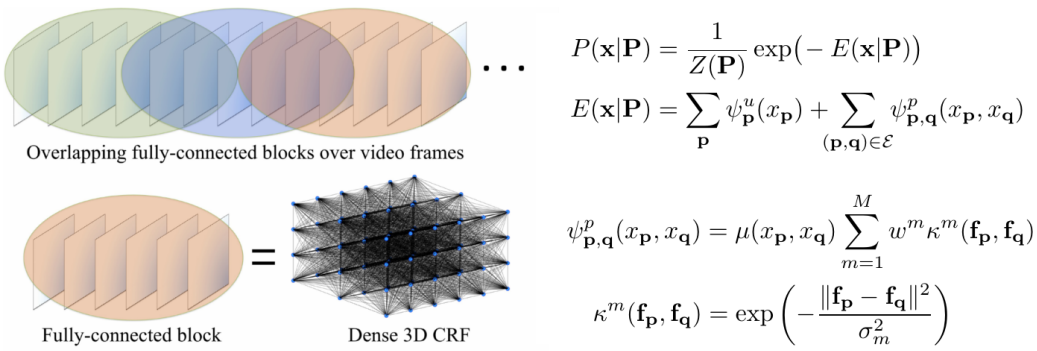


Figure 85: Semantic segmentation combined with CRF. Taken from [18].

We can also apply the idea of semantic segmentation in 3D directly, by parsing input images through a semantic segmentation network while at the same time computing a local map using SLAM techniques and then obtaining a semantic 3D map by joining the results.[**?**] This can be seen in Figure 86.

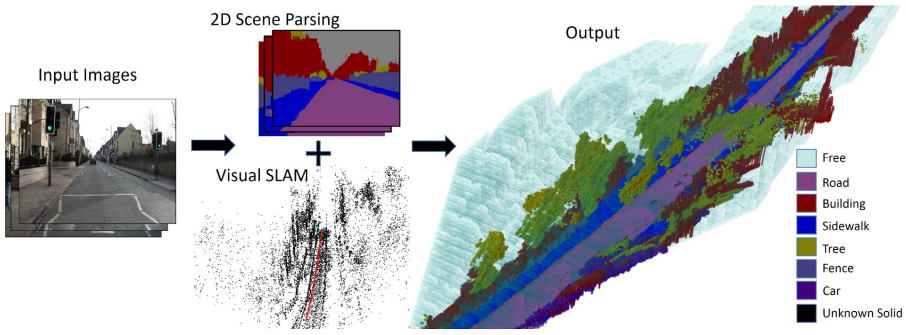
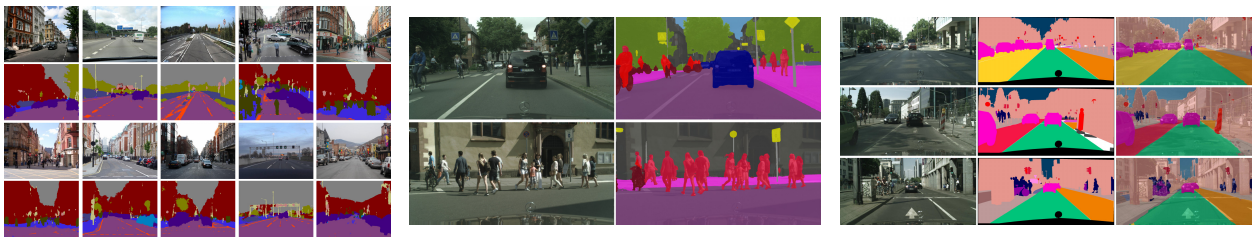


Figure 86: Joint semantic segmentation and 3D reconstruction. Taken from [**?**].

It is also possible to train a model for a fine-grained semantic segmentation specifically for a self-driving task [**?**] For example, instead of the label for road we can use the labels for ego/parallel/opposite lanes and obtain a more detailed semantic segmentation. This can be seen in Figure 87c, where green represents ego lane, yellow represents parallel lane and red represents opposite lane.



(a) Results taken from [**?**].

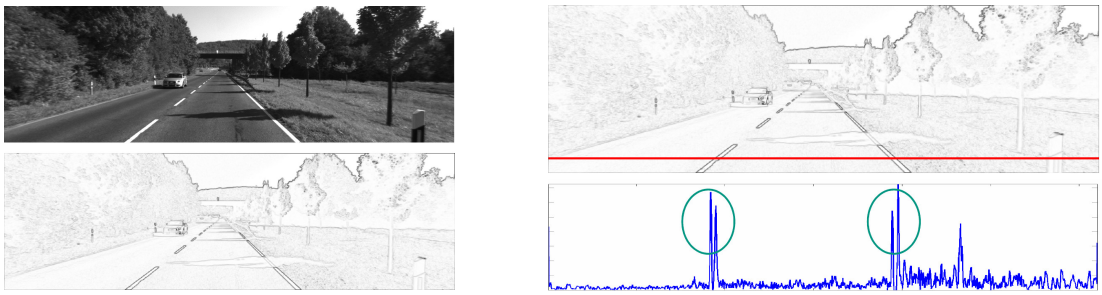
(b) Results taken from [**?**].

(c) Results taken from [**?**].

Figure 87: Semantic segmentation results - different methods.

8.3 Lane Marking Detection

The lane markings on the image are typically bright and are easy to distinguish from the background. This means that we can compute the gradient of the image and find the contour of these lane markings as strong gradient responses. For example, we can use simple horizontal kernel filter $[-1\ 0\ +1]$ and obtain an image that can be seen in Figure 88a. Alternatives to using these simple filters can be: steerable filters, specific templates (for example, if we know some road features or geometry in advance), learned features etc. Figure 88b shows an example of searching for large gradients along one image row - the red row marked is being searched and the gradients shown in blue are absolute value gradients. The gradient peaks can be clearly seen (usually two of them close to each other - one for the left side and one for the right side of the lane marking). We can use this information to filter outliers. If depth is available, we can filter the points that are not on the ground plane (e.g., by plane fitting). Also, we can remove the isolated points or points for which no opposite gradient exists in its vicinity (one can be seen in Figure 88b). The results of this simple algorithm can be seen in Figure 88c. We can further reject false positives using heuristics or robust fitting (e.g., RANSAC). Another thing we can



(a) Gradient Image

(b) Search for large gradients along image row

(c) Results of detection: green - left boundary of lane marking; red - right boundary of lane marking;

Figure 88: Simple algorithm for lane marking detection using gradients.

see in the results is that not all the lane markings were detected (the leftmost marking is not detected). This is a consequence of the gradient threshold chosen for the detection. If we set a lower threshold, we will be able to detect more lane markings but we will also get more false positives.

Since we are not interested in the lane markings in the image domain, but the bird's eye view, we need a way of mapping the lane markings (typically a set of pixels) from the image domain to the bird's eye view. This process is called **Inverse Perspective Mapping** (IPM) and it is illustrated in Figure 89.

In this process, each pixel point from the image plane (u, v) is mapped to the respective ground plane point (x, y), while (u, v) is represented in the image coordinate system (Image CS), and (x, y) is represented in the road coordinate system (Road CS). IPM is only possible if either the per pixel depth is known (because then we can unproject that into 3D), or if we have a model for the ground (e.g., plane) that is known.
The other components that can be seen in Figure 89 are $[\mathbf{R}\ \mathbf{t}] = [r_x\ r_y\ r_z\ t]$ - rigid body transformation, the camera coordinate system (Camera CS) and K - the calibration matrix (intrinsics that maps the camera coordinate to the respective image coordinate). Usually, the Road CS is directly below the Camera CS, so it

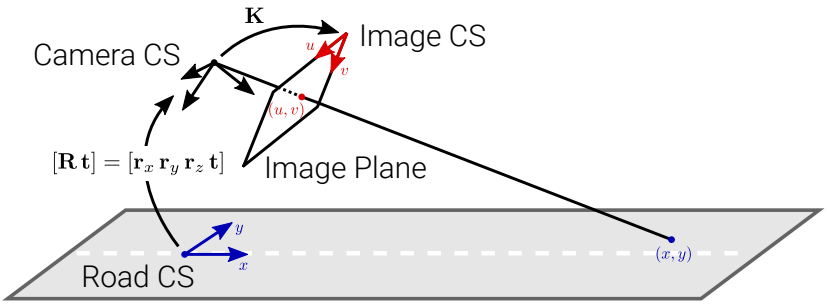


Figure 89: Inverse Perspective Mapping

is uniquely defined. All of these components together define a chain of operations that needs to be followed in order to obtain the mapping.
How can we obtain a ground plane (i.e., extrinsics $[\mathbf{R}\ \mathbf{t}]$*)?*
One solution is to assume we have a fixed plane or to fit a plane/curve surface to depth measurements.
Given $[\mathbf{R}\ \mathbf{t}]$, a pixel in the image can be mapped to the ground plane as follows:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \propto \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \overset{z=0}{\Rightarrow} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \propto \mathbf{K} \begin{bmatrix} \mathbf{r}_x & \mathbf{r}_y & \mathbf{t} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \begin{bmatrix} \mathbf{r}_x & \mathbf{r}_y & \mathbf{t} \end{bmatrix}^{-1} \mathbf{K}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

On the left side, we can see the standard equation of the pinhole camera model, where we take augmented vector of road coordinates $[x\ y\ z\ 1]^T$, then multiply it with the extrinsics $[\mathbf{R}\ \mathbf{t}]$, then with the intrinsics K and we arrive to the pixel coordinates $[u\ v\ 1]^T$. Because we assume the road is flat, we have $z = 0$, so the equation further simplifies. After this simplification, we have square matrices 3x3 so we can invert the equation to obtain the expression for calculating $[x\ y\ 1]^T$ as can be seen on the right side of the expression. IPM example can be seen in Figure 90. In Figure 90a we can also see that the pixels far away get heavily distorted (a lot of uncertainty there), but now, we have the lane markings in a bird's eye view perspective, which is useful for the driving task.

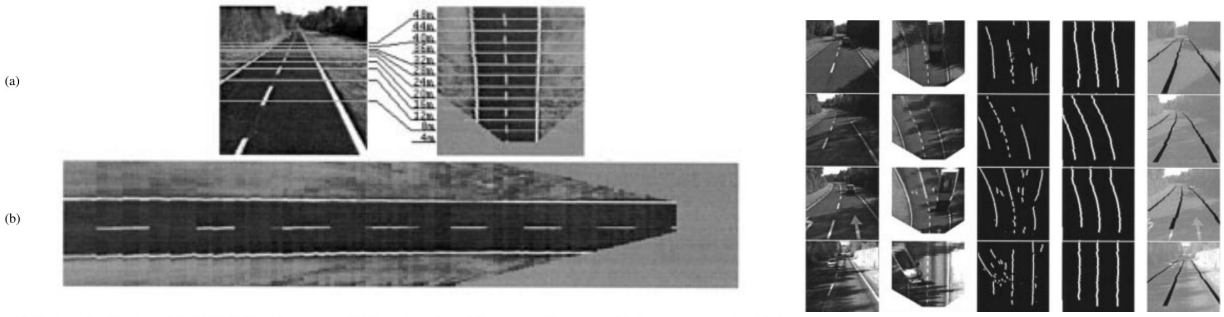


Fig. 8. (a) Horizontal calibration of the MOB-LAB vision system. (b) Rotated version of the remapped image considering an aspect ratio of 1 : 1.

(a) (b)

Figure 90: IPM examples, taken from: [**?**].

However, we still only have the pixels (just in the bird's eye view). We want to obtain a set of parametric curves, that we can further use in order to control the vehicle, because pixels are not sufficient for this task. Often low-dimensional parametric models are used for this (lines, polynomials, bezier curves, splines), and it is a multi-modal fitting problem (1 model per lane marking). Furthermore, the outliers must be handled and the model must be robust to noise.

One example of the algorithm for this parametric lane marking estimation problem can be seen in Figure 91. In this case, the lane markings have been detected and transformed into the bird's eye view and then a parametric spline is being fit onto the lane markings using RANSAC procedure. RANSAC works by sampling points on consecutive segments, fitting a spline to these points and evaluating how well this spline overlaps with the identified pixels for the lane markings. This is illustrated in Figure 91a, along with the scoring function for evaluation of the spline fitted. Figure 91b shows the results of this algorithm. We can notice that it is not so reliable, that it fails on intersections, and it does not detect the very left and the very right lane markings reliably.
We have described a very simple algorithm for the task of detecting the lane markings, but *can we do better*

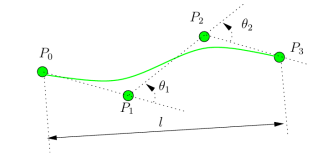


Fig. 7. Spline score computation.

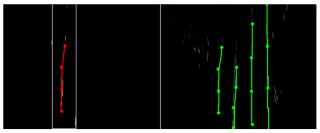


Fig. 8. RANSAC Spline fitting. Left: one of four windows of interest (white) obtained from previous step with detected spline (red). Right: the resulting splines (green) from this step

Algorithm 1 RANSAC Spline Fitting

for $i = 1$ to $numIterations$ **do**
  $points$=getRandomSample()
  $spline$=fitSpline($points$)
  $score$=computeSplineScore($spline$)
  if $score > bestScore$ **then**
    $bestSpline = spline$
  end if
end for

Fig. 10. Post-processing splines. Left: splines before post-processing in blue. Right: splines after post-processing in green. They appear longer and localized on the lanes.

(a) Algorithm description

(b) Results

Figure 91: Parametric Lane Marking Estimation Algorithm. *Taken from: [**?**]*

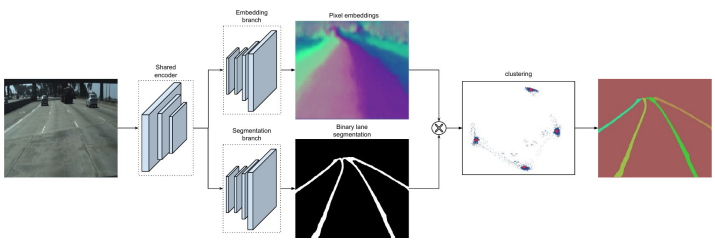
using deep learning?

We need to detect multiple lanes using deep learning, so we can not use the standard cross-entropy loss (used for semantic segmentation) as it is not good enough. One possible solution is to predict per-pixel feature embeddings and cluster those features in feature space [**?**] (instead of predicting classes) - End-to-End Lane Marking Detection. This process can be seen in Figure 92a. As an input we have an image that goes through a shared encoder that produces a feature embedding, this encoded information then goes through the two decoders. The first decoder produces per pixel feature embbedings, while the second decoder does binary lane marking segmentation. Then, we mask the per pixel feature embbedings got from the first decoder with the lane marking segmentation got from the second decoder in order to only consider the pixels that are on the lane markings. We train this model such that different lane markings are far away in the embedding space, so when we apply a clustering algorithm to the feature space, we can easily determine which pixels belong to which lane. Important thing to note is that this doesn't depend on the number of lane markings. The loss function used in this learning algorithm is:

$$L_{var} = \frac{1}{C}\sum_{c=1}^{C}\frac{1}{N_c}\sum_{i=1}^{N_c}\left[\|\mu_c - x_i\| - \delta_v\right]_+^2$$

$$L_{dist} = \frac{1}{C(C-1)}\sum_{c_A=1}^{C}\sum_{c_B=1, c_A\neq c_B}^{C}\left[\delta_d - \|\mu_{c_A} - \mu_{c_B}\|\right]_+^2$$

This is a clustering loss with two components. Since we are trying to learn an embedding, the first component tells us that pixels belonging to the same lane should be pulled together, and the second component tells us that pixels belonging to the different lanes should be pushed away from each other. Again, we assume that the lane labels are available, but we are not trying to directly predict the instance labels, because we don't know the number of instances. Instead, we use this ground truth to define these loss functions. The results are shown in Figure 92b. We can see that this algorithm learns to detect the lanes that are further away a lot better than the previously mentioned non-learning algorithm. End-to-End lane marking detection is used as an intermediate process, and the curve fitting still needs to be done afterwards. This can be seen in the bottom row of the results.



(a) Model description

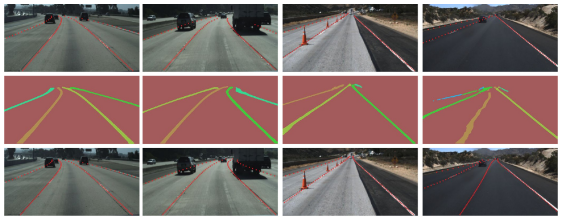


Fig. 5. Visual results. *Top row*: ground-truth lane points. *Middle row*: LaneNet output. *Bottom row*: final lane predicts after lane fitting.

(b) Results

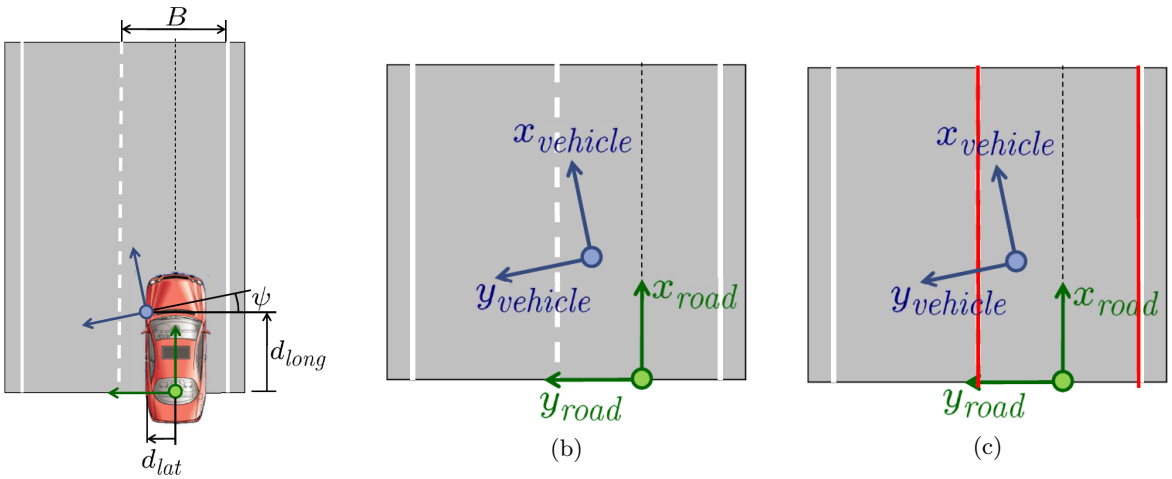
Figure 92: End-to-End Lane Marking Detection. *Taken from: [**?**]*

8.4 Lane Detection

In the previous section, we have seen how we can detect the lane markings and project them into the bird's eye view perspective. However, knowing just where the lane markings are, is not sufficient for controlling the vehicle. We need to aggregate the lane markings into a single lane, such that a car can, for example, follow the centerline. In this section, we will describe a very simple model for this task (linear road model). Such a model does not apply everywhere, but it can be applied on driving straight on a highway.

A *model for straight lanes* is a geometric model, and the question is: *can we fit a single model for the entire lane?*
Parameters that need to be modelled are: lane width $B \in \mathbb{R}^+$, lateral vehicle offset (distance) wrt. centerline $d_{lat} \in \mathbb{R}$, longitudinal position (distance) $d_{long} \in \mathbb{R}$, yaw angle $\psi \in [-\pi, \pi]$. All of these parameters can be seen in Figure 93a. We can model the state with these 4 parameters as:

$$\mathbf{s} = \begin{bmatrix} B & d_{long} & d_{lat} & \psi \end{bmatrix}^T$$



(a) (b) (c)

Figure 93: Simple lane detection model.

The goal is to obtain these parameters through simply observing the lane markings. First, lets observe the transformations between the vehicle and road coordinate systems (both of them shown in Figure 93b). We have:
Transformation vehicle → road coordinates:

$$\begin{bmatrix} x_{road} \\ y_{road} \end{bmatrix} = \begin{bmatrix} d_{long} \\ d_{lat} \end{bmatrix} + \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix} \begin{bmatrix} x_{vehicle} \\ y_{vehicle} \end{bmatrix}$$

Transformation road → vehicle coordinates:

$$\begin{bmatrix} x_{vehicle} \\ y_{vehicle} \end{bmatrix} = \begin{bmatrix} \cos(\psi) & \sin(\psi) \\ -\sin(\psi) & \cos(\psi) \end{bmatrix} \left(\begin{bmatrix} x_{road} \\ y_{road} \end{bmatrix} - \begin{bmatrix} d_{long} \\ d_{lat} \end{bmatrix} \right)$$

Note: Different colors correspond to the concepts in the figures.
In the road coordinate system, we can easily describe the position of the lane markings (Figure 93c), and using the mentioned transformations we can easily obtain the respective vehicle coordinates ($l \in \mathbb{R}$ is the path length):

$$\begin{bmatrix} x_{road} \\ y_{road} \end{bmatrix} = \begin{bmatrix} \ell \\ \pm\frac{B}{2} \end{bmatrix} \longrightarrow \begin{bmatrix} x_{vehicle} \\ y_{vehicle} \end{bmatrix} = \begin{bmatrix} \cos(\psi) & \sin(\psi) \\ -\sin(\psi) & \cos(\psi) \end{bmatrix} \begin{bmatrix} \ell - d_{long} \\ \pm\frac{B}{2} - d_{lat} \end{bmatrix}$$

Further, eliminating l and assuming $\psi \approx 0$ yields:

$$y_{vehicle} = -\psi \cdot x_{vehicle} \pm \frac{B}{2} - d_{lat}$$

We now have the relationship between x and y in the vehicle coordinate system, which means we can use observations to estimate the parameters ψ, B and d_{lat}. So this results in simple regression problem per frame:

- Given N^L points on the left marking (x_i^L, y_i^L) and
  Given N^R points on the right marking (x_j^R, y_j^R)
- Minimize:

$$\begin{aligned} B^*, d_{lat}^*, \psi^* &= \underset{B, d_{lat}, \psi}{\operatorname{argmin}} \sum_i (e_i^L)^2 + \sum_j (e_j^R)^2 \\ e_i^L &= \left(-\psi \cdot x_i^L + \frac{B}{2} - d_{lat}\right) - y_i^L \\ e_j^R &= \left(-\psi \cdot x_j^R - \frac{B}{2} - d_{lat}\right) - y_j^R \end{aligned}$$

- Closed form solution:

$$\begin{bmatrix} \frac{1}{4}(N^L + N^R) & \frac{1}{2}(-N^L + N^R) & \frac{1}{2}(-\sum x_i^L + \sum x_j^R) \\ \frac{1}{2}(-N^L + N^R) & N^L + N^R & \sum x_i^L + \sum x_j^R \\ \frac{1}{2}(-\sum x_i^L + \sum x_j^R) & \sum x_i^L + \sum x_j^R & \sum (x_i^L)^2 + \sum (x_j^R)^2 \end{bmatrix} \begin{bmatrix} B \\ d_{lat} \\ \psi \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(\sum y_i^L - \sum y_j^R) \\ -\sum y_i^L - \sum y_j^R \\ -\sum x_i^L y_i^L - \sum x_j^R y_j^R \end{bmatrix}$$

In summary, in order to detect the lanes: we first need to detect the lane markings in image, further, we need to transform the positions of the lane markings into vehicle coordinates via IPM and then we need to estimate pose parameters and lane width using this information. Another thing to note is that model parameter d_{long} is not observable, thus it is set to 0 in this simple model.

8.5 Lane Tracking

In the previous section we have seen how we can estimate the lane using observations from a single frame, and the problem with this is that it is not robust, as we might ignore significant information from the previous frame(s). Also, we have seen that we can not estimate the longitudinal position along the lane, and that brings us to **lane tracking**. In lane tracking, we want to formulate the estimation problem, not only for a single frame, but for multiple frames. We want to find somewhat optimal estimation with respect to multiple observations, in hope of getting a better estimate.

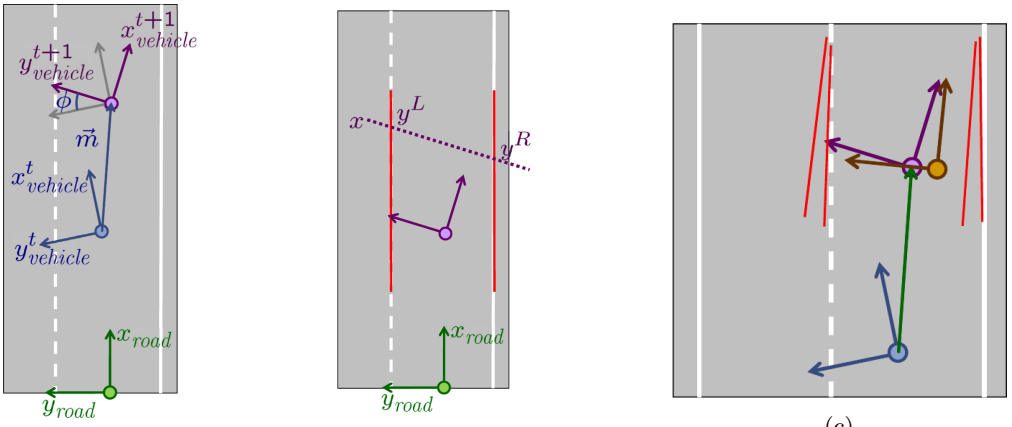


Figure 94: Model for Straight Lanes: Incremental Localization.

Again, we will only consider a ***Model for Straight Lanes***, and this localization is called *Incremental Localization*, because we are trying to incrementally estimate the parameters (track them over time). This is illustrated in Figure 94a where we have the vehicle coordinate system at time t in blue and at time $t+1$ in purple.
The rigid body motion that translates the vehicle coordinates at time t to the vehicle coordinates at time $t+1$ is given via the translation vector $\vec{m}$ and the relative change in the heading ϕ. We use the same state vector as before, $\mathbf{s} = \begin{bmatrix} B & d_{long} & d_{lat} & \psi \end{bmatrix}^T$, just now we want the estimation over time. The fist thing that we can do is define the **state transition** equations as follows:

$$\begin{aligned} \begin{bmatrix} d_{long}^{t+1} \\ d_{lat}^{t+1} \end{bmatrix} &= \begin{bmatrix} d_{long}^t \\ d_{lat}^t \end{bmatrix} + \begin{bmatrix} \cos(\psi^t) & -\sin(\psi^t) \\ \sin(\psi^t) & \cos(\psi^t) \end{bmatrix} \mathbf{m} \\ &\approx \begin{bmatrix} d_{long}^t \\ d_{lat}^t \end{bmatrix} + \begin{bmatrix} 1 & -\psi^t \\ \psi^t & 1 \end{bmatrix} \mathbf{m} \\ \psi^{t+1} &= \psi^t + \phi \\ B^{t+1} &= B^t \qquad \text{(constant road width)} \end{aligned}$$

These equations show us how the state transitions over time. With the assumption we made (second row) we get the equations in the following linear form:

$$\mathbf{s}^{t+1} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -m_y \\ 0 & 0 & 1 & m_x \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{A}^t} \mathbf{s}^t + \underbrace{\begin{bmatrix} 0 \\ m_x \\ m_y \\ \phi \end{bmatrix}}_{\mathbf{u}^t}$$

where $\mathbf{s}^t$ is the state at time t and $\mathbf{s}^{t+1}$ is the state at time $t+1$. The other thing we want to consider are the **observations** where we have:N^L points on left marking (x_i^L, y_i^L) and N^R points on right marking (x_j^R, y_j^R) (Figure 94b). Now, y^L and y^R are given to us, but can be determined from $\mathbf{s}$ and x:

$$\begin{aligned} y^L &= -\psi \cdot x + \frac{B}{2} - d_{lat} \\ y^R &= -\psi \cdot x - \frac{B}{2} - d_{lat} \end{aligned}$$

Or, equivalently:

$$\underbrace{\begin{bmatrix} y_1^L \\ \vdots \\ y_{N^L}^L \\ y_1^R \\ \vdots \\ y_{N^R}^R \end{bmatrix}}_{=\mathbf{y}^{t+1}} = \underbrace{\begin{bmatrix} \frac{1}{2} & 0 & -1 & -x_1^L \\ \vdots & \vdots & \vdots & \vdots \\ \frac{1}{2} & 0 & -1 & -x_{N^L}^L \\ -\frac{1}{2} & 0 & -1 & -x_1^R \\ \vdots & \vdots & \vdots & \vdots \\ -\frac{1}{2} & 0 & -1 & -x_{N^R}^R \end{bmatrix}}_{=\mathbf{C}^{t+1}} \underbrace{\begin{bmatrix} B \\ d_{long} \\ d_{lat} \\ \psi \end{bmatrix}}_{=\mathbf{s}^{t+1}}$$

Now, we have a linear state transition model and a linear observation model, so the entire state space representation (combining these two) can be described with two linear equations:

$$\begin{aligned} \mathbf{s}^{t+1} &= \mathbf{A}^t \, \mathbf{s}^t + \mathbf{u}^t \\ \mathbf{y}^{t+1} &= \mathbf{C}^{t+1} \, \mathbf{s}^{t+1} \end{aligned}$$

So, for this estimation problem, we have a linear dynamic model, we have a linear measurement model and if we now assume Gaussian noise, we can estimate this optimally and in closed form using one of the famous state estimation algorithms: Kalman filter [**?**]. This algorithm is able to solve simple Gaussian linear systems, such as the mentioned one, efficiently and optimally. This is the main reason why we simplified the original system in order to get to these linear equations.

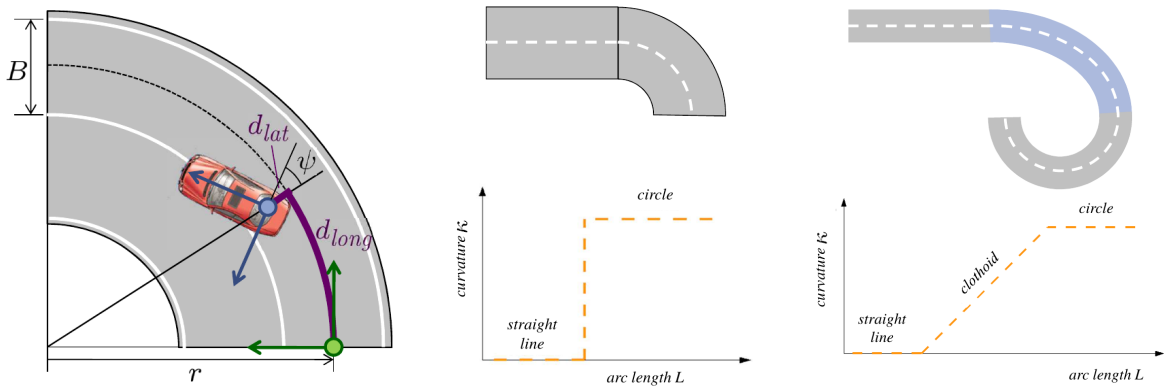
In practice, Incremental Localization works as follows (Figure 94c):

- **Detection Step** - First, we need to detect the road markings in the image.
- **Inverse Perspective Mapping** - Then, we need to transform the detected road markings into the vehicle coordinate system, using IPM.
- **Prediction Step** - Further, we have a prediction step, where we can, e.g., determine how the vehicle is moving using visual odometry.
- **Innovation Step** - Last, we should update the state wrt. the observed markings. This needs to be done because, as we can see in the Figure 94c, that the calculated state (purple) does not always match with the position obtained from the lane markings - current measurement, so we do the correction (brown).

Unfortunately, straight roads are not sufficient to model all the driving scenarios, so another model we should consider is a ***Model for Circular Lanes***. Parameters that need to be modelled here are: lane width $B \in \mathbb{R}^+$, signed radius $r \in \mathbb{R}$ or curvature, $\kappa = \frac{1}{r}$, lateral vehicle offset $d_{lat} \in \mathbb{R}$, longitudinal position $d_{long} \in \mathbb{R}$ and yaw angle $\psi \in [-\pi, \pi]$. These parameters can be seen in Figure 95a. Now, we have the state as:

$$\mathbf{s} = \begin{bmatrix} B & r & d_{long} & d_{lat} & \psi \end{bmatrix}^T$$

This model can not be represented using linear equations, thus it requires more sophisticated state filtering techniques such as: Extended Kalman Filter (EKF), Unscented Kalman Filter (UKF) or Particle Filter.
Note: When $r \to \infty$, we should get a straight lane model.



(a) Model for Circular Lanes (b) (c)

Figure 95: Model for Circular Lanes

However, adding the circular trajectories is again not enough to cover all the possible lane curves that can occur while driving. For example, as seen in Figure 95b, if we move from a straight road segment to a circular road segment (which seems plausible), the curvature increases with a step function, which means that acceleration jumps. This however does not happen in real-world scenarios. We usually have a smooth transition from a straight road to circular road. This is why, in practice, we mostly use the ***Clothoid Model***, that can be seen in Figure 95c. The example where this scenario can be found is exiting the highway.

Mathematically, a clothoid is a curve (in Figure 96) whose curvature linearly depends on the arc length:

$$\frac{1}{r(\ell)} = \kappa(\ell) = \kappa_0 + \kappa_1 \ell.$$

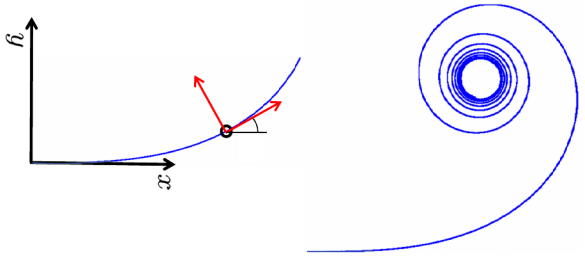


Figure 96: Clothoid

This means that yaw angle is:

$$\psi(\ell) = \int_0^\ell \kappa(\lambda) d\lambda = \kappa_0 \ell + \frac{1}{2}\kappa_1 \ell^2,$$

and the position:

$$\begin{aligned} x(\ell) &= \int_0^\ell \cos(\psi(\lambda))\, d\lambda \\ y(\ell) &= \int_0^\ell \sin(\psi(\lambda))\, d\lambda. \end{aligned}$$

If we assume that yaw angle ψ is small (close to 0), we get:

$$\begin{aligned} x(\ell) &= \int_0^\ell \cos(\psi(\lambda))\, d\lambda \approx \int_0^\ell 1\, d\lambda \\ y(\ell) &= \int_0^\ell \sin(\psi(\lambda))\, d\lambda \approx \int_0^\ell \psi(\lambda)\, d\lambda \end{aligned}$$

with $\psi(\lambda) = \kappa_0 \lambda + \frac{1}{2}\kappa_1 \lambda^2$. Thus, we obtain a 3rd order polynomial:

$$\begin{aligned} x(\ell) &\approx \ell \\ y(\ell) &\approx \frac{1}{2}\kappa_0 \ell^2 + \frac{1}{6}\kappa_1 \ell^3. \end{aligned}$$

This has been done in the seminal paper [**?**] by self-driving pioneer Ernst Dickmanns. In this paper he uses a visual recursive state estimation and a clothoid model in 3D. He models horizontal and vertical road curvature through clothoids (as can be seen on the left part of Figure 97). He considers a dynamic bicycle motion model (center of Figure 97), and uses a projection-based measurement model instead of IPM (right part of Figure 97).

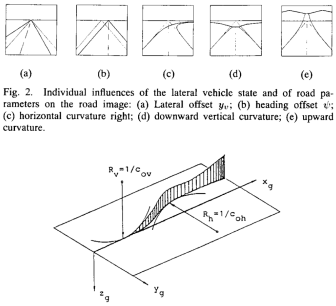


Fig. 2. Individual influences of the lateral vehicle state and of road parameters on the road image: (a) Lateral offset y_0; (b) heading offset ψ; (c) horizontal curvature right; (d) downward vertical curvature; (e) upward curvature.

Fig. 3. Spatial road segment with horizontal and vertical curvature.

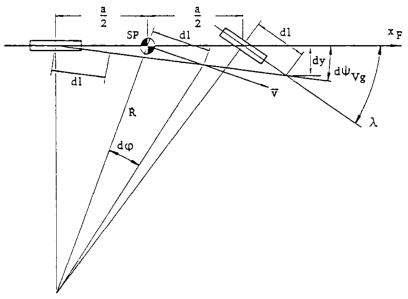


Fig. 8. Steering kinematics.

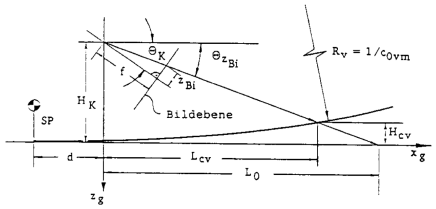


Fig. 6. Vertical road and mapping geometry.

Figure 97: Model for Circular Lanes

Figure 98: Recursive 3D Road and Relative Ego-State Recognition. *Taken from: [**?**].*

8.5.1 Summary

We have seen that road and lane detection are important for advanced driver assistant systems (ADAS) and self-driving. They allow to localize locally with respect to the road without requiring a map. Also, multiple different representations exist with their advantages/disadvantages. We have further discussed road segmentation and lane marking detection, seen how lanes can be estimated from lane markings, and discussed how we can incrementally track lanes over time. We haven't addressed geometric freespace estimation which was introduced in this lecture, but this will be discussed in the next lecture in the context of 3D reconstruction.

9 Reconstruction and Motion

This lecture deals with 3D scene reconstruction from stereo cameras, basic scene understanding using freespace and stixels and basic motion analysis using optic/scene flow.

9.1 Stereo Matching

Why do we require 3D perception abilities in a self driving system? For example, consider the problem of driving a car in a lane while a vehicle in front is performing a complex maneuver such as parking. To pass the vehicle safely, and for all other navigation tasks, we need precise knowledge of the environment's geometry. Self driving cars achieve this by combining multipe 3D sensors of different types (see Fig. 99). We can classify them as *active* or *passsive* sensors based on whether they emit sound or light waves to perform their function.

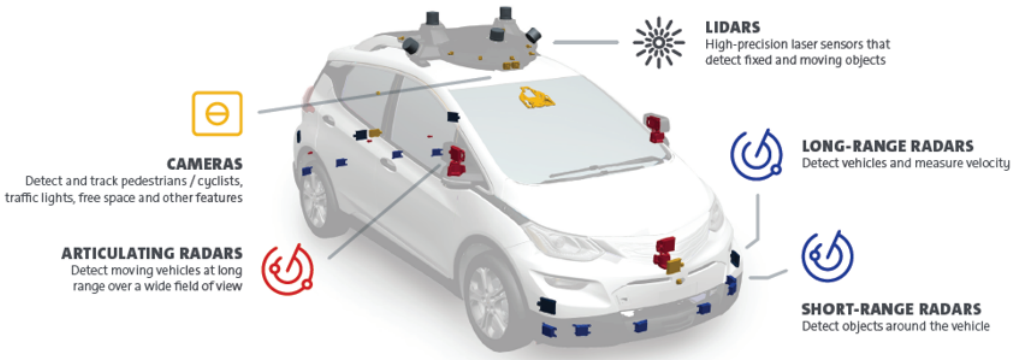


Figure 99: Self driving cars come equipped with a variety of sensors such as radar, lidar and cameras.

1. Active Sensors : They typically emit sound/radio/light waves,
   - Ultrasonic sensors are short range (5 m) sensors that are useful for tasks such as parking or blind spot detection.
   - Radar sensors are long range (300 m) but low resolution sensors. An example application is Adaptive cruise control (ACC) which is the task of automatically adjusting the speed of the vehicle to maintain a safe distance from vehicles ahead.
   - Lidar sensors are long range (100 m) sensors with mid-level resolution. They are typically used only in self-driving vehicle prototypes due to their high cost.
2. Passive sensors : These sensors don't emit any waves, but measure the signal coming from the environment.
   - Cameras are an example of mid range (50 m) passive sensors. Although they are cheap and high resolution, they have disadvantages such as requiring an additional processing step to compute the depth map whose accuracy depends on the distance and texture of a 3D point. To estimate the 3D geometry of the environment we need *stereo cameras* as we require two images of the same scene from different viewpoints at the same time. The other option is to capture consecutive images of the scene from a moving camera (For example : using Structure-from-motion techniques, See CV Lecture 3)

In principle, passive sensors should be sufficient to solve driving - as humans do. Passive sensors have the advantage of being cheaper and do not suffer from interference, but they instead have their own problems. In practice self-driving systems use different types of sensors and fuse their measurements for more robust prediction.

9.1.1 Binocular Stereo Matching Overview

We are given as input a *stereo image pair*, 2 images of the same scene captured simultaneously from two laterally dispaced cameras. Our ultimate goal is to estimate the depth of every 3D point visible in the image. However, to do so we first consider an intermediate quantity - the *disparity*.

The disparity of a pixel in a reference image is the horizontal displacement of its correspondence in a target image wrt the reference coordinates. Consider the example of two stereo images L, R. Let $\mathbf{x} \in \mathbb{R}^3$ be a 3D point which is visible in both images at $\mathbf{x}_L = (x_L, y_L)$ in image L and $\mathbf{x}_R = (x_R, y_R)$ in image R. Note that a pixel

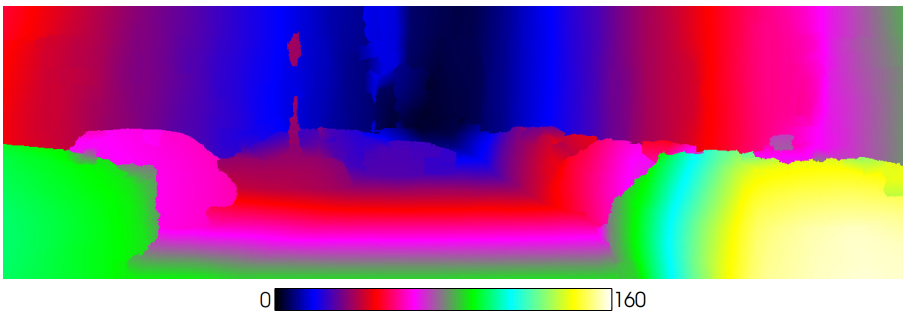
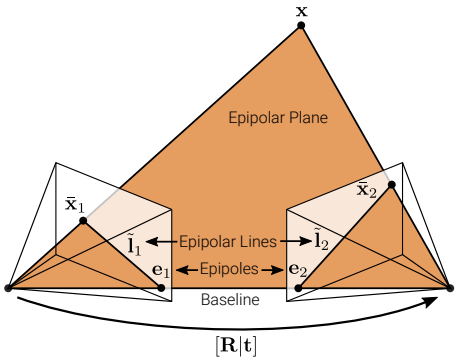
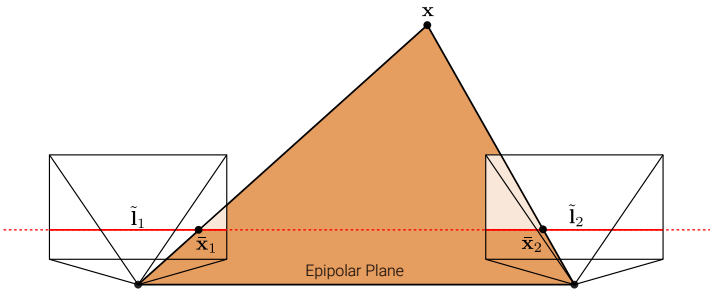


Figure 100: Disparity of a road scene in the KITTI dataset.



(a) 3D point $\mathbf{x}$ and a pair of cameras form the epipolar plane. Here $\tilde{\mathbf{l}}_1, \tilde{\mathbf{l}}_2$ are the epipolar lines corresponding to $\mathbf{x}$. $\mathbf{e}_1, \mathbf{e}_2$ are the epipoles, through which every epipolar line passes.



(b) A camera pair is *rectified* when their image planes are parallel. Observe that the epipolar lines $\tilde{\mathbf{l}}_1, \tilde{\mathbf{l}}_2$ are horizontal. Images taken with an initial relative rotation (e.g. Fig. 101a) can be warped into a rectified view if they are calibrated.

in L would be displaced to the *left* in R. The disparity would then be $x_R - x_L < 0$ negative for the left image, and positive for the right image. Computing the disparity at every pixel gives us the disparity map.

If we observe Fig. 100 we see that the disparity is smaller for points farther away from the camera than nearby points. In fact the disparity is inversely proportional to the depth, as we see in Section 9.1.4. This allows us to derive the *scene depth* from a stereo pair - this is called a *2.5D depth map* because we only know the depths of 3D points along the viewing direction, but not the entire 3D geometry. However, we can fuse multiple 2.5D maps into large 3D reconstructions, for example in a *visual odometry* task. Next we introduce the Binocular Stereo Matching Task and the components of its pipeline.

The goal of the **Binocular Stereo Matching** task is to construct a **dense** 2.5D disparity map from 2 images of a static scene. This is done with thee following pipeline.

1. **Calibrate cameras** intrinsically (i.e. the calibration matrix $\mathbf{K}$ comprising of the focal length, principal offsets, skews etc is known) and extrinsically (i.e. the relative rotation $\mathbf{R}$ and translation $\mathbf{t}$ between the views is known).
2. **Rectify stereo images**, i.e. we warp each image so that they correspond to a pair of parallel views.
3. **Compute disparity map** from the recitified images using *block matching*.
4. **Remove outliers** using left-right consistency.
5. **Obtain depth** from disparity using triangulation.
6. **Construct 3D representation** (volumetric fusion/meshing)

9.1.2 Epipolar Geometry and Image Rectification

Fig. 101a illustrates the setting for epipolar geometry. The two cameras C_1, C_2 at O_1, O_2 and the 3D point of interest $\mathbf{x} \in \mathbb{R}^3$ define the *epipolar plane*. The intersection of the epipolar plane with the image planes are the *epipolar lines* $\tilde{\mathbf{l}}_1, \tilde{\mathbf{l}}_2$ wrt $\mathbf{x}$. The intersection of the epipolar lines with the *baseline* (i.e. the line O_1O_2) are the *epipoles* $\mathbf{e}_1, \mathbf{e}_2$. The epipoles are independent of $\mathbf{x}$ since all epipolar lines intersect at the epipoles.

Suppose the world coordinate system is the same as C_1, andd let the relative pose from C_1 coordinates to C_2 coordinates be $\mathbf{R}, \mathbf{t}$. Let $\mathbf{K}_1, \mathbf{K}_2$ be the calibration matrices for C_1, C_2 respectively. Then the projections

of $\mathbf{x}$ to the two image planes given by $\bar{\mathbf{x}}_1 \propto \mathbf{K}_1\mathbf{x}$ and $\bar{\mathbf{x}}_2 \propto \mathbf{K}_2(\mathbf{R}\mathbf{x}+\mathbf{t})$. Note that $\bar{\mathbf{x}}_i$ are 2D *homogeneous coordinates* in $\mathbb{R}^3$.

Starting in the opposite direction with image coordinates $\bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2$, we can invert the projections to get the *ray directions* in each camera's own coordinate system given by $\tilde{\mathbf{x}}_i = \mathbf{K}_i^{-1}\bar{\mathbf{x}}_i \in \mathbb{R}^3$. Note that these are *not* 2D homogeneous coordinates, contrary to usual notation. Now starting with $\mathbf{K}_2^{-1}\bar{\mathbf{x}}_2 \propto \mathbf{R}\mathbf{x}+\mathbf{t} = \mathbf{R}\alpha\mathbf{K}_1^{-1}\bar{\mathbf{x}}_1 + t$, where α is a proportionality constraint, we can eventually derive the epipolar constraint which relates the 2 image correspondences,

$$\tilde{\mathbf{x}}_2^\top \tilde{\mathbf{E}}\,\tilde{\mathbf{x}}_1 = 0 \quad \text{with} \quad \tilde{\mathbf{x}}_i = \mathbf{K}_i^{-1}\,\bar{\mathbf{x}}_i \tag{124}$$

$\tilde{\mathbf{E}}$ is called the essential matrix and is given by $[\mathbf{t}]_\times\mathbf{R}$. Note that correspondence of $\bar{\mathbf{x}}_1$ lies on the line $\tilde{\mathbf{l}}_2 = \tilde{\mathbf{E}}\tilde{\mathbf{x}}_1$, since (124) $\Rightarrow \tilde{\mathbf{x}}_2^\top\tilde{\mathbf{l}}_2 = 0$. In fact $\tilde{\mathbf{l}}_2$ is an *epipolar line*. We can see this since $\mathbf{e}_2 = -\mathbf{t}$ (See Fig. 101a), thus

$$-\mathbf{t}^\top(\mathbf{t}\times\mathbf{R})\tilde{\mathbf{x}}_1 = 0 \tag{125}$$

since $\mathbf{t}\times\mathbf{R}$ is orthogonal to $\mathbf{t}$. Thus starting with a pixel on image 1 $\bar{\mathbf{x}}_1$, we need only search along the epipolar line on the second image. This turns the correspondence search from a 2D problem ($640\times480 \approx 300k$ pixels) to a much simpler 1D problem (~ 640 pixels).

We further simplify the correspondence search by *rectifying* the images. An image pair is already rectified if their *image planes are co-planar* i.e. $\mathbf{R} = \mathbf{I}$. Thus the cameras are only translated. In this ideal scenario, the epipolar lines are horizontal (and hence parallel) and the epipoles are at infinity. Fig. 101b shows the epipolar geometry for rectified cameras.

In practice, zero rotation is hard to achieve. When we maps both images to a common plane parallel to the baseline this is called *rectification*. To do this we need the essential matrix $\tilde{\mathbf{E}}$ and the calibration matrices $\mathbf{K}_i$. Conceptually this involves reversing the projection to get the 3D coordinate (upto a factor) and then projecting it onto the common plane, but for more details see CV Lecture 4.

9.1.3 Block Matching

The next problem is finding pixel correspondences between the two images, which is simplified to searching along a horizontal line due to rectification. To actually decide whether two pixels are similar a single pixel is not informative enough, and so we consider a *patch* centered around the pixel.

Around each pixel we consider a $K\times K$ window flattened to a vector. Denote by $\mathbf{w}_L, \mathbf{w}_R : \Omega \to \mathbb{R}^{K^2}$ the patch (arrays) for the left and right images respectively, where Ω denotes the image domain. The next step is to evaluate the similarity between two patches $\mathbf{w}_L(x_1, y_1), \mathbf{w}_R(x_2, y_2)$ which we can do using a variety of similarity metrics,

1. Sum of squared differences (SSD) :

$$\text{SSD}(x, y, d) = \|\mathbf{w}_L(x, y) - \mathbf{w}_R(x-d, y)\|_2^2 \tag{126}$$

2. Zero Normalized Cross-Correlation (ZNCC) : $\bar{\mathbf{w}}$ denotes the mean of $\mathbf{w}$, i.e. the mean patch.

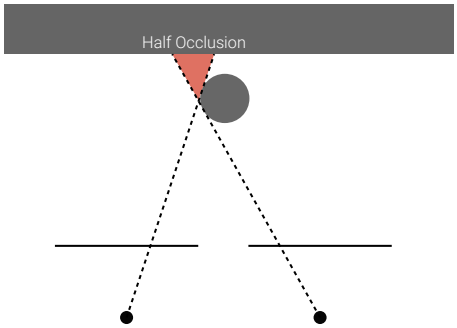
$$\text{ZNCC}(x, y, d) = \frac{(\mathbf{w}_L(x, y) - \bar{\mathbf{w}}_L(x, y))^T\,(\mathbf{w}_R(x-d, y) - \bar{\mathbf{w}}_R(x-d, y))}{\|\mathbf{w}_L(x, y) - \bar{\mathbf{w}}_L(x, y)\|_2\,\|\mathbf{w}_R(x-d, y) - \bar{\mathbf{w}}_R(x-d, y)\|_2} \tag{127}$$

Numerous other similarity metrics exist (see, e.g., Szeliski, Chapter 12.3).
The complete *block matching* algorithm can be summarised as follows

1. Choose disparity range $[0, D]$.
2. For all pixels $\mathbf{x} = (x, y)$ compute the best disparity ("winner-takes-all" solution).
3. Do this for both images, apply *left-right consistency check* to remove outliers.

Note that for a pixel at (x, y) in the left image we check in the interval $[x-D, x]$ and for the right image $[x, x+D]$. Block matching however suffers from several problems,

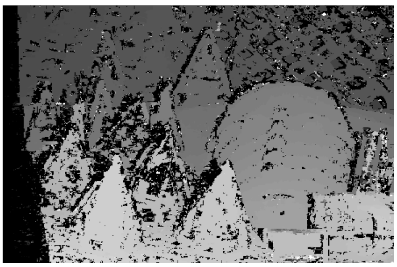
1. *Half occlusions* violate the assumption that each point is visible in both images. This may occur in a situation like Fig. 102a or if the point is outside of the field of vision of one camera. Half occlusions result in no matches or incorrect matches.
2. Block matching assumes that all pixels within a window are displaced by the same disparity - hence corresponding patches should be similar. However *slanted surfaces* and *depth discontinuities* violate this so-called (*fronto-parallel*) assumption. This results in ambiguity in matching.



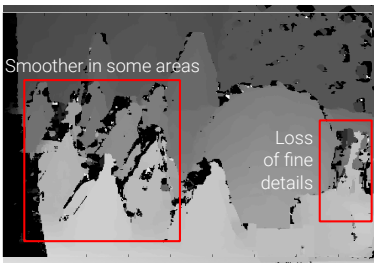
(a) Some points may be visible in one view, but occluded in the other. These are called *half-occlusions*. This results in noise in the disparities.

Window Size: 11x11 Pixels

(b) Using large window sizes results in the foreground disparity bleeding into the background. This is because the edge at the boundary dominates the feature vector at pixels near the boundary.

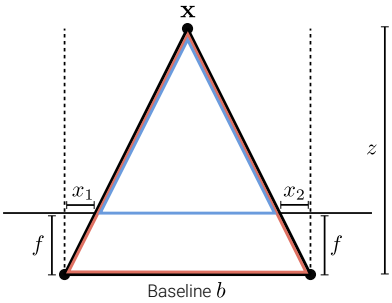


Window Size: 5x5 Pixels



Window Size: 11x11 Pixels

(a) Small window sizes lead to noisy results, while large window sizes lead to oversmooth results and border bleeding.



(b) This figure shows the top view of the epipolar plane for rectified cameras. The two triangles are similar, which can be exploited to compute the depth z from the disparity $d = x_1 - x_2$

3. *Border bleeding* (see Fig. 102b occurs when points/pixels at different disparities (and depths) at borders are assigned the same disparity. Example : In case of a textureless surface, the pixels adjacent to the border although are projections of different points are matched (incorrectly), as their representations are dominated by the edge.

4. *Effect of window size* : (See Fig. 103a) The algorithm also depends on the choice of window size. A small window size leads to more matching ambiguities since the patches are less informative. This shows up as noise in the result. Large window sizes result in smoother but less detailed results and experience *border bleeding*. Border bleeding refers to when the foreground bleeds into the background. This is due to the ambiguity of patches at the border which are dominated by the sharp gradient of the border.

A *left-right cycle consistency test* can detect half occlusions and outliers (other anomolous results). This requires us to compute the disparity maps for both images. Then for a pixel on the left image $\mathbf{x}_L = (x_L, y)$ with a disparity $d_L(x_L, y)$, we expect that a correct matching satisfies $d_R(x_L + d_L(x_L, y_L), y) = -d_L(x_L, y_L)$. Violations can then be marked.

9.1.4 Disparity to depth

Fig. 103b illustrates the situation of two rectified views. Let $x_1 > 0$ and $x_2 < 0$. Then we note that the disparity is $d = x_1 - x_2$ because the sum the horizontal distances from the focal points (which defines a common origin for both images) must be equal to the horizontal displacement. Let z be the depth of the point, f be the focal length of the cameras (which can be corrected to be the same during rectification) and b be the horizontal

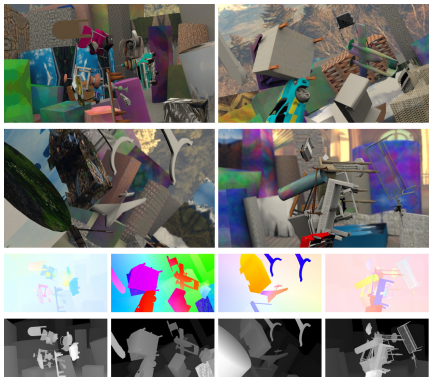


Figure 104: On the left are some samples from the Monkaa dataset, and on the right is the Flying Things dataset.

translation. Then by the similarity of the inner and outer triangles we get,

$$\begin{aligned} \frac{z-f}{b-d} &= \frac{z}{b} \\ zb - fb &= zb - zd \\ z &= \frac{fb}{d} \end{aligned}$$

Thus we can compute the depth map from the disparity. The error in depth grows quadratically with depth since $\frac{dz}{dd} = -\frac{fb}{d^2} = \frac{z^2}{fb} \Rightarrow \Delta z = \frac{z^2}{fb}\Delta d$ i.e. for unit error in disparity there is quadratically more error at points which are far.

9.1.5 Spatial Regularization and Deep Learning

Today there are much more sophisticated methods than block matching. One example is *spatial regularization* [14]. The idea behind spatial regularization is that instead of using a large window, we can use a small window and integrate the smoothness requirements by specifying an energy functional on the disparity map and solving for $\mathbf{D}$ jointly. The energy functional is of the form

$$E(\mathbf{D}) = \sum_i \psi_{data}(d_i) + \lambda \sum_{i \sim j} \psi_{smooth}(d_i, d_j) \tag{128}$$

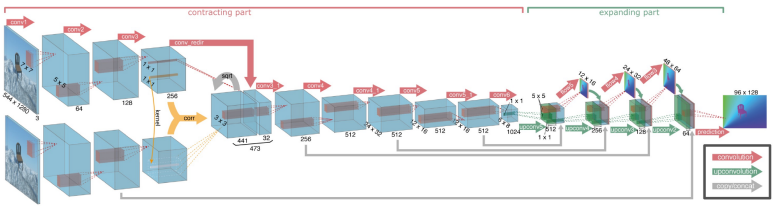
The smoothness constraint is captured by the binary term that penalizes deviations in disparity between neighboring pixels. $i \sim j$ denotes neighboring pixels on a 4-connected grid. Possible choices for the penalty function include,

- Potts : $\psi_{smooth}(d, d') = [d \neq d']$, where [] denotes the Iverson bracket that evaluates to 1 when the condition is true, else 0.
- Truncated l_1: $\psi_{smooth}(d, d') = \min(|d - d'|, \tau)$

The unary term $\psi_{data}(d_i)$ captures the block matching cost. To optimize (128) we can use various optimization techniques in the literature such as belief propagation or graph cuts (See CV Lecture 5.4).

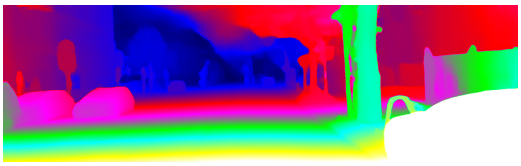
Another alternative to using smoothness constraints is to directly learn the stereo matching problem from data. However, in contrast to the simpler methods presented earlier we now require huge amounts of data. Since more data is available today, these methods now dominate the benchmarks. One of the first methods to use deep learning for end-to-end stereo reconstruction was the *DispNet* network [**?**]. DispNet uses a *U-Net* [**?**] like architecture (see Fig. 105a) with an encoder and a decoder and skip-connections to retain details. It also incorporates some ideas from the classical stereo matching literature like the correlation layer which does something similar to the cross correlation we encountered earlier. It also uses a multi-scale loss that penalizes disparity error in pixels and curriculum learning (examples are presented easy-to-hard).

In order to train the model [**?**] pretrained it on large synthetically generated datasets (See Fig. 104) where the ground truth could be directly computed, due to the lack of annotated data available at the time. This was followed by fine-tuning on a real dataset. Even though the synthetic datasets were not realistic, the pre-training was very effective. Fig. 105c shows the result of this model on a road scene. The overall quality was pretty good at the time, although it suffered from problems such as errors in the sky region and border bleeding.

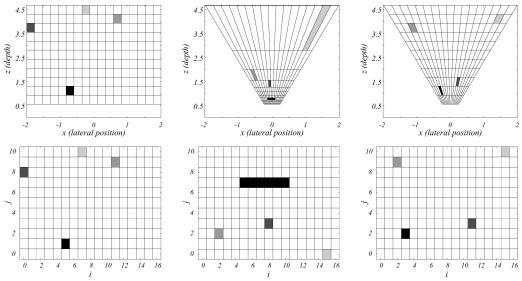


(a) DispNet uses a convolutional encoder -decoder network with a correlation layer and skip connections.

(b) An image from the KITTI dataset.

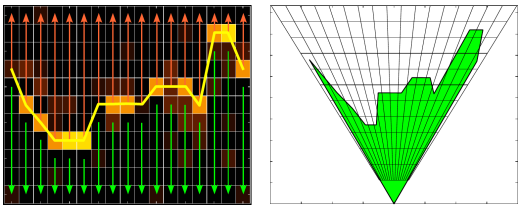


(c) Disparity map estimated by DispNet.



(a) Cartesian (b) Column/Disparity (c) Polar

(a) From left to right are the *cartesian, column* and *polar* representations for the occupancy grid presented in [1]. Polar and column representations allow for higher resolution at nearby points, compared to a cartesian representation.



(b) The goal of free space estimation in BEV is to learn a boundary $z = f(x)$ on the $x - z$ plane to segment the free space and occupied regions. The left image shows a boundary learnt in cartesian representation, while the right image shows the same boundary in polar representation.

9.2 Freespace and Stixels

For freespace estimation *the input* is a depth map per frame obtained from stereo reconstruction or directly via LIDAR scanners. The goal is to convert this into a freespace estimate in a bird's eye view (BEV), i.e. an overhead view. The BEV freespace map is much more directly useful to a self-ddriving car and provides crucial information for (local) path-planning and collision avoidance.

We will look at the method presented in [1]. Their approach can be described as follows,

1. Integrate multiple disparity maps temporally by aligning them through visual odometry so that the estimate for the disparities is more robust and less noisy.

2. Convert depth measurements into BEV occupancy map.

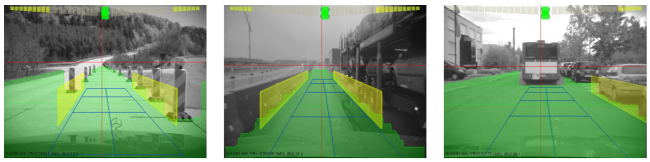
3. Optimize freespace segmentation via energy minimization.

The first choice we have to make is how to represent the occupancy. Fig. 106a illustrates 3 representations that differ in terms of computational requirements. The *cartesian* representation creates an *occupancy grid* made up of equally sized bins in the lateral (x) direction and along the depth (z). Our final goal is to estimate the likelihood of each cell to be occupied. The *column/disparity* representation discretizes the space into equal bins laterally (i.e x) and disparity values (d). Thus the bins are not equally sized wrt. the x, z coordinate system, and infact cells at higher depth are larger because disparity varies more slowly as depth increases. The *polar* representation is similar to the column/disparity representation except the bins are now equally sized wrt. z.

9.2.1 Depth-to-Occupancy Conversion (polar representation)

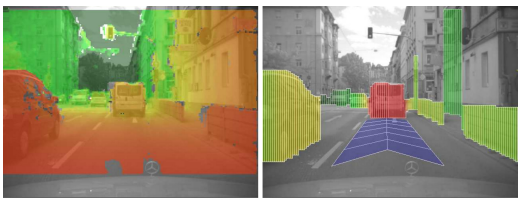
In this section we consider how to convert a depth map to an occupancy map in polar representation, although the same is possible for cartesian or column representations as well. Let (x_i, z_i) denote the x-coordinate and depth value ($z = fb/d$) of pixel $i \in \{1, \ldots, N\}$. We estimate the occupancy likelihood at location (x, z) in the occupancy map using,

$$O(x, z) = \sum_{i=1}^{N} \exp\left(-\gamma_x (x - x_i)^2 - \gamma_z (z - z_i)^2\right) \tag{129}$$



(a) Highway. (b) Freeway. (c) Downtown.

(a) Results of [1] on different road scenes. Note that although it segments the free space and non-free space correctly, it cannot detect the height of objects in view.



(a) Dense disparity image (SGM result) (b) Stixel representation

(b) [2] explored a "stixel representation" of the scene for freespace segmentation. Stixels allow for more granular representation than pixels but are more compact than pixels. Note that this algorithm successfully segments objects from the background concurrently with free space estimation.

This kernel estimator accumulates evidence of nearby observations. The exponential term decays as it gets farther away from (x, y). Thus, a nearby observation contributes a value close to 1 to the likelihood, while far away observations contribute 0. Thus if a cell has a lot of nearby observations, we decide that it is occupied. Larger values of γ_x result in faster decay i.e. a smaller receptive field.

[1] define a data term for each image column $j \in \{1, \dots, M\}$ based on (129),

$$\psi_z(z_j) = \frac{1}{O(x_j, z_j)} \tag{130}$$

Fig. 106b illustrates the desired output. We wish to segment the space into free space and non-free space or occluded area, using a boundary that passes through the areas with large number of (point) observations. Further we want the boundary to be smooth. We do this by minimizing the following objective

$$E(z_1, \dots, z_M) = \sum_j \psi_z(z_j) + \lambda \sum_j \psi_s(z_j, z_{j+1}) \quad \text{with} \quad \psi_s(z_j, z_{j+1}) = \min(|z_j - z_{j+1}|, \tau) \tag{131}$$

Thus for each column we wish to estimate the depth at which we have an object/surface occluding our view. The unary term is the data term corresponding to (129). The binary term constrains the boundary to be smooth by penalizing sharp changes in the boundary (which are large than τ). It is a truncated L_1 penalty. Because the graphical model is one-dimensional, we can find the optimal solution using *dynamic programming*.

Fig. 107a shows us the results of this algorithm on some road scenes. It should be noted that the height of the object is unknown, only the free space is estimated. This can be used for navigation and collision avoidance.

[2] extends this work using the concept of *stixels* which are pixels elongated in the vertical direction (like sticks) (see Fig. 107b). This still allows us to represent the environment compactly, but with more detail than a BEV grid. [2] formulates this as a segmentation problem where the goal is to segment the image into stixels from a dense disparity map and semantic segmentation map. They not only segment the image into free space and non-free space, but also segment the objects from the background at the top.

9.3 Optical Flow

Optical flow is the *apparent motion* of objects, surfaces, and edges in a visual scene caused by the *relative motion* between an observer and a scene. Knowing the past/current motion allows to make predictions about the future. For example, we might predict the location of a vehicle 1 second into the future, by extrapolating its motion from the past. However, these predictions are in image space i.e. concerns the movement of pixels, not 3D points (which is dealt with by scene flow). Scene flow however requires more observations.

Like stereo vision, optical flow also attempts to make inferences about the environment from 2 images. Stereo vision considers 2 images taken at the same time, while optical flow considers 2 images taken at 2 time steps. However, while stereo vision considers a stationary scene with only camera motion, optical flow deals with both camera motion and object motion (i.e. does not have assume a stationary scene). Thus while stereo vision boils down to a 1D correspondence search, optical flow is a full 2D problem since pixels are not constrained in any way (due to object motion).

The motion field is a 2D velocity field $(a(x, y), b(x, y)) : \Omega \to \mathbb{R}^2$ representing the projection of the 3D motion of points in the scene onto the image plane. Optical flow $(u(x, y), v(x, y)) : \Omega \to \mathbb{R}^2$ however is a 2D velocity field that describes the rate of displacement of *pixels*. Optical flow is a *proxy* for 2D motion, but is not the same! This arises from the fact that the pixel correspondences between two images is determined by patch similarity and not the ground truth 3D coordinate. Consider a Lambertian (non-reflective) ball rotating on its axis under a stationary light source. A stationary camera sees a fixed image because the surface of the ball

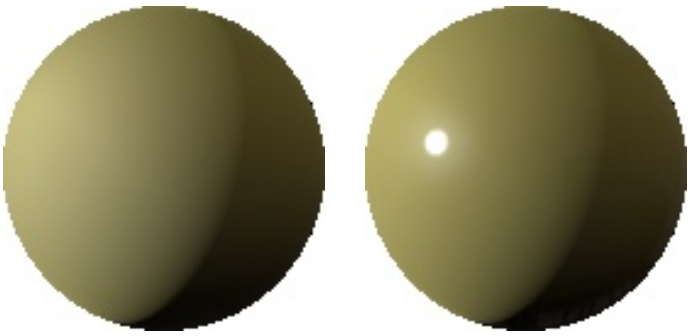


Figure 108: The left image protrays a Lambertian ball rotating on its axis under a stationary light. Note that this image will remain the same because the ball is textureless. Hence the optical flow is 0, despite there being motion. The right image portrays a specular ball under a rotating light source. Although the motion field is 0, the reflection of the light source causes a non zero optical flow.

Figure 109: Consider the two images shown above. If the left image is considered as the reference, than the optical flow observed is diagonal (north-east). However this does not allow us to infer the actual motion of the object, since we cannot see its ends. In fact motion to the right, or motion to the left both result in the same change in the image and hence the same optical flow. This is because we cannot detect the motion along the line since each pixel is identical. This is the *aperture problem.*

has no defining characteristics. Thus the optical flow is a zero field, although the motion field is non zero. The opposite scenario is that of a stationary specular (reflective) ball and a rotating light source. Although the ball is stationary, the reflection of the light on the ball has motion resulting in a non-zero optical field. However the motion field is zero! Fig. 108 illustrates these two scenarios. Fortunately scenes typically have a lot of texture and thus optical flow is a good proxy for the motion field in many situations.

The optical flow fields tell us something (maybe ambiguous) about:

- The **3D structure** of the world
- The **motion of objects** in the viewing area
- The **motion of the observer** (if any)

In contrast to stereo we cannot exploit epipolar geometry and thus have to deal with a 2D estimation problem. This is illustrated by the aperture problem. The aperture problem refers to the fact that the motion of a one-dimensional spatial structure, such as a bar or edge, cannot be determined unambiguously if it is viewed through a small aperture such that the ends of the stimulus are not visible. *It illustrates the ambiguity in estimating motion from optical flow.* Fig. 109 illlustrates this.

9.3.1 Horn-Schunck algorithm

In this section we present the [**?**] algorithm to estimate optical flow. Consider the image I as a function of continuous variables x, y, t. Consider $u(x, y)$ and $v(x, y)$ as continuous flow fields (i.e. functions from Ω to $\mathbb{R}^2$). The Horn Schunck algorithm minimizes the following *energy functional*,

$$\begin{aligned} E(u,v) &= \iint \underbrace{\left(I(x+u(x,y),y+v(x,y),t+1)-I(x,y,t)\right)^2}_{\text{quadratic penalty for brightness change}} \\ &+ \lambda \cdot \underbrace{\left(\|\nabla u(x,y)\|^2+\|\nabla v(x,y)\|^2\right)}_{\text{quadratic penalty for flow change}} dx\,dy \end{aligned} \tag{132}$$

Here λ is the regularization parameter and the gradient $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right)$. The first term squared difference between the original pixel (x, y) at time t and the predicted pixel location $(x+u(x,y), y+v(x,y))$ at time $t+1$. This is called the *brightness constancy assumption*. The second term penalizes sharp changes in the flow field, i.e. it prefers a smooth optical flow field. In other words the first term is the data term and the second term is the regularizer. Matching pixels between the two images is *ill posed*, since there might be no match or many matches. Strong regularization resolves this ill-posedness to find a unique solution.

The objective is highly *non-convex* due to the image function and has many local optima. Consequently, minimizing (132) directly is a hard problem. We solve this by *linearizing* the brightness constancy assumption using the first order multivariable Taylor approximation,

$$f(x,y) \overset{a,b}{\approx} f(a,b) + \frac{\partial f(a,b)}{\partial x}(x-a) + \frac{\partial f(a,b)}{\partial y}(y-b) \tag{133}$$

This approximation basically ignores the higher order terms in the series which is a reasonable thing to do when the higher order derivatives are small i.e. the function is smooth. Applying this to I we have,

$$\begin{aligned} I(x+u(x,y), y+v(x,y), t+1) \overset{x,y,t}{\approx} & I(x,y,t) + I_x(x,y,t)(x+u(x,y)-x) \\ & + I_y(x,y,t)(y+v(x,y)-y) + I_t(x,y,t)(t+1-t) \\ = & I(x,y,t) + I_x(x,y,t)\,u(x,y) + I_y(x,y,t)\,v(x,y) + I_t(x,y,t) \end{aligned} \tag{134}$$

Substituting (134) in (132) we obtain the following *linearized objective*,

$$\begin{aligned} E(u,v) = & \iint \left(I_x(x,y,t)u(x,y) + I_y(x,y,t)v(x,y) + I_t(x,y,t)\right)^2 \\ & + \lambda \cdot \left(\|\nabla u(x,y)\|^2 + \|\nabla v(x,y)\|^2\right) dx\, dy \end{aligned} \tag{135}$$

Finally, we replace the double integral with a sum which corresponds to *spatial discretization* leading to the discretized objective,

$$\begin{aligned} E(\mathbf{U},\mathbf{V}) = & \sum_{x,y} \left(I_x(x,y)\,u_{x,y} + I_y(x,y)\,v_{x,y} + I_t(x,y)\right)^2 \\ & + \lambda \cdot \left((u_{x,y} - u_{x+1,y})^2 + (u_{x,y} - u_{x,y+1})^2 + (v_{x,y} - v_{x+1,y})^2 + (v_{x,y} - v_{x,y+1})^2\right) \end{aligned} \tag{136}$$

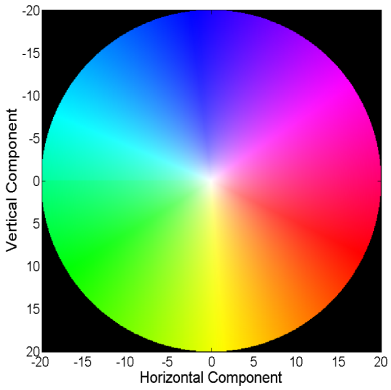
While we originally minimized over functions $u(x,y), v(x,y)$, we now minimize over fixed size flow maps $\mathbf{U}, \mathbf{V}$. Further, the objective is quadratic in $\mathbf{U}, \mathbf{V}$ and thus has a unique optimum. The image derivatives I_x, I_y can be obtained by applying a Sobel filter or other methods.

To solve the optimization problem we differentiate (136) wrt. $\mathbf{U}, \mathbf{V}$ and set the gradient to 0. This results in a huge but *sparse linear system*. This can be solved using *standard techniques* (e.g., Gauss-Seidel, Successive over relaxation).

However, linearization works only for *small motions* and so the algorithm performs poorly when this assumption is violated. In practice, this is mitigated by *iterative linearization*. To do this we first linearize the images anf compute an estimate for the optical flow. We apply this estimate to warp the first image, and then relinearize and solve (136). We repeat this till the first image is warped into the second, or the estimated flow field is very small. The final flow field is the composition of all the intermediate flow fields. The other strategy utilized is *coarse-to-fine estimation*, which is similar. In this method the images are downsampled to lower resolutions to form an image pyramid. On an image pair that is say $\frac{1}{16}$ the original resolution, the optical flow is much smaller than the original and the linearity approximation is better motivated. Using the estimated flow field we warp the first image at the next resolution and optimize (136) again to solve an optical flow field that is not as large anymore. Although these tricks work to some extent they also have their own problems, for example coarse-to-fine estimation loses thin structures early on which cannot be recovered later.

An optical flow field (estimate or ground truth) provides two values per pixel: flow in x- and y- direction. How can we visualize 2 values per pixel? A dense arrow grid is inconvenient. Insteed we encode the flow direction as color hue and the flow magnitude as flow intensity. Fig. 110c shows us an example of an optical flow map encoded using the accompanying color chart (Fig. 110a).

Fig. 110c shows the results for an optical flow field estimated using the Horn-Schunk algorithm. Although the results are good it suffers from over smoothness and blurred boundaries. This is a consequence of the quadratic penalties applied which strongly penalize outliers and sharp changes. Follow up works such as [25] introduce various heuristics such as replacing the quadratic penalty to improve the results and make them more robust (See Fig. 110d). Fig. 111b shows the results for Horn-Schunck algorithm with robust penalties on the KITTI dataset. The results are not very good, illustrating the difficulty of optical flow for self-driving cars due to challenges such as large motion vectors and textureless surfaces.



(a) Color chart for encoding optical flow. Hue indicates the direction and intensity indicates the magnitude.

(b) Input image

(c) Optical flow estimated by the Horn Schunck algorithm. We see that it suffers from smoothness artifacts especially at the boundaries.

(d) The results of the Horn Schunck algorithm with robust penalties are sharper (including at the boundaries) than Fig. 110c

9.3.2 Deep Learning for Optical Flow

While originally deep learning methods were not on par with classical methods, today they are the state of the art on the KITTI dataset. Flownet [**?**] is one of the earliest examples of deep learning for optical flow. The architecture of the network is very similar to Dispnet [**?**] (See Fig. 105a). It has an encoder with strided convolutions and a decoder with upconvolutions. Skip-connections and a multi-scale loss (EPE in pixels) are incorporated to better propagate gradients through the network. Curriculum learning is also applied, and the network is trained on large amounts of synthetic data and then fine-tuned on real data.

A more recent followup to Flownet is FlowNet2 [15] uses classical ideas in a deep learning framework. It essentially stacks multiple FlowNets and uses the intermediate estimates to warp the images to form intermediate inputs (similar to iterative linearization and course-to-fine estimation, but with an end-to-end model). Fig. 111d illustrates the architecture and Fig. 111c shows its results on KITTI.

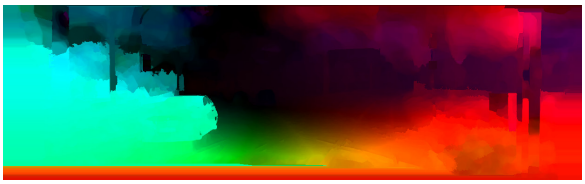
9.4 Scene Flow

In practice, *2D motion* estimation (in the image domain) is often not sufficient. We live in 3D space, hence we must estimate/predict *motion in 3D* which is the problem of scene flow estimation. For example, for a self driving car we must be able to estimate the 3D motion of other traffic participants reliably.

[27] defines scene flow as "Scene flow is a dense three-dimensional vector field defined for each point on every surface in the scene". Thus for every visible 3D point we wish to measure both its 3D location and 3D velocity. The scene flow is the *instantaneous 3D motion* of every point $(x, y, z)^\top$ on every surface in the scene. Optical flow is the 2D projection of the scene flow into an image. [27] compute 3D scene flow from 2D optical flow estimates. Later works estimate scene flow directly from images using the brightness constancy assumption across multiple images, similar to the Horn-Schunck algorithm.

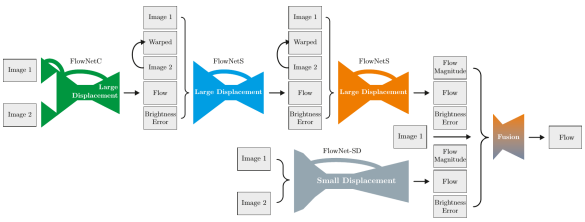
How many observations are required for estimating scene flow? While optical flow estimation requires 2 consecutive images, scene flow images require at least 2 pairs of stereo images or 2 images with *depth*. In a self-driving car we typically have a stereo camera, and thus to compute scene flow we have to compute correspondence across all 4 images. The pairs gives us a 3D location at time t and $t + 1$, from which the

(a) An image from the KITTI dataset.



(b) Optical flow estimated by Horn-Schunck algorithm with robust penalties. Clearly it fails to generalize to traffic scenes.

(c) Optical flow estimated by FlowNet2. The results are much better than the classical method.



(d) The architecture of FlowNet2 is composed of multiple FlowNets.

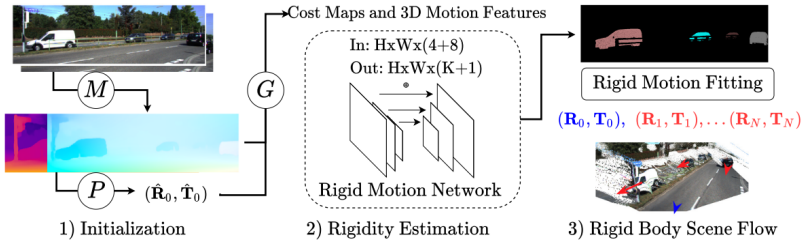


Figure 112: This schematic shows the pipeline used by [**?**] (Picture credits : [**?**]). This is explained in more detail in Section 9.4

actual 3D motion can be obtained. Hence, scene flow effectively combines the problems of stereo and optical flow. A common representation that is used to estimate the scene flow in images is the pixel location in the reference (say left) image (2 parameters), the optical flow between two corresponding images at time $t, t+1$ (2 parameters), and the disparities in both frames ($1+1=2$). In theory, this completely describes the scene flow since it has just 6 parameters (2×3 coordinates).

[**?**] currently ranks 1st on KITTI Scene Flow Benchmark. It exploits a recurring idea in computer vision that the scene is composed of *rigidly moving objects*. By incorporating this assumption into our algorithm we can achieve more robust estimation. The pipeline used by [**?**] can be described as follows (See Fig. 112),

1. First depth and optical flow are computed using off-the-shelf networks.
2. The camera motion is estimated using epipolar geometry for each frame.
3. The results of the previous steps are used to compute rigidity cost maps and rectified scene flow.
4. This is then fed into a two stream network that segments the image into a rigid background and an arbitrary number of rigidly moving objects.
5. Finally rigid transformations (i.e. translation and rotation) are fitted for the background and each rigid instance to update their depth and 3D scene flow.

10 Object detection

In this section, we focus on object detection which is about locating objects in 2D or 3D, and recognizing the object category. This section is structured into 5 units: Introduction, Performance evaluation, Sliding window object, Region Based CNNs and 3D Object Detection.

10.1 Introduction

We can distinguish recognition problems, as shown in Fig. 113, into four different problem categories:

- **Image Classification** - the goal is to assign a single class label or image category to an image, for example, the "street scene" label.
- **Semantic Segmentation** - the goal is to assign a semantic label to every pixel in the image for both objects and background, for example, we want to determine the pixels that comprise the road area.
- **Object Detection** - the goal is to localize in terms of a bounding box and classify all objects in an image.
- **Instance Segmentation** - the goal is to assign a semantic and an instance label to every pixel of an object in the image.

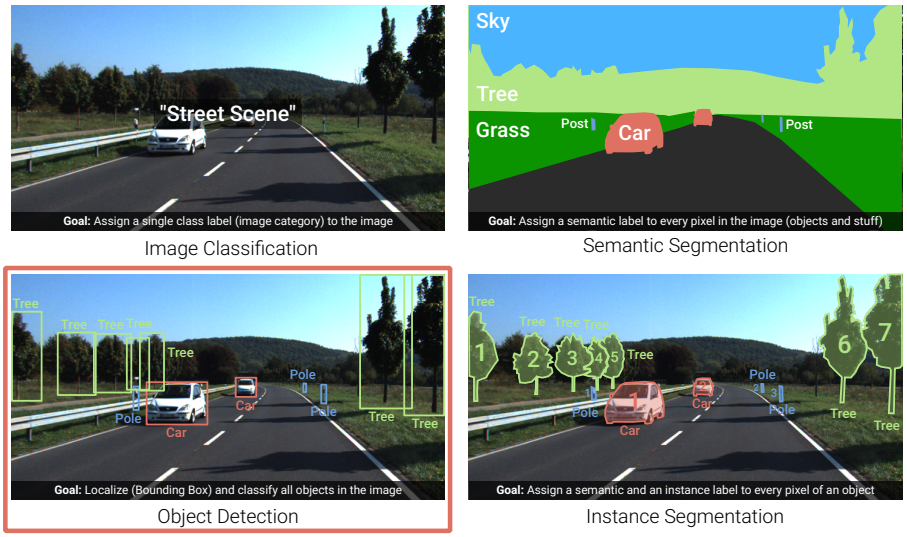


Figure 113: **Different recognition problems**

This lecture mainly focuses on the object detection problem, a relevant problem for self-driving. To motivate object detection problems, you can have a look at the following videos:

- Mobileye (https://www.youtube.com/watch?v=HXpiyLUEOOY) - the video from Mobileye that illustrates how a company produces smart cameras for drivers assistance systems utilizes object detectors in their products. (System continuously scans the area in front of a vehicle, detecting vehicles, etc.)
- Chris Urmson's TED talk (https://www.youtube.com/watch?v=tiwVMrTLUWg) - the video about how the car "sees" the world and what challenges self-driving cars should solve. (Bounding boxes assigning, construction detection, police signals detection, etc.)

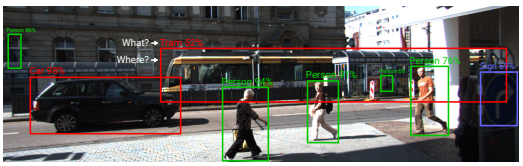
10.1.1 Problem setting

We regularly have an RGB image or a laser range scan for an input, like in Fig. 114a, and what we want to infer is a set of 2D or 3D bounding boxes with category labels and a confidence value. For example, in Fig. 114b we have detected a "person" with that bounding box and the confidence of 94%. Almost all object detectors give confidence because the benchmarks and the downstream applications usually need to know with which confidence something is detected to make a decision or evaluate the algorithm.

In Fig. 114c there is an example for the corresponding 3D scenario where bounding boxes are detected in 3D LiDAR point clouds.

Basically, our goal is to figure out what is wherein an image at a coarse bounding box level. This can be a problem because if we consider the continuous coordinates there are infinitely many possible bounding boxes, and after the quantization to pixels, there are still enormous bounding boxes, for the number of entities that is unknown a priori.

(a) Input image



(b) 2D detection in RGB image

(c) 3D detection in LiDAR point clouds

Figure 114: **Input and detection examples**: (a) 2D RGB image input image example; (b) Detected bounding boxes, with classes and confidence level for 2D RGB image input; (c) 3D LiDAR point clouds input example and corresponding camera view scene;

10.1.2 Challenges

There are some more challenges of the recognition problem:

- **Large Number of Image Categories**. In the ImageNet paper [**?**], authors estimate about 10000 to 30000 categories. But luckily in the self-driving setting, there is only a few object categories that are occurring really frequently, such as pedestrians, cyclists, motorcyclists, and vehicles.
- **Intra-class Variation**. In practice, we can have same-class images which look completely different at the pixel levels, which results into different input signals to a neural network that has to classify all as same-class objects.
- **Viewpoint Variation**. An object image can be taken from different perspectives, which again results in different input signals to a neural network despite all of them show the same object at about the same size.
- **Illumination Changes**. The pixel intensities change dramatically depending on where the light source is located, while the scene content stays the same.
- **Background Clutter**. Sometimes, it is hard to recognize objects if they are not distinct and not salient enough in the image.
- **Deformation**. Objects deformation in particular animals or humans can deform the body and depending on the deformation they could look quite different.
- **Occlusion**. Occlusion can happen for the cars, for example where you have multiple cars parking in a row, and you see only the back of each car, so they are heavily occluding each other.

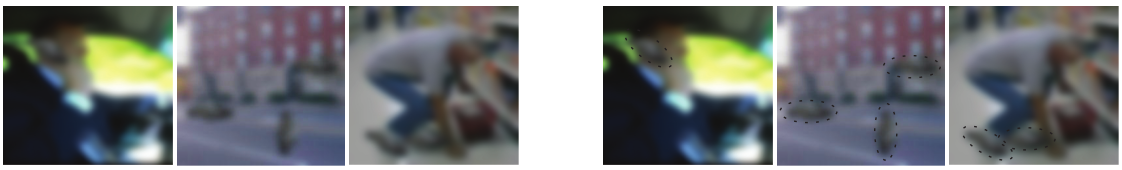
10.1.3 Context Matters

Content context matters a lot for object recognition, but using too much context which is required to recognize objects at high accuracy can at the same time be also misleading.

In Fig. 115a you can see heavily downsampled images, but still, we as humans can recognize something here. But in all images, there are exactly the same blobs inserted at different locations (Fig. 115b), illustrating how much we humans also rely on contextual information to recognize objects.

10.2 Performance Evaluation

How can we evaluate object detection performance? Let's first consider the case of a single object in the scene such as the stop sign in Fig. 116a.



(a) No same parts are visible

(b) Same blobs outlined

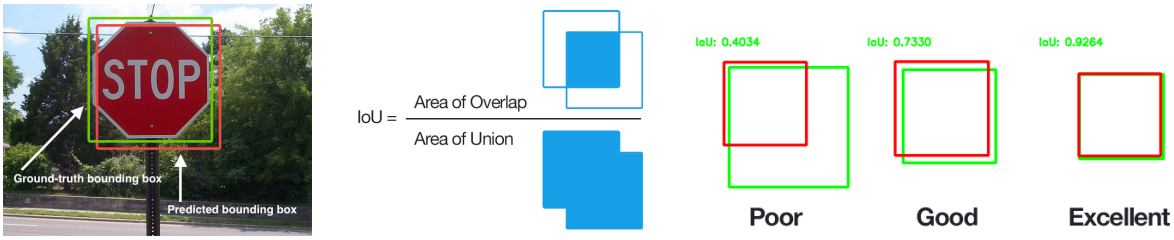
Figure 115: **Contextual information importance**: (a) Triplet of images with no obvious same parts; (b) Same blob outlined in every image;

10.2.1 Intersection-over-Union

The Intersection-over-Union (IoU) is a detection metric that measures the ratio of the area of overlap of ground truth and predicted boxes with respect to the area of the union of these two boxes, which is graphically illustrated in Fig. 116b.

The highest possible IoU = 1 when the predicted box and the ground truth box are identical, and the lowest IoU = 0, where these boxes are disjoint.

In Fig. 116c we can see examples: a poor fit, a good fit, and an excellent fit. We can consider a particular threshold for IoU which will be considered as a successful detection. For example, in Fig. 116c if we consider an IoU threshold of 0.5 which is commonly used for 2D object detection evaluation, then the only right example will be considered as correct.



(a) Detection in scene example

(b) Intersection-over-Union equation

(c) Prediction examples

Figure 116: **The Intersection-over-Union (IoU) explanation**: (a) Detection in scene example; (b) IoU equation; (c) Bounding boxes fit examples;

10.2.2 Average Precision Metric

How can we fairly measure detection performance in the case of multiple objects?

We can do this by setting the confidence score per box, which is still allowed to be counted as detections and varying the number of bounding boxes that are an output of the detector for a particular image. Because the number of detections depends on the threshold (Section 10.2.1) we should consider a whole range of thresholds because none of them will be optimal.

Average precision metric is used for evaluating how the multi-object detection algorithm is doing. Let's derive how this metric is working in several steps:

1. **Run detector with varying thresholds** - run a search through a whole range of the thresholds starting with thresholds where none of the objects are returned to a range where a lot of boxes are returned.
2. **Assign detections to the closest object** - assign to the closest ground truth bounding box respectively, that each ground truth object is maximally associated with one detection. For this process, typically bipartite graph matching techniques like the Hungarian algorithm are used.
3. **Count TP, FP, FN** - count the number of true positives, false positives, and false negatives.
   - **True Positives (TP)**: Number of objects correctly detected (IoU $>$ 0.5)
   - **False Negatives (FN)**: Number of objects not detected (IoU $<$ 0.5)
   - **False Positives (FP)**: Wrong detections

   The false positives are the remaining detections that are not assigned to any ground truth objects because either they are redundant or there is no object in the area.

4. **Average Precision (AP)** - compute the precision P which is defined below, and compute the recall R. High precision means that the objects that are detected by the object detection algorithm are actually correct, and the higher recall means that most of the objects that are in the ground truth are actually detected.

$$\text{Precision } P = \frac{TP}{TP+FP}, \quad \text{Recall } R = \frac{TP}{TP+FN}, \quad \text{Avg. Prec. } AP = \frac{1}{11} \sum_{R \in \{0,\dots,1\}} \max_{R' \geq R} P(R')$$

In practice, there is a trade-off between precision and recall: either the object detector returns too many boxes, so it will create false positives (high recall, low precision). Or the object detector can only return the most confident boxes (high precision, low recall).

Computing precision and recall for every possible threshold we can plot a precision over recall, which is an orange curve in Fig. 117. Typically, this curve is "zig-zaggy", so we smooth it by max to the right (green curve).

Typically, it's hard to have that curve reach point $(1, 1)$ because that would mean you would have a perfect detector that for the particular detection threshold returns exactly the objects that are present in the image at the correct location with the correct semantic class.

In order to compute the average precision metric, we take the average of $\max_{R' \geq R} P(R')$ values. We can think of this as computing the integral under that green curve in Fig. 117. In this case, it has $1/11$ in definition, because we have 11 points on this plot. This gives us an AP metric that is independent of a specific detection threshold choice.

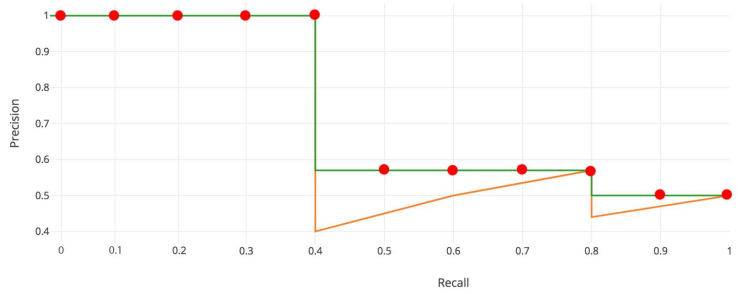


Figure 117: **Precision over recall plot**: is used to calculated average precision metric

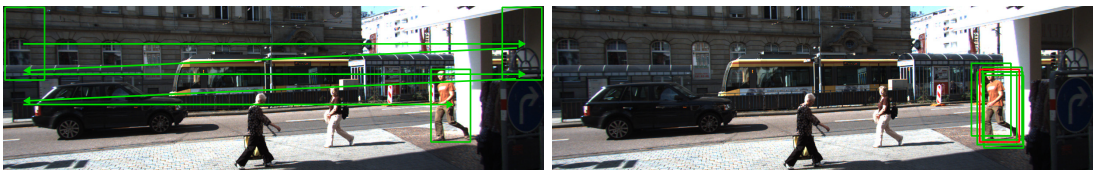
10.3 Sliding Window Object Detection

In this unit, we consider the classic sliding window object detection methods before moving to deep learning-based methods in unit Section 10.4.

10.3.1 Sliding window

Sliding window object detection works as follows:

1. Run **sliding window** of fixed size over the image (green rectangular in Fig. 118a) and extract features for each window.
2. **Classify each crop**, using a binary classification "object vs. background", using a standard classification model based on extracted features from that window (e.g. SVM, random forest, etc.)
3. **Search across aspect ratios/scales** of the box, to recover objects of varying size/distance (exhaustive search problem).
4. **Non-maxima suppression** (=clustering) to retain only one prediction per object. If we slide this window through all possible locations there will be some windows in the vicinity of the target object (green boxes near red one in Fig. 118b), which will also respond highly in terms of the classification score. Because we want to keep only the best predictions and only once for each object in the scene, we apply non-maximum suppression. It's a clustering heuristic that looks at the large overlap of bounding boxes and keeps only the one that has the highest confidence.



(a) Sliding the window over the image (b) Multiple detections in the vicinity of the object

Figure 118: **Sliding window**: (a) We apply a sliding window to every part of the image in order to extract features; (b) Then we apply non-maxima suppression to retain only one prediction per object;

10.3.2 Histograms of Oriented Gradients

On which feature space should the predictive model classify these crops? The simplest is to use the RGB pixel space, but we have seen problems with this in Section 10.1.2.

A better choice is to use a **histogram of oriented gradients (HoG)** [**?**], which was the de facto standard for more than 10 years of research in object detection. The idea of the method is to represent patches with histograms which are representing gradient angles weighted by the gradient magnitude.

Let's illustrate this with the example in Fig. 119. We first differentiate the image spatially in the x and y direction, and then we compute the gradient magnitude and the gradient angle, in Fig. 119 we have pinned that gradient angle into only eight different orientations. The next step what we do is to subdivide that image into cells or subregions, after that for each cell we compute a histogram where we sum up the weights for each of the angles that are contributing to this histogram. In the end, we concatenate all of these histograms and use it as the feature vector of the image.

The advantage of this compared to using the RGB pixel space is that this histogram of oriented gradient features is invariant to small deformations (translation, rotations, etc.), and it will not alter the gradients too much because they are summary statistics of an entire region and angles are binned into this very coarse discretized representation. Another advantage is becoming independent of the absolute brightness because these histograms are not based on the original intensity information.

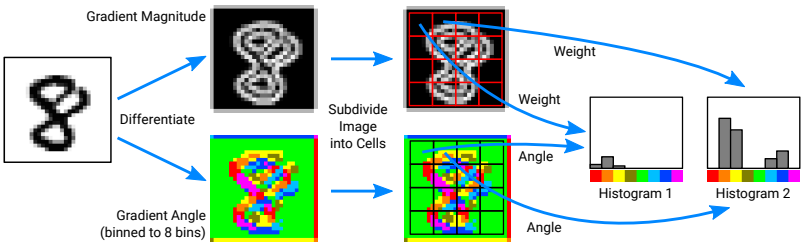


Figure 119: **Histograms of Oriented Gradients**: pipeline of creating final histogram-based representation

10.3.3 Part Based Models

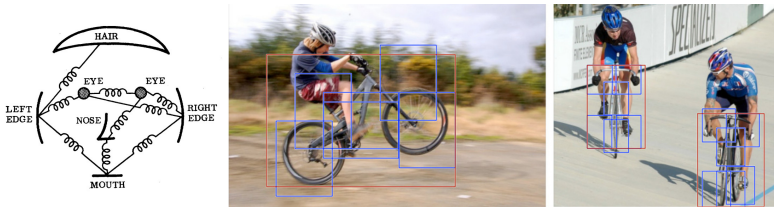
An extension of the HoG idea that became **part-based models** [**?**], where the idea is to model an object based on its parts and then model the distribution of part configurations. This idea goes back to the 1970s (left part of Fig. 120a), but in our case, it is used in combination with a graphical model and HoGs for each of these parts.

Part-based models are more invariant because we can allow some non-rigid deformations of the object by adjusting the relationship of the different parts (right part of Fig. 120a) and modeling this in terms of the graphical model distribution. However, these models lead to slower inference and don't lead to a huge gain, compared to training a HoG model for the entire object but doing that separately for each different view of the object that we might perceive.

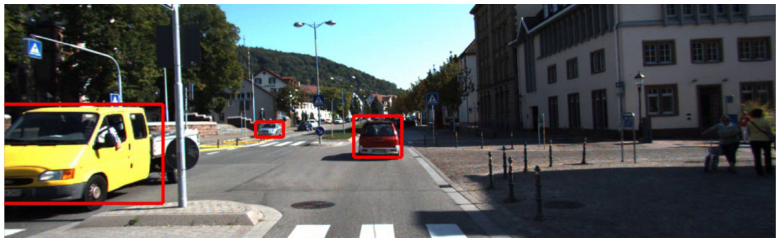
However, cars are relatively easy to detect, so these algorithms already led to impressive performance. In Fig. 120b is non-deep learning-based inference, based on these graphical models with simple HoG.

10.4 Region-Based CNNs

In this unit we focus on **two-stage** or so-called **region-based convolutional neural networks**, that are the de-facto standard nowadays for object detection.



(a) Concept of part configurations



(b) Results on the KITTY dataset

Figure 120: **Part Based Models**: (a) Illustration of part configurations concept; (b) Inference based on graphical models with simple HoG;

10.4.1 Hand-crafted Representations vs. Learned Representations

Previous approaches haven't led to major breakthroughs in object recognition, only for specific non-safety critical tasks. These are the reasons why is that: **1)** Computer vision features are actually very difficult to hand-engineer; **2)** It is unclear for humans how a good representation should look like; **3)** Sliding window approaches are typically quite slow in practice because they have to search through this vast space of all possible bounding boxes; **4)** Inference in complicated part based graphical models makes these methods even slower; **5)** Computation is often not efficiently reduced

All of these approaches suffer from the problem that the intermediate processing layers have been hand-engineered without any focus on optimality. The difference in deep learning of course is that we try to learn all those features, and we learn all of these low-level features these are the early layers in our deep neural network, and the mid-level features, and the high-level features which are the later layers, and even the classifier it's all one big neural network that is trained end-to-end.

This has led to dramatic improvements in performance. With the first deep learning-based models such as Fast R-CNN using a very simple backbone like AlexNet we already got a three times improvement ($5 \rightarrow 15$) in AP metric compared to deformable part-based model benchmarks. Then within a short period of development of deep neural network-based detection methods we got increased to 46 AP metric.

10.4.2 Object Detection with Deep Neural Networks

How can we detect objects using deep neural networks? The idea on the **first stage** is **detect** a smaller set of candidate bounding boxes that might contain an object, where some of them are correct, so we are effectively quantizing the space. Here we should choose the set size of candidate bounding boxes (e.g. 2000) to be over complete, to be sure that they contain all the ground truth boxes. On the **second stage**, we take all of these boxes and **classify** them into a set of labels, and also **refine** the location of the boxes using a deep neural network.

So these are the two stages and therefore this approach is called a proposal-based or **two-stage object detector**. Note that we have split the original object detection problem now into two proxy tasks the task of detecting a smaller set of candidate bounding boxes (or the task of quantization of bounding box space) and the second task of classifying and refining these set of candidate bounding boxes.

This is illustrated [4] in Fig. 121: by approximation with a finite set of bounding boxes we might get an up a **quantization error** with respect to the true box, and now we refine the location of the boxes to the final prediction in dashed green by **regressing the location offset**. And also we remove redundant predictions using non-maximum suppression because there can be multiple proposal boxes that regress to the same object.

[4] Illustraions credits: Ross Girschick [**?**]

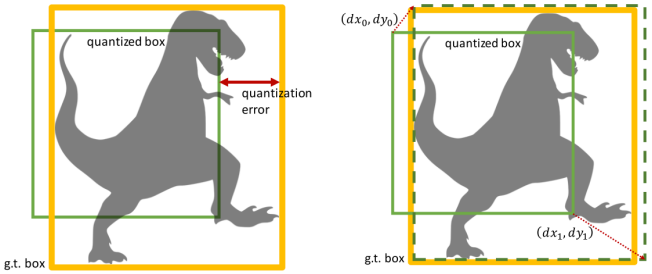


Figure 121: **Two-stage example**: detecting a candidate box, classifying object and refining the location offset

10.4.3 R-CNN: Region-based Convolutional Neural Network

The idea of R-CNN [**?**] is to compute proposals, these regions, and then for each of these regions we apply a convolutional neural network classifier. In Fig. 122a you can see the main steps of the algorithm:

1. Use an off-the-shelf region detector proposal algorithm, for example, selective search, edge boxes, MCG, etc., which are don't take into account the category of the object.
2. Crop and warp the original image into a fixed representation.
3. Forward propagates this fixed size representation through a standard convolutional neural network to get a feature representation.
4. Classify with the first output head that region (e.g. horse rider, pedestrian)
5. Regress the offset of the bounding box corners with the second output head, to refine them to fit the ground truth better than the original proposals.

We can see in Fig. 122a, the yellow region which is indicating the components that require only a per image computation. The area in green indicates the components that require per region computation, so per region, we need to crop and warp and run a ConvNet and this is the reason why it is slow.

In Fig. 122b you can see the generalization of R-CNN into the framework, where there are two steps in the per-image computation part: $f_I = f(I)$, which is a transformation of the input image into featured representation form, and then we have $r(I)$, which is a mechanism for region proposal detection. Then, in per-region computation part we have $g_i = g(f_I, r_i)$ which is computing featured representation for every proposed region. Then we apply additional processing $h(g_i)$ to each proposed feature, and we have multiple heads to make task-specific predictions. How the R-CNN blended into this framework is illustrated in Fig. 122c.

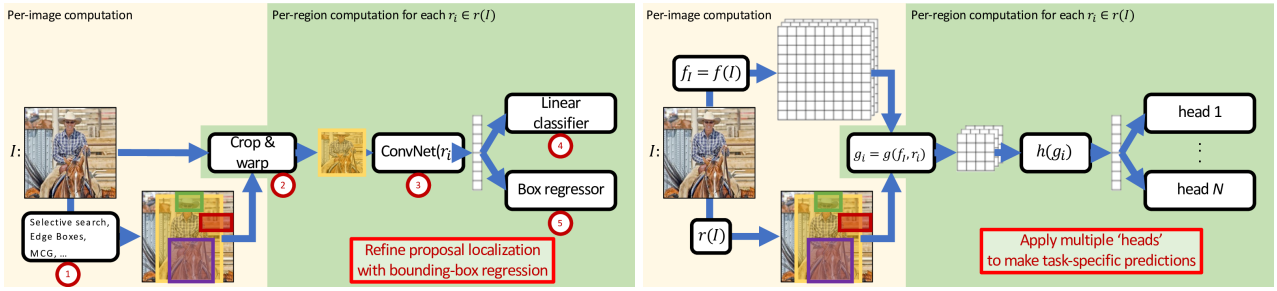
Some R-CNN problems: **1)** Heavy per-region computation (e.g. 2000 full network evaluations); **2)** No computation sharing or feature sharing; **3)** Slow region proposal methods; **4)** Generic region proposal techniques have limited recall.

10.4.4 Fast R-CNN

Fast R-CNN - fast region convolutional neural network [**?**] which has a lighter weight per-region computation and it's illustrated in Fig. 123a. These are the difference with the R-CNN model:

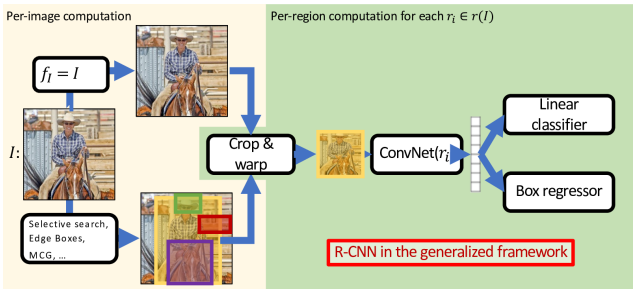
1. **FCN**(I) - we don't copy the image as a feature representation, but we run the fully convolutional network **backbone** in order to produce features from that image, but we do this only once per image. As backbone can be used any standard convolutional network (e.g. AlexNet, ResNet, Inception...). However, from the backbone, the global pooling layer is removed, in order to have output spatial dims proportional to input spatial dims.
2. **RoIPool** - region of interest pooling converts each region into a fixed dimensional representation [**?**]. RoI is illustrated in Fig. 123b, so the idea is to snap the proposal region into the grid of the feature map, and then we are max-pooling to fixed 2×2 spatial representation which is the input for the next part.
3. **MLP** - after all processing here we use a lightweight multi-layer perceptron to process each of region feature map.
4. **Softmax cl.** and **Box regressor** - head outputs, as before in R-CNN (Section 10.4.3).

The main difference to before is that a lot of the computation has been moved from the per-region area in green to the per-image area in yellow and this makes the processing much faster, and that's why it's called Fast R-CNN.



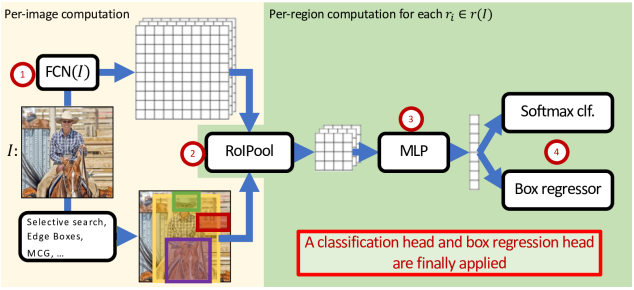
(a) R-CNN

(b) General framework

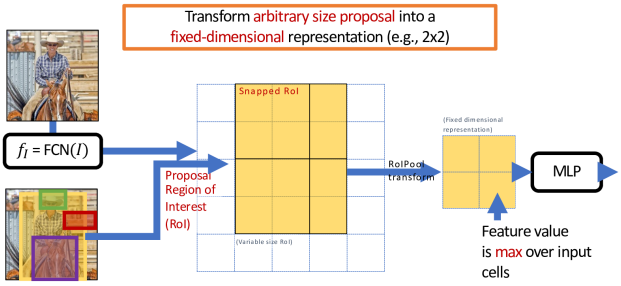


(c) R-CNN blended into the framework

Figure 122: **R-CNN and two-stage framework**: (a) Main steps of R-CNN algorithm; (b) General two-stage detection framework; (c) R-CNN blended into the framework;



(a) Fast R-CNN architecture



(b) RoI pooling

Figure 123: **Fast R-CNN**: (a) Model architecture; (b) Region-of-Interest pooling;

10.4.5 Faster R-CNN: Region Proposal Network (RPN)

The next two-stage detector is called Faster R-CNN [**?**]. The model is illustrated in Fig. 124a, and it's not using this generic region proposal mechanism, but it uses a learned proposal instead. Now we have an arrow that goes from the features of the backbone to the region proposal network, which directly regresses the proposals as well and trains the entire model jointly end-to-end.

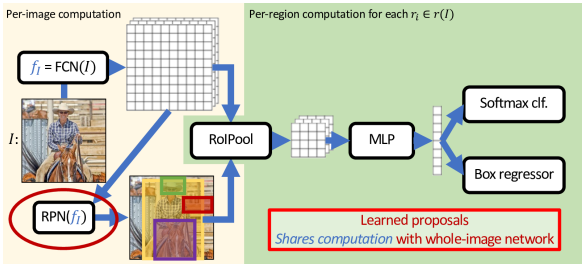
Region Proposal Network is a simple 3×3 kernel that's swept over the input convolutional feature and scans the feature map looking for generic objects, and it defines an anchor box and the predictions are with respect to this box. The anchor box is chosen larger than the 3×3 because features that are stored in this 3×3 sliding window have a larger receptive field. For this anchor box and for this region proposal network again we have an objectness classifier, where we classify if there is some object or is it background. Then we have a box regressor that predicts the offset to the anchor box.

In practice, the authors of Faster R-CNN use multiple anchor boxes to accommodate multiple ratios of objects. So we have K objectness classifiers and K box regressors.

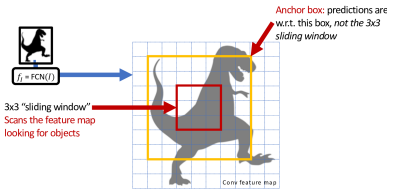
10.4.6 Feature Pyramid Network

The goal of the feature pyramid network is to improve scale equivariance because the detector needs to classify and localize objects over a wide range of scales (Fig. 125a). Some of the approaches that we can do in order to detect objects at different scales:

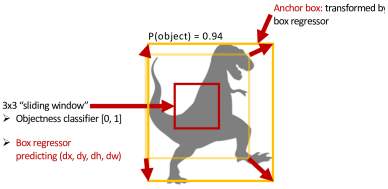
1. Fast, but suboptimal approach is to use the multiscale features like in Fast or Faster R-CNN, but we yet



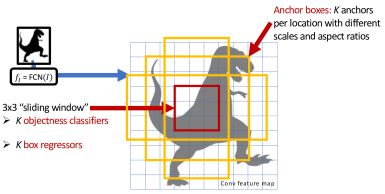
(a) Faster R-CNN architecture



(b) Anchor box prediction



(c) Prediction of the offset to the anchor box



(d) Multiple bounding boxes are used

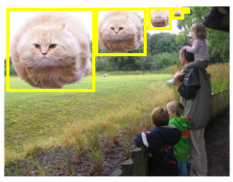
Figure 124: **Faster R-CNN**: (a) Model architecture; (b) Object classifying and anchor box prediction; (c) Prediction the offset to the anchor box; (d) Multiple bounding boxes usage;

can't detect objects at all scales equally well (Fig. 125b).

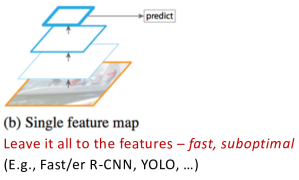
2. Next idea is an in-network pyramid where we use the internal pyramid and make predictions at every individual feature scale. The drawback is that in the first pyramid level there has relatively little processing happened when we still have high resolution and at the lower levels where more convolutional layers have been processed we have stronger features but for smaller objects (Fig. 125c).

3. The feature pyramid network idea is U-Net where we have an encoder and skip connections and then we have a decoder, so in this case we have strong features both at the high resolution and at the low resolution (Fig. 125d).

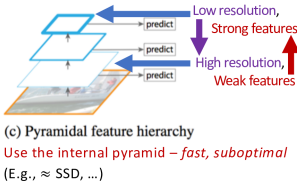
We can also generalize the proposed framework very easily to other output modalities (for example Mask R-CNN [**?**] or DensePose [**?**]).



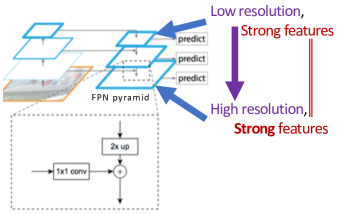
(a) Objects of different scales



(b) Single feature map approach



(c) Pyramidal feature hierarchy



(d) FPN pyramid

Figure 125: **Multiscale problem**: (a) Input image example; (b) Multiscale features approach; (c) Individual feature scale prediction approach; (d) FPN (U-Net like) approach;

10.4.7 Foreground-Background Imbalance

Foreground-Background problems: **1)** Number of positive/negative boxes heavily imbalanced; **2)** Infinite imbalance before quantization and after quantization there are only 0.01-0.1% foreground boxes; **3)** Learning from imbalanced data is difficult, because ignoring minority class gives you $\Rightarrow$ 100% accuracy.

How can this problem be improved:

1. Loss function which pays attention to hard examples, e.g. Focal Loss, where well-classified examples receive less penalty (Fig. 126).

2. Cascades, e.g. proposal-based detectors (the second stage sees more balanced data)

3. Quantization, e.g., single-stage detectors (imbalance is small by construction)

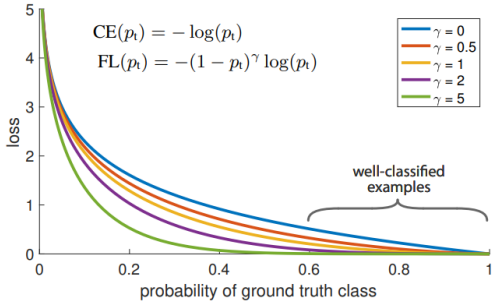


Figure 126: **Focal Loss (FL)**: gives less penalty to well-classified examples

10.4.8 Single Stage Detection

The idea of a single-stage detector is to predict the object boxes in a single forward pass, and examples of such single-stage detectors are YOLO [**?**] and SSD [**?**]. The idea here is to give the input image to predict at a very coarse resolution, in this case, 5×5 only, the class probability and to regress the bounding boxes to each of the cell and confidence scores that directly leads to the final detectionsFig. 127. Through this extreme quantization, we can do the entire processing in a single stage, and it's extremely fast, but in general, the performance is worse due to this dramatic sub-sampling of the output space and the large quantization error is hard to correct in practice which leads to a lower AP metric, than the two-stage detector.

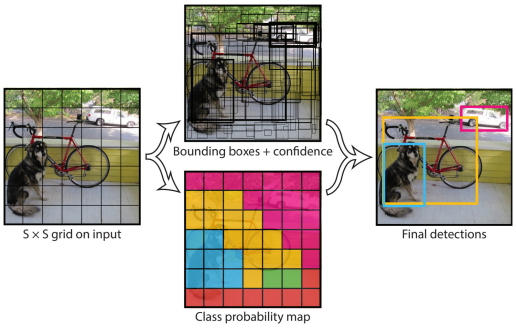


Figure 127: **YOLO**: one-stage detector

10.4.9 Paradigm Shift: DEtection TRansformer (DETR)

One prominent example of an alternative paradigm to object detection is **DEtection TRansformer (DETR)** [**?**], which uses a transformer encoder and a transformer decoder. It is inspired by language models that use a lot of transformers these days when the input image is passed through a backbone and then tokenized and the transformer produces a set of features for these tokens. Then these features are input to a decoder and then we learn jointly not only these features but also a set of object queries that are then decoded into the individual predictions.

10.5 3D Object Detection

10.5.1 3D vs 2D detection

The goal of 3D object detection is to predict a 3D bounding box for every object of interest in the scene. The performance of 3D object detection algorithms can be evaluated similarly to the 2D case by defining a 3D IoU metric computing AP metric based on 3D IoU.

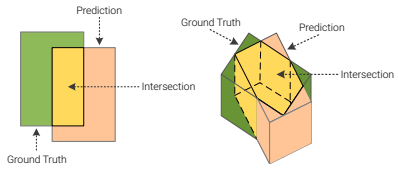
There are different parametrizations for 2D and 3D object detection which is illustrated in Fig. 128b:

- 2D Object Detection (2D Box (x, y, w, h))
- 3D Object Detection (3D Box (x, y, z, w, h, l, r, p, y))

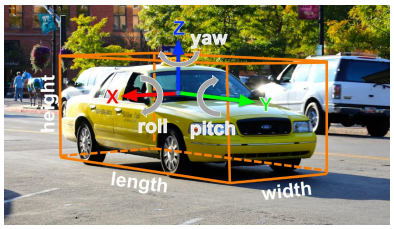
An assumption that's commonly made is that vehicles are driving on the road surface, and there is zero roll and zero pitch, so we can remove these two parameters and only estimate the yaw angle (**7-dim problem**).

One problem of 3D illustrated in Fig. 128c, the red cone is called a frustum in 3-dimensional space and the object can be located anywhere inside that frustum. That means that objects of different scales and distances may look exactly the same which is a scale ambiguity.

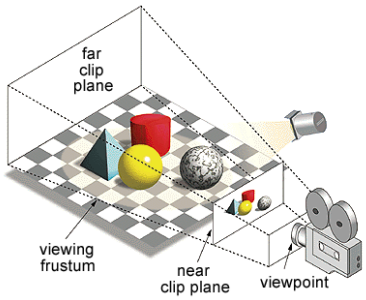
Luckily in the case of self-driving, this is something we can learn to resolve through machine learning because cars typically tend to be $4-6$ meters long, so by knowing that a car in a physical world has a typical size we can get cues about the distance to a car knowing the size of the 2D projection of that car.



(a) 2D and 3D IoU metric



(b) 3D bounding box parametrization



(c) Object detection in the frustum

Figure 128: 3D Object Detection: (a) 3D IoU; (b) 3D bounding box parametrization; (c) Detection in viewing frustum

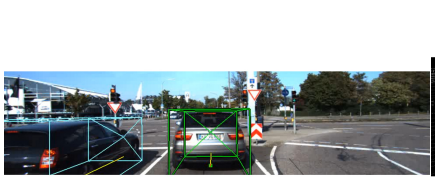
10.5.2 Input Modality

The difficulty depends on the input modality and the hardest the task is to regress 3D boxes from monocular images as illustrated in Fig. 129a because this requires very good object size priors, but particularly at the far range, it's very hard to predict the location of a 3D box image precisely.

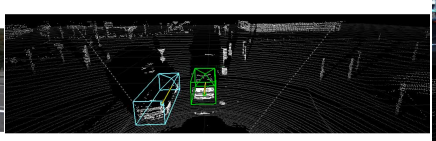
If we have two images a stereo camera then this helps to localize the box in 3D space, but we have a quadratic increase in error with distance and so it's still not as accurate as if we would have a 3D point cloud.

In contrast to images, LiDAR sensors are sparse but provide good 3D accuracy also at far range, as shown in Fig. 129b, however, due to their sparsity it could be harder to detect small objects like pedestrians.

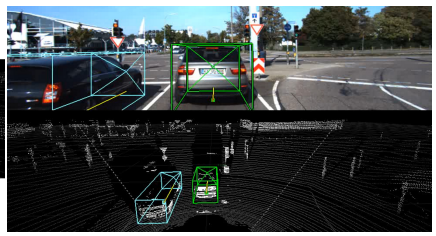
That's the reason why several methods also try to combine both modalities as they provide complementary information (Fig. 129c).



(a) Monocular input



(b) LiDAR input



(c) Both modalities combined

Figure 129: **Input modalities**: (a) Monocular RGB image; (b) LiDAR point clouds; (c) Modalities combined;

10.5.3 Image-based 3D Object Detection

- **Monocular 3D Object Detection for Autonomous Driving** [**?**]. The idea here is to sample candidate 3D boxes on the ground plane and remove all the 3D boxes that are in non-road areas. So we first do a semantic segmentation of the input image, and we assume the camera at a certain height and inclination angle.

  Then we project the semantic segmentation onto that ground plane that's what you can see in Fig. 130, we ignore the road area and sample 3D boxes of typical sizes of 3D cars in the gray area.

  Now we have sampled cars at the plausible 3D location, but there are a lot of proposals, so we project them into the image plane in order to extract features. Then we use a feature-based scoring mechanism based on features shown in Fig. 130 (shape, context, location prior, etc.) to re-rank these hypotheses. And then for the top-ranking ones they do a final scoring and a 2D refinement using a Fast R-CNN in the 2D image domain to retain the 3D proposals that are most likely.

So it's a technique that first samples in 3D and, through that sampling process automatically constrains the location and the sizes of the cars to be plausible, but then does the classification and detection actually in the 2d image space.

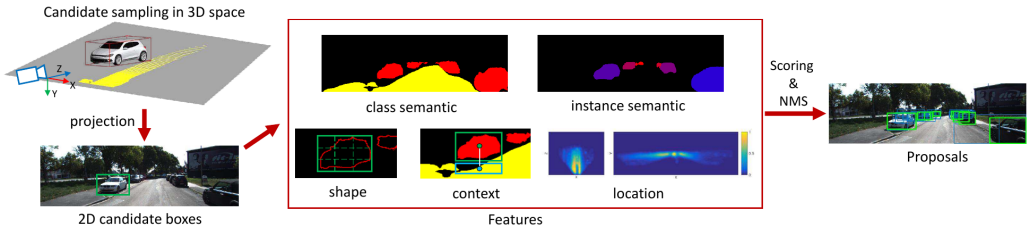


Figure 130: **Monocular 3D Object Detection for Autonomous Driving**: method pipeline

- **MonoRCNN: Monocular 3D Object Detection** [**?**]. This technique is called geometry-based distance decomposition for monocular 3D object detection (Fig. 131). The idea here is to improve localization and the main problem is that regressing depth is really hard.

  They observe that there is this relationship between the distance to the object the actual 3D height of the object and the height in the image domain of the 2D bounding box. So they regress both the physical H and the image height plus uncertainties $\Rightarrow Z = \frac{fG}{h}$

  Then fuse those in an optimal way to yield a better depth estimate, which significantly boosted the performance.

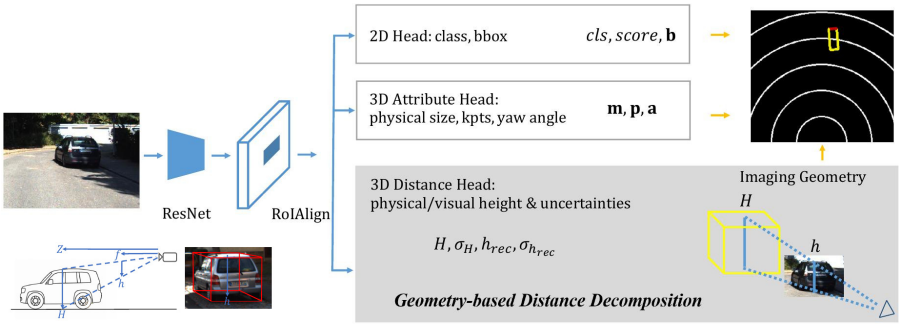


Figure 131: **Geometry-based Distance Decomposition for Monocular 3D Object Detection**: method pipeline

- **Pseudo-LiDAR: 3D Object Detection from Point Clouds** [**?**]. The idea to use a monocular image or a stereo pair to a depth map and then back-project that depth map into a 3D point cloud which the authors called a **pseudo-LiDAR point cloud**, and then you can apply standard point-based LiDAR-based 3D detection algorithms directly to this pseudo-LiDAR point cloud.

  The authors of this paper claim that performance is competitive with LiDAR-based detectors. This paper had a big splash in the community because it was believed that 3D detection from monocular images is much harder than LiDAR-based detection.

  More recently there has been a discussion around this topic because there's not that much prior knowledge actually that is incorporated into this approach, why should this work so much better going through this depth map than directly regressing 3D boxes from the image. And recently it was found that some of these improvements have actually not corresponded to actual performance.

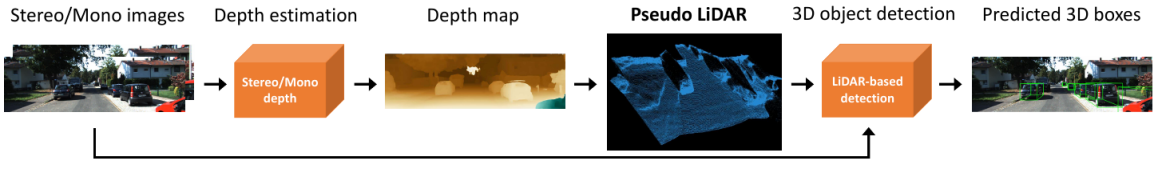


Figure 132: **Pseudo-LiDAR**: method pipeline

10.5.4 LiDAR-based 3D Object Detection

LiDAR-based 3D object detection where the input is either pseudo-LiDAR point cloud or real LiDAR point cloud from a LiDAR scanner.

Learning with point clouds poses some unique challenges compared to learning from images where we know how to deal with the image domain by using for example 2D ConvNets:

- The learned model must accommodate point clouds of arbitrary size. So if we sub-sample the point cloud we still want to have a similar result, and of course, different LiDAR point clouds have a different number of points, so the algorithm must be able to handle this.
- The learned model needs to be invariant to all (N!) permutations. As illustrated in Fig. 133 this point cloud it's basically a set, then the left image represents the same set as the right one, it doesn't depend on which index each point has in the input representation.
- The learned model should be invariant to rotations of the point cloud. The object is always the same no matter how we orient it, so we should incorporate this as well.

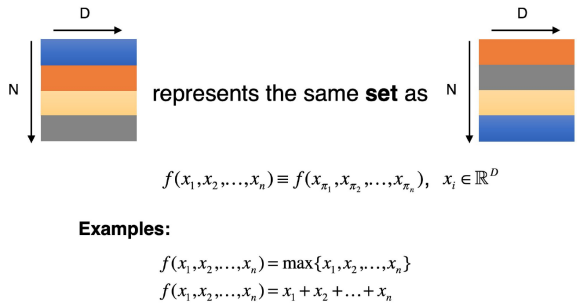


Figure 133: **Index invariant operations**: that can be used for the set representation of point cloud

In **PointNet** paper [**?**] authors tried to apply deep neural networks to point clouds on three tasks: classification, part segmentation, and semantic segmentation. In classification, the goal is to predict a category label for an entire point cloud, in semantic segmentation to predict a semantic class label for each point of that point cloud, so it's not directly the 3D detection problem, but it's a foundation of many of the 3D detection that operates on LiDAR point clouds.

- **PointNet** [**?**]. In Fig. 134 architecture of PointNet illustrated. PointNet applies a multi-layer perceptron with shared parameters per point, so they take a single three-dimensional point and project that point into a 64-dimensional feature space, and they do this independently for each point. In the second stage of this algorithm, they take a 64-dimensional feature vector of that point and transform that into a 1024-dimensional feature vector again independently per point. After that PointNet applies a max-pooling operation where this permutation invariance comes into play, in order to obtain a global feature vector of the same dimension 1024. Then a final global MLP do either classification or a decoder for semantic segmentation: it takes these global features concatenates them with some per point local features and passes them through some more PointNet layers in order to predict an output score per point.

  In addition, there is this input transformation and feature transformation blocks in the first part of Fig. 134. Inside these blocks, we use small PointNets that are called T-net, so these T-nets are entire blue blocks replicated in a small and more shallow block. T-nets are predicting a transformation in input case a 3×3 transform and in feature case a 64×64 transform. So they are predicting an affine transformation that's applied again independently per point and this helps in learning this geometric invariance.

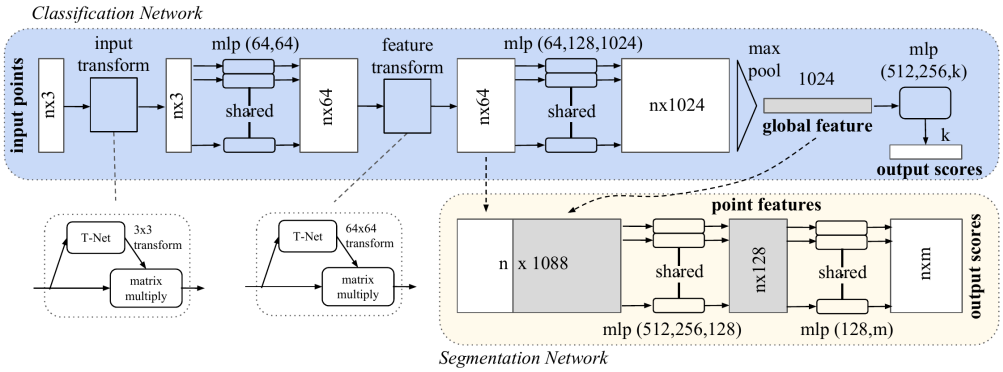


Figure 134: **PointNet**: architecture

- **PointNet++** [**?**]. One downside of PointNet is that it doesn't capture local structures and depends on global coordinates. The follow-up innovation from the same group was PointNet++ which applies PointNet recursively as you can see in Fig. 135 on nested partitions of the point set. So it is aggregating stronger features through this nested partitioning and it learns features with increasing contextual scale, so you can think of this as a "Multi-scale PointNet", and similarly to PointNet it has a decoder for semantic segmentation or MLP for classification.

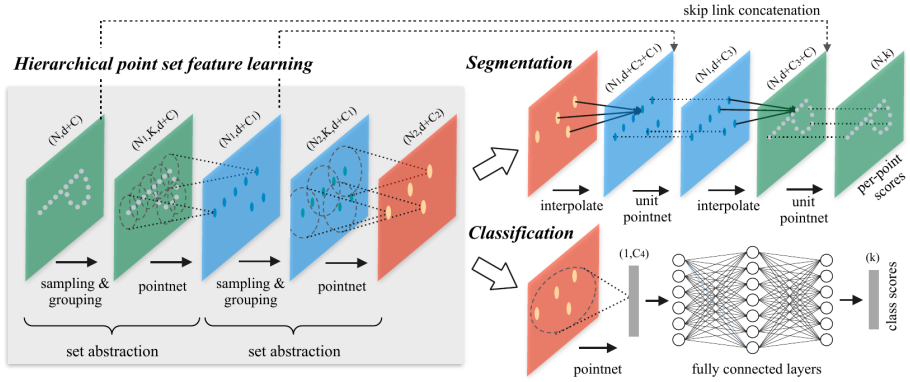


Figure 135: **PointNet++**: architecture

These models were about point cloud classification and semantic segmentation. The next methods are showing how we can use them for 3D detection:

- **PointRCNN: 3D Object Proposal Generation and Detection from Point Cloud**. [**?**] We can call this method Faster R-CNN for Point Clouds as it's basically doing exactly the same as Faster R-CNN but for point clouds.

  It has a proposal stage that generates a set of 3D box proposals that is ideally over complete space, and then it has a 3D box refinement stage where a 3D box offset is predicted as well as the confidence to yield the final 3D detections.

  So the top part of Fig. 136 is the first stage, and first, we have a backbone, point cloud encoder, and the point cloud decoder that learns discriminative point-wise features using a standard PointNet++, which gives us per point features, and then we have a 3D box proposal regression module.

  They have another module that predicts foreground versus background, and what they use is a combination of regression and classification+regression losses (for example for the spatial coordinates in bird's eye view for every point they define a local coordinate system, and then they discretize that local coordinate space and first do a classification of the location of the object in which bin it should be, and then an additional regression was exactly in that bin) which they call "bin-based" loss.

  After 3D box proposal regression, they pass the top 100 proposals in which they confident most into the next box refinement stage which is illustrated bottom part of Fig. 136.

  Then in the first stage, it pools all the 3D point features based on the 3D box proposals, so simply aggregates all the features inside any of these 3D proposal boxes, which leads to a set of semantic features and local spatial points.

Then in a second stage, these local spatial point coordinates are transformed into a local 3D box relative coordinate representation to have them in a canonical coordinate system, and then merged with the semantic features and fed as an input to a point cloud encoder that then does the 3D box refinement by predicting residuals similar to the second stage in Fast R-CNN, and again it's using a combination of regression and classification+regression losses and this gives the final 3D boxes as output.

An interesting observation from the method results is that the PointRCNN method outperforms multi-modal methods, except for pedestrians detection.

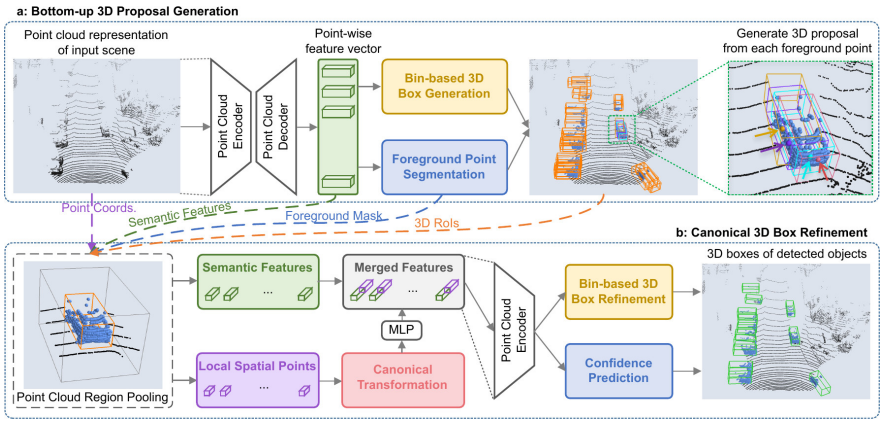


Figure 136: **PointRCNN**: method pipeline

- **VoxelNet: Single Stage Detection with 3D Convolutions** [**?**] (by Apple team). An alternative to the PointNet model is to encode points into a voxel representation and then use 3D convolutions.

  The authors of VoxelNet partitioned the input space into a set of voxels and then converted the points that fall into these voxels into features for these voxels and then applied 3D convolutions on that voxel space.

  This is a single-stage detector based on these features and the region proposal network in Fig. 137 is proposing directly the 3D box output, so it's not a two-stage detector. This also works but of course, it requires a large amount of memory and computing because you have to work in a large 3D volume.

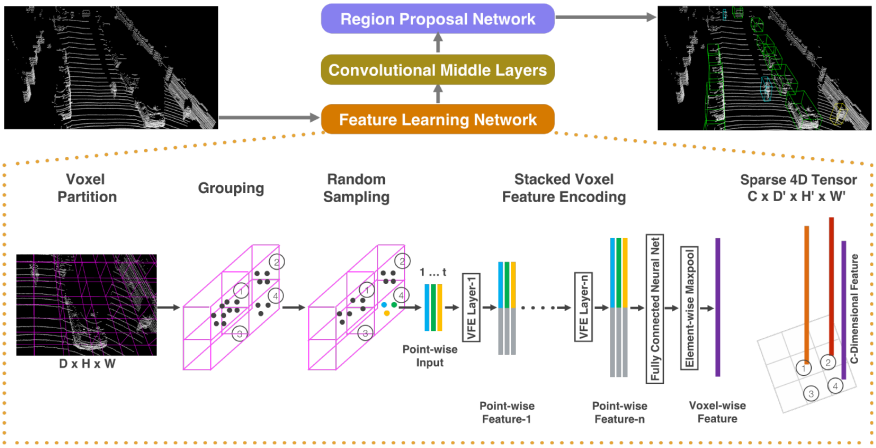


Figure 137: **VoxelNet**: method pipeline

- **PIXOR/HDNet: 2D BEV Representation** [**?** **?**]. Another idea is to use a 2D bird's eye view representation instead, which is much more memory and computation-efficient. The idea is to represent LiDAR information and possibly also a road map that can be used as auxiliary information to help object detection (Fig. 138). This is done for the LiDAR to treat the Z dimension as a featured channel, to discretize the Z space into different bins, and use these as the feature channel because then you can use efficient 2D convolutions instead of applying 3D convolutions.

  It has also been shown that if you supplement this LiDAR point cloud with a map, for example, a high definition HD map, where you have localized yourself in that map, these map priors can help you invalidate implausible 3D detections from your 3D detector and gain better performance.

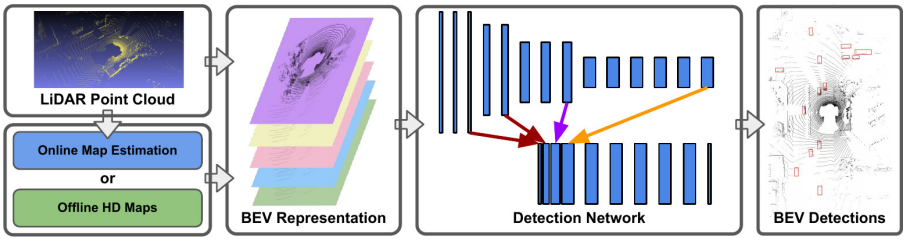


Figure 138: **HDNet**: method pipeline

10.5.5 Multi-modal 3D Object Detection

- **Frustum PointNet: Detect 3D Boxes in Object Frustum** [**?**]. The idea is to generate 2D object proposals in the RGB image first. So we first take the LiDAR scan, project the LiDAR points into the image domain, and use that with a standard 2D ConvNet to obtain 2D object proposals like in Faster R-CNN.

  Then they compute the object frustum corresponding to the proposal 2D bounding boxes, consider all the 3D points that lie inside that frustum, and predict a 3D bounding box from points inside the frustum.

  It's doing a little fusion in the first object proposal step, but then it's a two-stage mechanism where the first stage in the image domain proposals are generated, and then in a 3D domain with a PointNet, these proposals are validated.

  You can also find more details in Fig. 139, where 3D box predition problem is split into 3 stages: **1) Segmentation** of RGB point cloud (object vs. background); **2) Translation** of points such that centroid aligns with 3D box center; **3) 3D bounding box regression** (x, y, z, w, h, l, y).

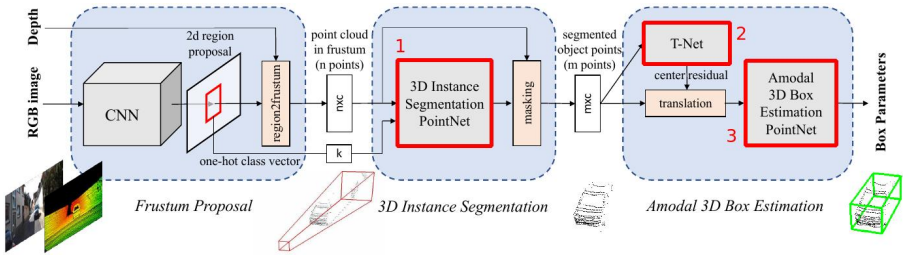


Figure 139: **Frustum PointNet**: method pipeline

- **MV3D: Combining Multiple 2D Views (RGB/LiDAR Projections)** [**?**]. This work is based on the popular two-stage object detection framework where we have a 3D proposal network and region-based fusion network here at the end (Fig. 140).

  In a first step, this method generates 3D proposals from the LiDAR bird's eye view and then projects them into the LiDAR front view, and we also project this 3D proposal into the RGB image.

  Each of three different input modalities has a convolutional network to extract features, and then we can apply RoI pooling just like in Faster R-CNN to extract those features in these projected regions. And then, we fuse the RoI pooled features from all modalities using this fusion mechanism which is iterating between doing computation per branch and then computing a mean pooling of these modalities. And at the end, we have a head where we regress the class and also the box residuals, so again it's a classical proposal plus refinement two-stage pipeline.

  From the results, we can conclude that combining the monocular image queue with the LiDAR queue gives something, but it's not quite as dramatic as we might hope.

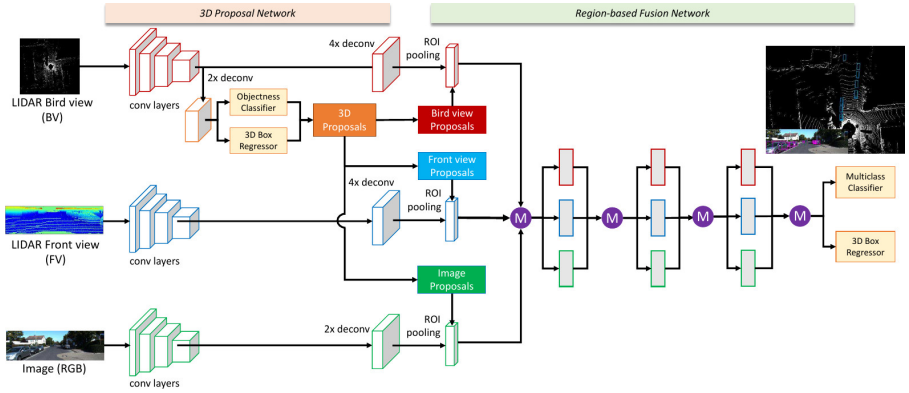


Figure 140: **MV3D**: method pipeline

10.5.6 Summary

Localizing and recognizing objects is crucial for self-driving. However, there are (∞) many possible boxes and a number of objects unknown. Challenges for raw RGB images are appearance/viewpoint variation, illumination, clutter, occlusion. Detection performance is typically measured using average precision. Classic sliding-window detection methods use hand-crafted features. Deep learning has boosted recognition performance by 10x in a few years. Two-stage detection methods (Faster R-CNN, Mask R-CNN) are state-of-the-art. Feature pyramid networks help detect features at various scales. 3D detection methods rely on RGB images, LiDAR point clouds, or both. LiDAR information is crucial for accurate 3D localization.

11 Object Tracking

11.1 Introduction

A common problem that is faced in self-driving is object tracking. As discussed in the previous lecture, we need to know what´s going on in a scene to navigate through it. Therefore, objects like cars or pedestrians must be detected. It is not enough to simply detect objects in every frame as we move through a scene. For a more refined understanding of the situation, each noisy detection must be associated to the physical object they belong to. This is the setting of the tracking problem.
Moreover, object detections that are false positives must be rejected and object tracks must be initiated or deleted as objects enter or leave the scene. An object track is a sequence of detections that are associated with one and the same physical object. Given the observation of an object track and a suitable motion model, the object state can be estimated. So, it can be predicted how an object will move through the scene, based on its previous locations and a basic understanding of how the object moves.

11.1.1 Elements of Tracking

Object tracking can be divided into three essential problems. That is **Detection**, **Association**, and **Filtering**. The input for these problems is either a 3D point cloud sequence or a video.
In **Detection**, candidate objects are detected in every frame. Subsequently, in **Association**, an attempt is made to assign these detections to the correct object. This is the most important paradigm of tracking and it is called 'tracking-by-detection'. Finally, we try to determine the most likely object state using **Filtering** to refine the noisy observation of object position or size using probabilistic motion models and previous observations. This is often done with a Bayes filter which comes in many flavours for different situations. More on Filtering in the following sections.

11.1.2 Online vs. Offline Tracking

In general, there are two main ways to go about applying object tracking. Online and Offline Tracking. In Online Tracking, the current state of an object is estimated given only the current and the past observations. In Offline Tracking on the other hand, all states are estimated simultaneously, using all observations. One might think that more observations give a better basis for state estimation of an object. While this is true, it is unpractical for the self-driving problem, since objects must be continuously recognized. We cannot wait for future observations to receive a better estimation. The states can also be estimated from smaller batches of observations, yielding a more feasible approach for live action object tracking. But it is still too slow for self-driving.

11.2 Filtering

State vs. Observation. To understand filtering better, we can look at it in a more abstract way. The parameters of interest, e.g. object location, can be seen as 'hidden states'. They usually differ from the observation that is actually made. This is due to limited observability of the hidden state or noise of the measurement. For instance, if we are interested in the velocity of a rolling ball, we can only observe its position at different time steps. We then need to infer the velocity from that using a motion model of the ball. The velocity can´t be measured directly. As seen in Fig. 141, a filter combines the probabilistic motion model and noisy measurements to infer a better estimate of the hidden state.

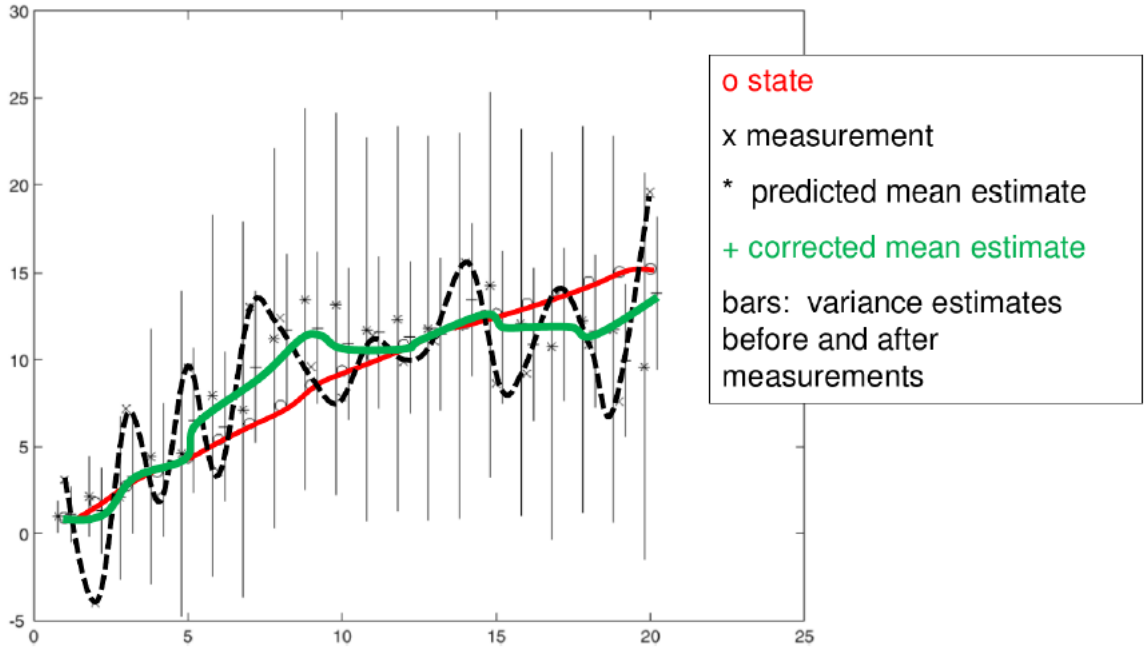


Figure 141: **Application of a filter.** The red line is the ground truth. The dashed line represents the measurements. The measurements and the uncertain predicted mean estimates (high variance) are combined to yield the corrected mean estimates with lower variance.

Representation and Filtering. This process can be illustrated by the following representation, which is a running example in this subsection. For this representation, let´s assume that we have a single object that needs to be tracked. The moving object of interest is characterized by an underlying state x. State x gives rise to measurements or observations y. At each time t, the state changes to x_t and a new observation y_t can be made. Given a sequence of observations $Y = \{y_t\}$, the aim is to recover $X = \{x_t\}$. This representation is depicted in Fig. 142.

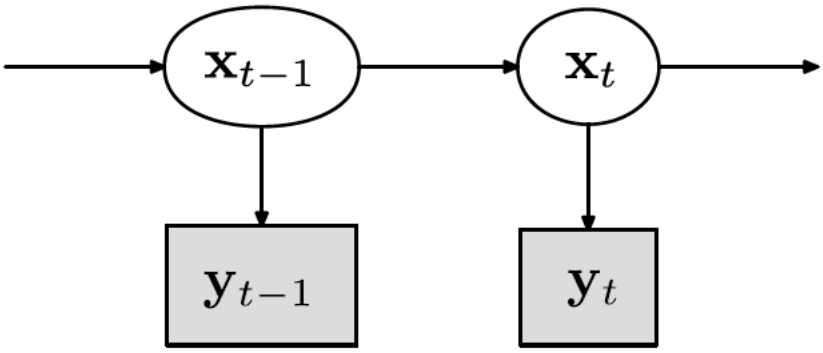


Figure 142: **Part of a Markov model** with hidden states X and observations Y.

Note that the observations are independent. They only depend on the state of the same time step. The current state depends on the previous state. The state is usually not a point estimate. It is rather a probabilistic estimate, which means it is a distribution of, e.g., object location. This will allow us to know the most likely location and how sure we are about it.

11.2.1 Probability Theory Recap

Random Variables. A discrete random variable, short r.v., $x \in \{1, ..., C\}$ relates to the probability that x takes the value c, namely $p(x = c)$. A continuous random variable, which is mostly considered in the following, relates to the probability that x takes a value in $\mathbb{A} \subset \mathbb{R}$, which is $p(x \in \mathbb{A})$. The distribution over r.v. x is written as $p(x)$ as short notation for $p(x = c)$ or $p(x \in \mathbb{A})$. A joint distribution of r.v. x and y is written as

$p(x, y)$ as short notation for $p(x = c, y = c')$. Some helpful properties or relation of these distribution are the marginal (over x) and the product rule:

$$p(x) = \int_y p(x, y) \tag{137}$$

$$p(x, y) = p(x|y) \cdot p(y) \tag{138}$$

Based on the product rule in Eq. (138), we have:

$$p(x, y) = p(x|y) \cdot p(y) \text{ or}$$
$$p(x, y) = p(y|x) \cdot p(x)$$

Solving for $p(x|y)$ yields **Bayes' rule**:

$$p(x|y) = \frac{p(y|x) \cdot p(x)}{p(y)} \tag{139}$$

Similarly, for more r.v.´s, we have:

$$p(x, y, z) = p(x|y, z)p(y, z) = p(y|x, z)p(x, z) = p(y, x, z) \tag{140}$$

Solving for $p(x|y, z)$ yields:

$$p(x|y, z) = \frac{p(y|x, z)p(x, z)}{p(y, z)} = \frac{p(y|x, z)p(x|z)p(z)}{p(y|z)p(z)}$$

$$p(x|y, z) = \frac{p(y|x, zp(x|z)}{p(y|z)} \tag{141}$$

So, Bayes´rule also holds when conditioning on more than one random variable. This property is needed later on.

11.2.2 The Bayes Filter

Recursive Bayesian estimation. As mentioned before, filtering is often done by using a Bayes filter. A Bayes filter is a probabilistic approach for estimating an unknown probability density function recursively over time using incoming measurements, e.g. detected object location, and a system process model like a constant velocity motion model. Ideally, the motion model is precise. It then can be used for association, since, e.g., the bounding box for an object is easy to find in the next frame if the prediction from the motion model is precise. The estimation is called recursive because it involves two alternating steps, namely prediction and correction. In the context of object tracking, the aim is to **predict** where the object should be in the next frame. Subsequently, the prediction must be **corrected** based on the current observation to yield a solid estimate and do accurate association with a detection of the next frame.

Assumptions. For formalizing the Bayes filter, let´s assume there is only one object to be tracked. Further assume that exactly one noisy observation is available per frame. The aim of the Bayes filter is to do probabilistic state inference. Recall that this means **distributions** of the hidden parameters are estimated.
In addition, general assumptions regarding our self-driving context have to be made. Assume the following: The camera is not moving instantly to a new viewpoint. Objects do not disappear and reappear in different places, which in truth might occur because objects can be occluded by other objects completely or for some period of time. These assumption mean that there is a gradual change in pose between camera and scene, not the camera nor the objects make large jumps in position. The biggest assumption of the Bayes filter is that the state follows a Markov process. It means that the state is only dependent on its previous state. The Kalman filter, a version of the Bayes filter, additionally assumes linear and Gaussian motion and observation models to yield a practical closed form that is easy to compute. There are other more advanced models like the Extended Kalman Filter (EKF), Unscented Kalman Filter (UKF) and Particle Filter, which are less restrictive but more complex and computationally heavy.

11.2.3 Formal Definition of Bayes Filter

Let us now formulate the Bayes filter mathematically. Let $x_t \in \mathbb{R}^M$ denote the true **hidden** state of the system at time t. Let $y_t \in \mathbb{R}^M$ denote some noisy measurement of x_t at time t. The Bayes filter assumes that $\mathbf{X} = \{x_t\}$ is an unobserved **Markov process** and $\mathbf{Y} = \{y_t\}$ are the corresponding measurements:

$$p(\mathbf{X}, \mathbf{Y}) = p(x_1) \prod_{t=2}^{T} p(x_t|x_{t-1}) \prod_{t=1}^{T} p(y_t|x_t) \tag{142}$$

Here, $p(x_1)$ is the prior distribution over the state at the beginning of the sequence. The goal is now to perform inference in this **Hidden Markov Model**, which means estimating **X** from **Y**. Remember Fig. 142 to visualize this.
For **recursive state estimation**, we take advantage of **Bayes rule:**

$$\underset{\textit{Posterior}}{p(x_t|y_{1:t})} = \frac{p(y_t|x_t, y_{1:t-1})p(x_t|y_{1:t-1})}{p(y_t|y_{1:t-1})} \propto \underset{\textit{Observation}}{p(y_t|x_t)} \; \underset{\textit{Prediction}}{p(x_t|y_{1:t-1})}$$

The **predictive distribution** can be further decomposed as:

$$\begin{aligned}
\underset{\textit{Prediction}}{p(x_t|y_{1:t-1})} &= \int p(x_t, x_{t-1}|y_{1:t-1})dx_{t-1} \\
&= \int p(x_t|x_{t-1}, y_{1:t-1})p(x_{t-1}|y_{1:t-1})dx_{t-1} \\
&= \int \underset{\text{Motion Model}}{p(x_t|x_{t-1})} \; \underset{\text{Prev. Posterior}}{p(x_{t-1}|y_{1:t-1})}dx_{t-1}
\end{aligned}$$

Due to Eq. (137), it is clear that the prediction is equal to the joint distribution of x_t and x_{t-1} given $y_{1:t-1}$ and marginalized over x_{t-1}. With Eq. (138), we yield the second derivation. To get to the last line, the Markov property is applied. The measurements $y_{1:t-1}$ can be omitted because the state x_t is independent of them.
A **recursive state estimation algorithm** is obtained:

1. Prediction

$$\underset{\textit{Prediction}}{p(x_t|y_{1:t-1})} = \int \underset{\text{Motion Model}}{p(x_t|x_{t-1})} \; \underset{\text{Prev. Posterior}}{p(x_{t-1}|y_{1:t-1})}dx_{t-1} \tag{143}$$

2. Correction

$$\underset{\textit{Posterior}}{p(x_t|y_{1:t})} \propto \underset{\textit{Observation}}{p(y_t|x_t)} \; \underset{\textit{Prediction}}{p(x_t|y_{1:t-1})} \tag{144}$$

The Bayes filter runs **online** (Section 11.1.2), so it updates per frame as it is needed in self-driving.

11.2.4 The Kalman Filter

Now let´s focus on the simplest version of the Bayes filter that is also most prominent. It is called Kalman Filter (KF). As mentioned before in Section 11.2.2, the KF assumes linear and Gaussian motion and observation models. Specifically, when looking at Eq. (142), the motion model $p(x_t|x_{t-1})$ and the observation model $p(y_t|x_t)$ are assumed to be linear and normally distributed. Since normal distributions are completely characterized by their mean and covariance, only these parameters are tracket with the KF. The solution for the KF is given in closed form as products and marginals in Eq. (144) and Eq. (143) are **Gaussian**. As a result, the maximum likelihood equals to a least squares fit, which is a really simple formula. The drawback of the KF are its strong assumptions. Most problems faced are of non-linear nature, so the KF cannot be applied there. That´s why there exist non-linear extensions of the KF, namely the EKF and UKF. Still, the KF is the optimal linear filter in least squares sense.

The Particle Filter also handles multimodal Gaussians and computes a sampling-based approximation to the sate posterior $p(x_t|y_{1:t})$. Multimodal Gaussians are Gaussian distributions that have multiple peaks. The KF only deals with unimodal Gaussians which have a single peak. More recently, learning-based motion models, e.g. RNN, have been explored to deal with non-linear problems.
For a better understanding of KF, the following figure illustrates the prediction and correction loop.

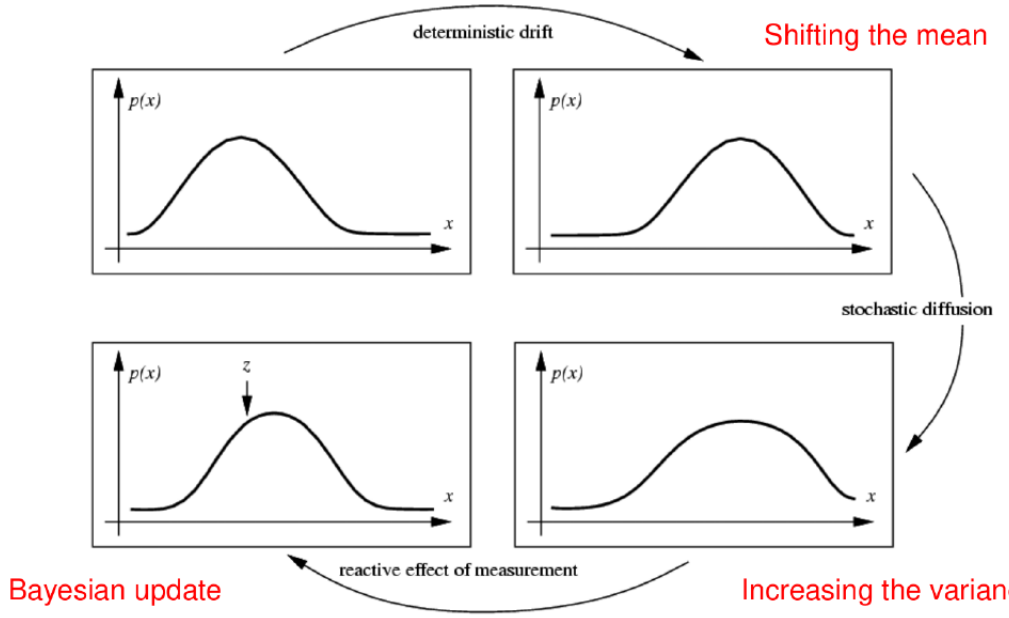


Figure 143: **Prediction and correction loop.** X is a linear and Gaussian hidden variable. $p(X)$ is its pdf. A prediction based on a linear motion model shifts the mean. However, the prediction is uncertain. This is expressed through stochastic diffusion, which increases the variance of the pdf. Averaging over the measurement z and the distribution yields a corrected estimate of the hidden variable. This is a Bayesian update.

Density propagation. Fig. 143 depicts a distribution of a hidden variable x, e.g. object location, which is tracked using the KF. Starting in the top right graph: Using our motion model, the distribution of the future location of the object is predicted slightly to the right of the old location. As the prediction is uncertain, the predicted distribution will have an increased variance through stochastic diffusion. Subsequently, we are averaging the effect of the prediction and observation z to yield the corrected prediction of our hidden variable. From there, the loop of prediction and correction can be continued.

Examples. Let´s formulate two example linear cases to clarify this:

Constant Velocity: (state vector: position/velocity $x_t = (p_t, v_t)^T$)

$$x_t = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} x_{t-1} + \underset{\sim \mathbb{N}(0, \boldsymbol{\Sigma}_m)}{\boldsymbol{\epsilon}_m} \qquad y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} x_t + \underset{\sim \mathbb{N}(0, \boldsymbol{\Sigma}_o)}{\boldsymbol{\epsilon}_o} \tag{145}$$

(146)

Constant Acceleration: (state vector: position/velocity/acceleration $x_t = (p_t, v_t, a_t)^T$)

$$x_t = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}(\Delta t)^2 \\ 0 & 1 & \Delta t \\ 0 & 1 & 1 \end{bmatrix} x_{t-1} + \underset{\sim \mathbb{N}(0, \boldsymbol{\Sigma}_m)}{\boldsymbol{\epsilon}_m} \qquad y_t = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} x_t + \underset{\sim \mathbb{N}(0, \boldsymbol{\Sigma}_o)}{\boldsymbol{\epsilon}_o} \tag{147}$$

(148)

These equations are linear, since they follow the linear form $y = \mathbf{A}x + \mathbf{b}$. Calculating x_t in Eq. (145) yields $\binom{p_{t-1} + \Delta t \cdot v_{t-1}}{v_{t-1}}$ plus a vector of uniformly drawn numbers between 0 and $\boldsymbol{\Sigma}_m$. So the new location according to this motion model is just the previous location plus the previous velocity over the time difference of frame $t-1$ and t. That´s the deterministic part of the model. The stochastic part $\boldsymbol{\epsilon}_m$ adds uncertainty. Eq. (146) and Eq. (148) show that the observation here only consists of the current location p_t plus random noise $\boldsymbol{\epsilon}_o$. The constant velocity model could be applied to our rolling ball example from Section 11.2.

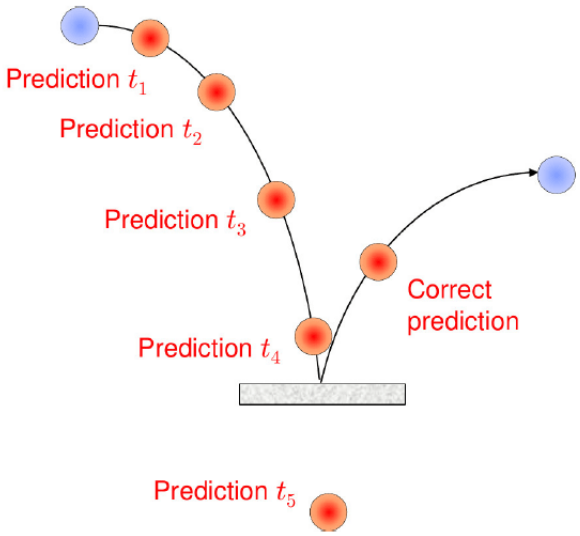


Figure 144: **Prediction of a bouncing ball.** The ball is falling to the ground and bounces back up, but the linear motion model cannot predict the bounce. It assumes that the ball will continue to fall. The **correct** prediction is far away.

Many interesting problems don´t have linear dynamics like Eq. (145) and Eq. (147), e.g., a pedestrian or even a bouncing ball. A constant acceleration model would predict too far from a bouncing balls true position as seen in Fig. 144. The linear model would give Prediction t_5, which is below the greyish floor, because it has no sense of a bounce. As a result, the prediction is too far away from the true position to yield a meaningful correction in the correction step and it would strongly violate the linear model. Incorporating the bounce in the motion model would make it non-linear.

Despite its shortcomings, the KF has seen great success. It can be a fast and simple method for Single Object Tracking, where it smooths or filters out much of the noise of observations. It was even successfully used on the moon landing.

11.3 Association

In this part, the second essential element of object tracking is explored, Association.

11.3.1 Multi-Object Tracking

In self-driving, **multiple objects** must be tracked at the same time. So how can we **associate detections** in a new frame to existing object tracks? The general approach is sketched in the following algorithm:

1. Predict objects from previous frame and detect objects in current frame
2. Associate detections to object tracks (initiate/delete tracks if necessary)
3. Correct predictions with observations (e.g., Kalman Filter)

In other words, the objects of the previous frame are given and their new location in the current frame must be predicted. Objects in the current frame are detected independent from predictions. Now, the detections have to be correctly assigned to the object tracks using some association measure. A naive approach is to measure the overlap of the prediction and the detection. There are several other association measures which are listed below and sketched in Fig. 145:

- Compare color histograms or normalized cross-correlation
- Estimate optical flow and measure agreement
- Compare relative location and size of bounding box
- Compare orientation of detected objects

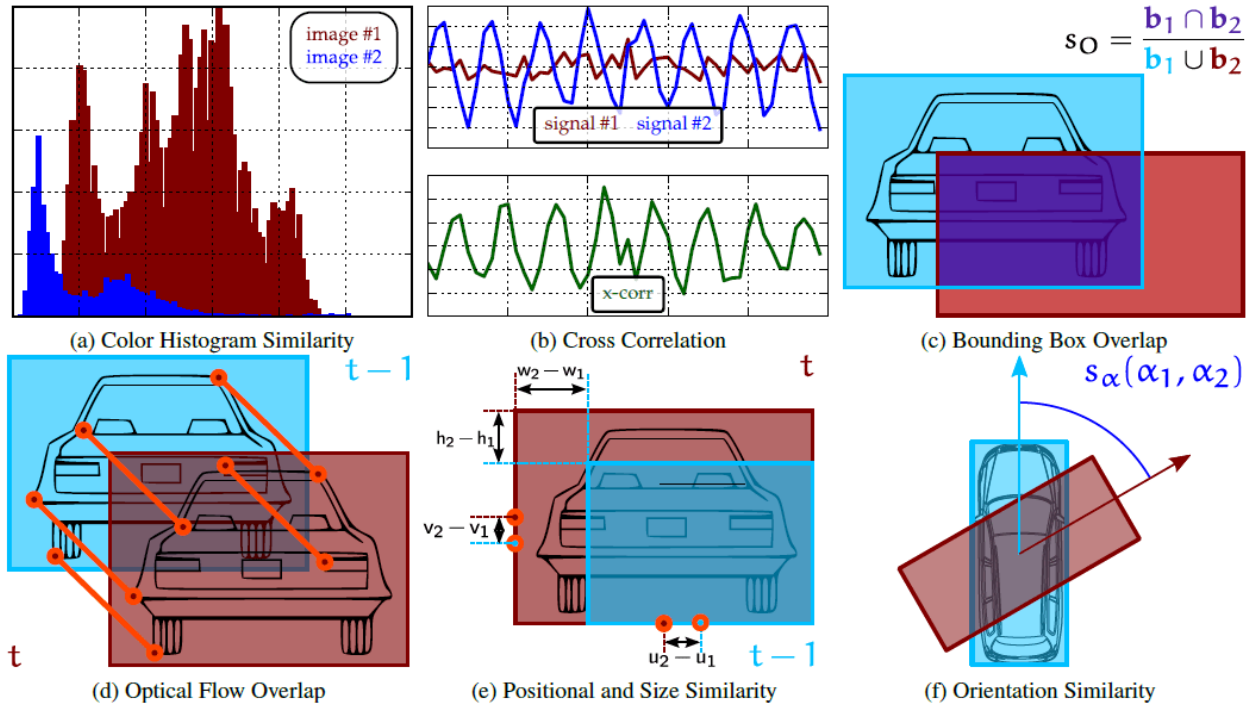


Figure 145: **Object association measures.**

11.3.2 Metric Learning

While some of the previous association measures can be simple and effective, a learning approach can do much better. The idea is to use convolutional networks to **learn an object representation** $f_\theta(\mathbf{I})$. A good representation maps images of the same object to **similar features**. Let´s consider an image of an example object that is called anchor $\mathbf{A}$. An image of a detection is sampled. If that image shows the anchor object it is called a positive sample $\mathbf{P}$. If it shows another object, it is called a negative sample $\mathbf{N}$. If f_θ is a good representation and maps to similar features, then the euclidean distance from the representation f_θ of $\mathbf{A}$ to the representation f_θ of $\mathbf{P}$ must be smaller than the distance from $\mathbf{A}$ to $\mathbf{N}$. Let´s formalize:

$$||f_\theta(\mathbf{A}) - f_\theta(\mathbf{P})||^2 < ||f_\theta(\mathbf{A}) - f_\theta(\mathbf{N})||^2 \tag{149}$$

This is a ranking that needs to be learned. It´s done by using the following **triplet loss** function:

$$\mathcal{L} = \max(0, ||f_\theta(\mathbf{A}) - f_\theta(\mathbf{P})||^2 - ||f_\theta(\mathbf{A}) - f_\theta(\mathbf{N})||^2 + m) \tag{150}$$

The loss in Eq. (150) is quite different to a normal loss function. Here, no gradient is created, because we are clipping at zero within max().

The right argument of max means that the distances to positive and negative samples $\mathbf{P}$ and $\mathbf{N}$ should be as far apart as possible. Since we solve Eq. (150) wrt. the inequality Eq. (149), we aim at $d(\mathbf{AP}) = ||f_\theta(\mathbf{A}) - f_\theta(\mathbf{P})||^2$ to be small and $d(\mathbf{AN}) = ||f_\theta(\mathbf{A}) - f_\theta(\mathbf{N})||^2$ to be big, so that the right argument of max() will be negative and zero is received overall. The margin m ensures that the loss focuses on the difficult cases. The triplet loss naturally focuses on the cases where $d(\mathbf{AP}) > d(\mathbf{AN})$, but if $d(\mathbf{AP})$ is only slightly smaller than $d(\mathbf{AN})$, this is still considered as a difficult case. A difficult case means that it´s harder to differentiate between $\mathbf{P}$ and $\mathbf{N}$. These are the cases the training needs to be focused on, and m makes sure they are included. It is effectively a threshold that determines which cases are considered worthwhile to train on. Cases that are not worthwhile are excluded by clipping at zero. Specifically searching for difficult cases and mainly presenting them to the learning algorithm is called Hard-negative mining. It is a method to improve training. Other methods are intelligent sampling or group loss [**?**].

11.3.3 Correspondence Ambiguities

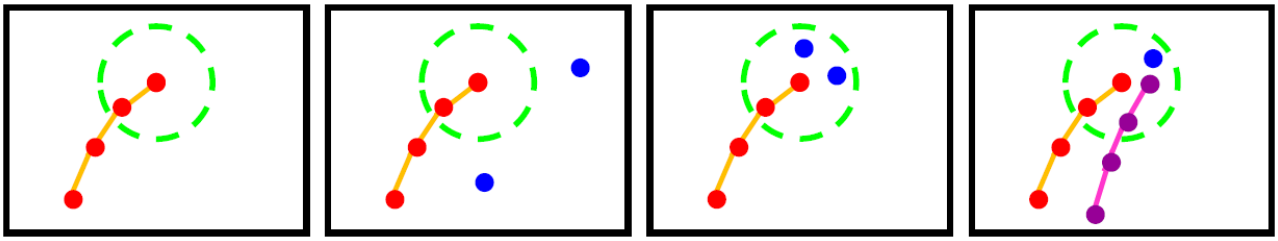


Figure 146: **Correspondence ambiguities.** The green dashed line is a threshold for association to a current prediction of an object track. Red and purple circles are state estimations including the current prediction, connected by orange and pink edges that stand for associations. Blue circles are observations. Description of images from left to right: No measurement is available. Multiple measurements are available, but don´t match the prediction. Multiple measurements match the prediction. A measurement matches multiple object tracks.

Fig. 146 shows different problems faced in Association. From left to right, the first case shows a situation where no measurement is in the vicinity of the prediction. Therefore, the prediction is not supported by the observation. In this case, the tracked object might have ceased to exist in the frame, got occluded by other objects or the detection simply failed. Next, there may be unexpected measurements that don´t fit the prediction, since they lie outside the threshold of association. New visible objects may have appeared. Detector noise could also be responsible. Furthermore, there is a chance that more than one measurement may match a prediction, so the correct one must be found. Lastly, a measurement may match multiple predictions in which case it must be assigned to the correct object track.

When looking at solutions for Association, correspondence ambiguities must be kept in mind. The following sections explore these solutions.

11.3.4 Nearest Neighbor Association

A naive approach to the association problem is Nearest Neighbor Association. Here, only measurements within a certain **gating area** around the prediction are considered. The detection $y_t \in \mathcal{Y}_t$ that is **closest to the prediction** is associated with it:

$$\underset{y_t \in \mathcal{Y}_t}{\operatorname{argmin}} \; ||y_t - x_t||_2 = \underset{y_t \in \mathcal{Y}_t}{\operatorname{argmin}} \; \sqrt{(y_t - x_t)^T (y_t - x)} \tag{151}$$

A better approach is to consider the detection that is **most likely under the prediction model** if a Gaussian prediction model $p(x_t|x_{t-1})$ is used. This is true, e.g., when using a KF. This is how to do it:

$$\underset{y_t \in \mathcal{Y}_t}{\operatorname{argmax}} \; \mathcal{N}(y_t|x_t, \mathbf{\Sigma}_t) = \underset{y_t \in \mathcal{Y}_t}{\operatorname{argmin}} \; \underbrace{\sqrt{(y_t - x_t)^T \mathbf{\Sigma}_t^{-1} (y_t - x)}}_{Mahalanobis\ Distance} \tag{152}$$

The Mahalonobis distance can also be used to define the gating area. Nearest Neighbor Association may seem like a simple solution. However, a simple example proves otherwise.

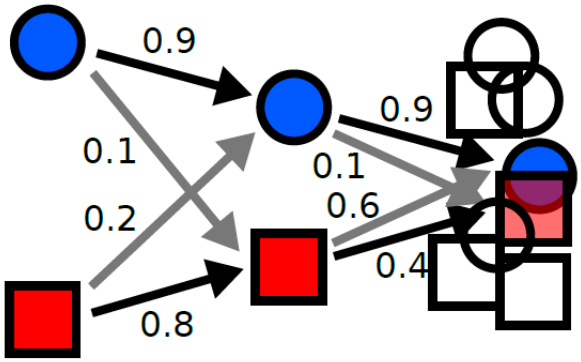


Figure 147: **Nearest Neighbor Association example.** Blue and red are object tracks. The edge values correspond to Eq. (152). As the tracks approach each other and more detections appear close, these probabilities become less helpful in differentiating the tracks.

As we can see in Fig. 147, choosing the nearest neighbor often fails due to ambiguities, so this approach is generally not a good idea.

11.3.5 Bipartite Graph Matching

In practice, there are many noisy detections, so correspondence ambiguities make it hard to decide when to continue, start or kill a track with the Nearest Neighbor approach. When solving Association with Bipartite Graph Matching, this problem can be circumvented, as it will be seen later on.

Let´s formulate Bipartite Graph Matching. Given a bipartite graph, a matching is a subset of the edges for which every vertex belongs to exactly one of the edges.
Frame-to-frame tracking can be formulated as a bipartite graph matching problem. All the objects in frame t and frame $t-1$ are nodes in the graph. All the nodes of one frame are connected to all the nodes of the other frame. The edges are equivalent to matches of the corresponding objects and hold positive association scores. They can be represented with a table. Note that objects of frame t already belong to an object track. So the rows of the table represent the tracks. The columns represent observations of objects. Typically there will be a different number of tracks and observations. The goal is now to choose at most one match in each row and column of the table while maximizing the score.

Formal definition. Assume N detections in the previous and M detections in the current frame. A M x N table of matching scores can be constructed. One matching must be chosen which maximizes the sum of scores. In detail, given an M x N matrix of association scores $\mathbf{S}$, a binary M x N matrix $\mathbf{Z}$ that maximizes the total score must be determined while only one entry of $\mathbf{Z}$ per column and per row can be 1:

$$\underset{\mathbf{Z}}{\operatorname{argmax}} \sum_{i=1}^{M}\sum_{j=1}^{N} s_{ij} z_{ij} \tag{153}$$

$$s.t. \forall_j : \sum_{i=1}^{M} z_{ij} = 1 \quad \forall_i : \sum_{i=1}^{N} z_{ij} = 1 \quad z_{ij} \in \{0,1\} \tag{154}$$

This is an Integer Linear Program (ILP), but is is not NP hard as we will see. $N!$ permutations of $\mathbf{Z}$ and therefore $N!$ different matching assignments are possible. In practice, there are 10-100 detections. So, searching through all possible permutations to find the optimal solutions becomes intractable very quickly. A faster algorithm is needed.

A Greedy Algorithm is a simple method for finding $\mathbf{Z}$. We start with an unmarked matrix. For columns $i = 1$ to N, we mark the largest value that isn´t in a row or column with already marked entries. See the following figure for visualization.

	1	2	3	4	5
1	0.95	0.76	0.62	0.41	0.06
2	0.23	0.46	0.79	0.94	0.35
3	0.61	0.02	0.92	0.92	0.81
4	0.49	0.82	0.74	0.41	0.01
5	0.89	0.44	0.18	0.89	0.14

(a) $i = 1$

	1	2	3	4	5
1	0.95	0.76	0.62	0.41	0.06
2	0.23	0.46	0.79	0.94	0.35
3	0.61	0.02	0.92	0.92	0.81
4	0.49	0.82	0.74	0.41	0.01
5	0.89	0.44	0.18	0.89	0.14

(b) $i = 2$

	1	2	3	4	5
1	0.95	0.76	0.62	0.41	0.06
2	0.23	0.46	0.79	0.94	0.35
3	0.61	0.02	0.92	0.92	0.81
4	0.49	0.82	0.74	0.41	0.01
5	0.89	0.44	0.18	0.89	0.14

(c) $i = N$

Figure 148: **Greedy algorithm.** The algorithm searches through the rows of the matching scores matrix to find the highest score and mark it.

The Greedy Algorithm is easy to implement, quick to run and it is found that it provides good solutions. However, greedy matching often does not yield the optimal solution. There are other algorithms like the Hungarian method [**?**] and Ford-Fulkerson algorithm which which find the optimal solution in acceptable polynomial time, $O(N^4)$ and $O(N^3)$, respectively.

Handling missing tracks and detections. So far, correspondence ambiguities of Section 11.3.3 were not considered in our algorithm. The last three ambiguities of Fig. 146 are resolved by the algorithm. However, missing detections or even missing object tracks can be a problem. When an object disappears the track is still alive but there are no viable detections to associate to that track, so all association scores in the row of that

track are low. The greedy algorithm doesn´t care if none of the objects in the next frame are a good match. It just maximizes the score. Having such low scores for all object can mean that the track has ended. In this case, that track should not be matched with anything. Additionally, when a new untracked object appears, there is typically no track that can be associated to it, so the association scores in the column of that object are low. In short, object tracks may end (no detection) or start (no existing track) frame by frame. To solve this problem, the table of matching scores must be augmented by extra nodes with a certain threshold score to absorb false matches. This way, no track will be associated with an object that has a score beneath the threshold. In Fig. 149, one node with the threshold of 0.30 is introduced.

	1	2	3	4
1	0.95	0.76	0.62	0.41
2	0.23	0.46	0.79	0.94
3	0.01	0.02	0.06	0.01
4	0.49	0.82	0.74	0.41

Score Matrix

	1	2	3	4	
1	0.95	0.76	0.62	0.41	0.30
2	0.23	0.46	0.79	0.94	0.30
3	0.01	0.02	0.06	0.01	0.30
4	0.49	0.82	0.74	0.41	0.30
	0.30	0.30	0.30	0.30	0.30

Augmented Matrix

Figure 149: **Augmentation of score matrix.** The red entries are the solutions of the greedy algorithm. The augmented matrix (right) features a new row and column (node) with fixed scores in blue.

11.3.6 Graph-based Tracking

Multi-frame tracking as Graph Optimization. We have seen how the association problem can be solved for two frames. This is not necessarily optimal for longer sequences. In particular, wrong associations cannot be undone at later frames. So, Multi-Frame Tracking must be considered. The setting is the same as before. The tracking-by-detection paradigm is used. The goal is to solve for all associations across all frames of a sequence. We can rephrase the problem so existing graph algorithms can be applied. Specifically, the problem can be cast as a min-cost flow network problem [**?**] as in Fig. 150.

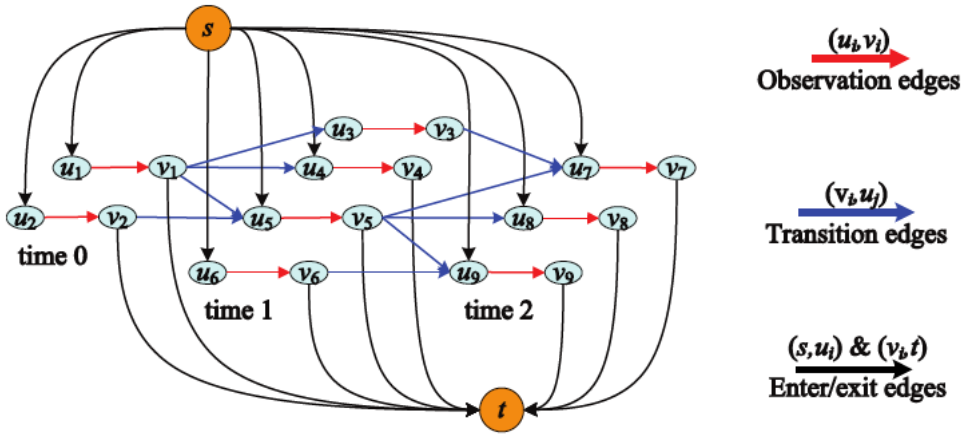


Figure 150: **Min-cost flow graph.** s is the source node any flow starts from. They end at sink t. Multiple observations with node pairs (u_i, v_i) are shown over multiple time steps. The graph is showing few observations and transitions for clarity [**?**].

The new goal is to push so called flow from the source s to the sink t such that it generates minimal cost. Observation edges carry negative cost proportional to their detection confidence. Transition edges carry positive cost to separate dissimilar objects. A high transition cost from a to b means that b should most likely not be associated with the track of a. Each detection i creates two nodes u_i and v_i in the graph. The first node u_i of a detection is connected to the source. The second node is connected to the sink. Flow is encouraged by the negative cost of observation edges to pass through highly confident observations. From observation to observation, the transition is taken that has the lowest cost and does not overlap with existing flows. The

mathematical formulation is given like this:

$$\underset{f}{\text{argmin}} \sum_i c_i^{en} f_i^{en} + \sum_i c_i^{ex} f_i^{ex} + \sum_{i,j} c_{i,j}^{li} f_{i,j}^{li} + \sum_i c_i^{det} f_i^{det} \tag{155}$$

$$s.t.\ f_i^{en} + \sum_{j,} f_{j,i}^{li} = f_i^{det} = f_i^{ex} + \sum_j f_{i,j}^{li}\ \forall i \tag{156}$$

$f_i \in \{0, 1\}$ are binary flow variables. They indicate whether a flow is passing through the entry (en) from source to u_i, the exit (ex) from v_i to sink, the i-th detection or the i, j-th link. $c_i \in \mathbb{R}$ represents the cost for entry, exit, link transition or a detection. Entry costs are costs for starting a track, exit costs for ending a track. A set f of f_i must be found that minimizes the cost.
The constraint basically means that different flows don´t overlap. A flow that is entering a node i from the source plus all the flow that is coming in from previous nodes must be the same as the flow that is going through the detection i. Remember that this is either 0 or 1. This must be the same as the flow that exits detection i to the sink plus all the flow that is going through the outgoing transitions. And that is for all nodes i. So, if flow passes through a detection, it must come either from the sink or from one incoming transition and vice versa for exiting the detection. Hence, no two flows can pass through the same detection so no object is assigned to multiple object tracks and no object track can be assigned to multiple objects as the flow cannot split.

Results on KITTI. The min-cost flow problem can also be solved by the **successive shortest path algorithm**. It has been found that this is the fastest algorithm for this problem. But only pairwise relationships can be modeled, since there are only pairwise edges between adjacent time steps. So, occlusions can´t be modeled. To model occlusions, detections over multiple frames would have to be linked, which becomes computationally expensive. As a result, only simplistic motion models can be used, where the object is not moving far. Additionally, solving the min-cost flow problem requires batch processing (Section 11.1.2) because the algorithm operates on the whole graph (all detections). Though, online versions have been proposed [**?**].

11.3.7 Advanced Graph-based Models

Previously, the object detectors were trained independent from solving the multi-object tracking or min-cost flow optimization problem. The detector is trained with metric learning (Section 11.3.2). Association and tracking is trained with given detections. But ideally, everything is trained jointly and end-to-end. It was found that this is in fact possible [**?**].

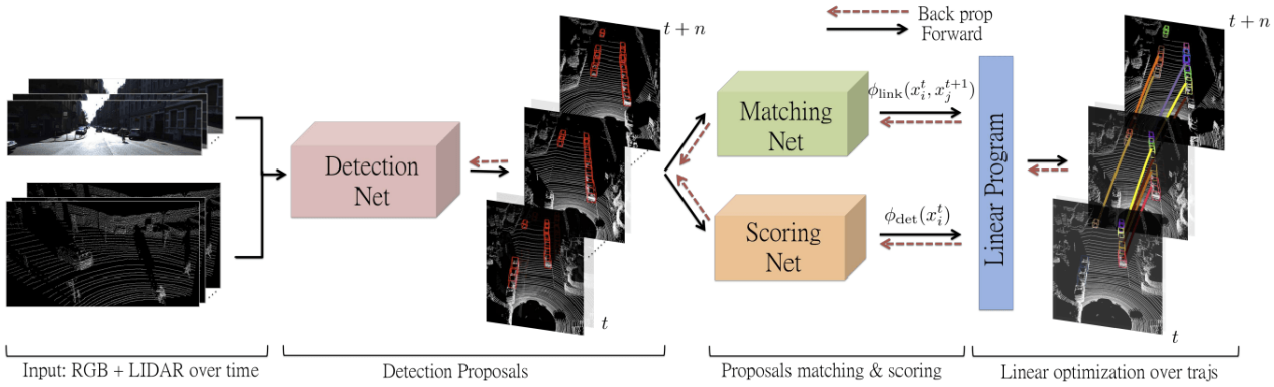


Figure 151: **Differentiable Graph Optimization.** The detection, matching and scoring net are part of one big network that learns end-to-end [**?**].

In Fig. 151, the input is propagated through a detection net, a matching net and scoring net on the same level. Lastly, it passes through a linear program. Detection and tracking are trained jointly by backpropagating through the linear program.

Another approach uses the fact that we basically have a graph based problem. It is depicted in Fig. 152. The input is encoded into features using convolutional neural networks. These feature encodings are then processed with neural message passing on that graph. As a result, the remaining problem is simply edge classification where edges that are associated to the same track are classified positively. This network is called graph neural network and learns end-to-end, but it´s using batch processing [**?**]. With today's frameworks, this method is simple and straightforward to implement and easy to train because it is all end-to-end differentiable.

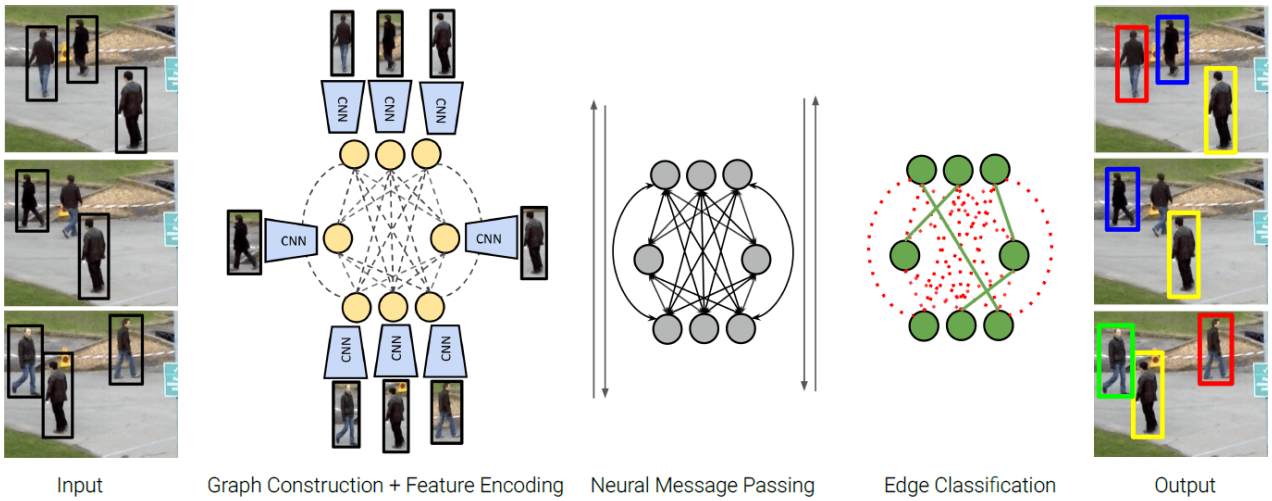


Figure 152: **Learning a neural solver.** The input objects are arranged in a graph and features are extracted from them. The features are then processed using neural message passing which turns the problem into simple edge classification. The algorithm uses batch processing [**?**].

11.3.8 Multi-Object Tracking Evaluation

For learning or comparison, the performance of Multi-Object Tracking algorithms must be evaluated. To do that, the predicted tracks must be associated with ground truth tracks using bipartite graph matching.

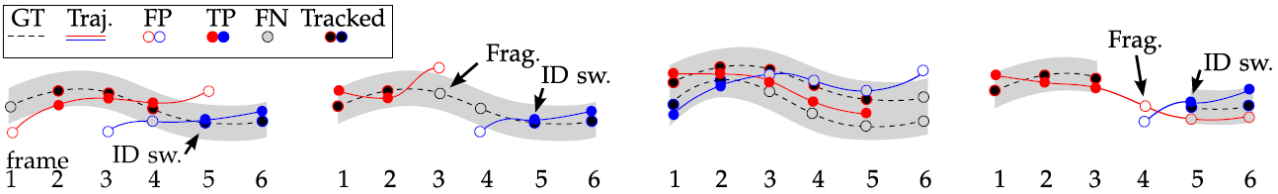


Figure 153: **Four common object tracking errors.** Each circle corresponds to a frame at time t. The dashed line is the Ground Truth (GT). Predictions in the grey area are accepted as True Positives (TP). The object tracks or trajectories are shown in red and blue. Empty circles are False Positives (FP). Grey circles are False Negatives (FN). Black filled circles are correctly tracked objects. ID sw. marks where an ID switch happened. A different object track is assigned to GT [**?**].

Fig. 153 shows different scenarios that have to be considered when associating predictions with ground truth tracks. These scenarios include matching multiple object tracks with one ground truth track (ID switches, IDSW), having false positives (FP) and false negatives (FN) and having ground truth tracks that are so close together that predicted object tracks may switch associations.

The most prominent metric to evaluate tracking is called MOTA - Multiple Object Tracking Accuracy, seen below:

$$MOTA = 1 - \frac{\sum_t FN_t + FP_t + IDSW_t}{\sum_t GT_t} \tag{157}$$

It produces a number which is 1 minus a sum of mistakes that were made normalized by the length of the ground truth track. The mistakes are sketched in Fig. 153. They include false negatives, false positives and ID switches. However, this metric has multiple shortcomings. A more refined alternative is HOTA - A Higher Order Metric for Evaluating Multi-object Tracking.

State of the art algorithms are evaluated on official benchmarks where participants can submit their results and get ranked. The KITTI benchmark is an example such an evaluation tool.

11.4 Holistic Scene Understanding

Holistic Scene Understanding means incorporating tracking in the wider context of tasks that must be solved in self-driving perception systems.

For instance, Lane Detection alone may be sufficient in an highway setting but lacks information on, e.g., intersections. Ideally, all cues in the scene must be fused to obtain a complete or holistic understanding of the situation. Fusing multiple cues, as a human would do, leads to more robust estimates. This is shown in the following figure.

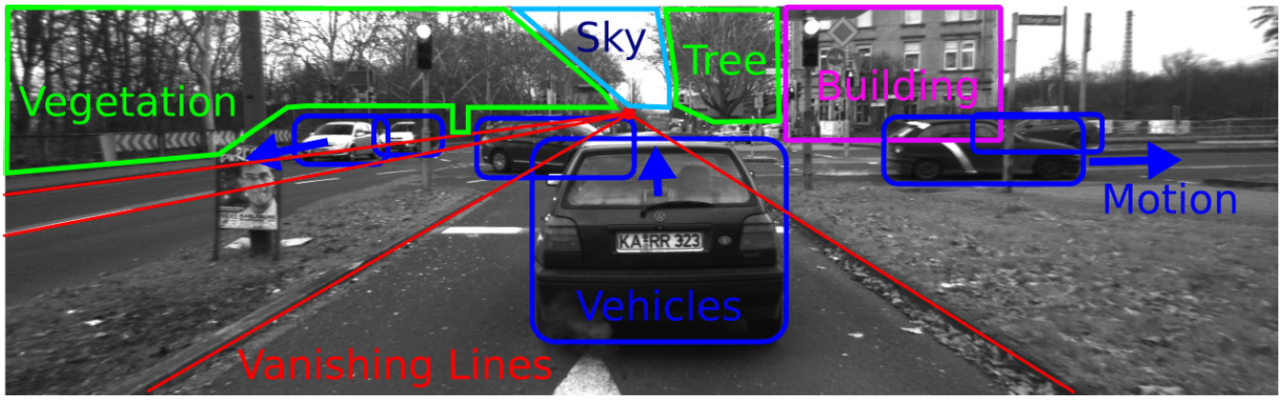


Figure 154: **Holistic Scene Understanding of an intersection from a Human perspective.**

Sensor fusion. Cues can be fused by, e.g., sensor fusion. Suppose an object detection is given by two sensors, e.g., camera and lidar. Let $\mathbf{x} \in \mathbb{R}^2$ denote the (unknown) object location. Let $\mathbf{z}_1 \in \mathbb{R}^2$ and $\mathbf{z}_2 \in \mathbb{R}^2$ be the noisy measurements of the camera and lidar. The **posterior probability distribution** over the object location is given by the bayes rule for multiple random variables (Eq. (141)):

$$p(\mathbf{x}|y_{1:t}) = \frac{p(\mathbf{z}_1, \mathbf{z}_2|\mathbf{x})p(\mathbf{x})}{p(\mathbf{z}_1, \mathbf{z}_2)} \propto p(\mathbf{z}_1|\mathbf{x})p(\mathbf{z}_2|\mathbf{x})p(\mathbf{x})$$

Here, it is assumed that $\mathbf{z}_i$ are independent and the proportional sign absorbs the constant $p(\mathbf{z}_1, \mathbf{z}_2)$. Assuming a uniform (uninformative) prior, which is $p(\mathbf{x}) = 1$, we obtain:

$$p(\mathbf{x}|\mathbf{z}_1, \mathbf{z}_2) \propto p(\mathbf{z}_1|\mathbf{x})p(\mathbf{z}_2|\mathbf{x}) \tag{158}$$

Assuming an uninformative prior means there is no knowledge about the prior location of our object.

Now that there is a way of combining data from different sensors, let´s go back to building a 3D traffic scene understanding. In the following approach, the goal is to infer scene understanding from a short stereo video sequence, without maps or lidar. The understanding includes **topology** and **geometry** of the scene and **semantic information**, e.g., the traffic situation. This is done by using a probabilistic generative model of urban scenes which fuses many different cues [**?**].

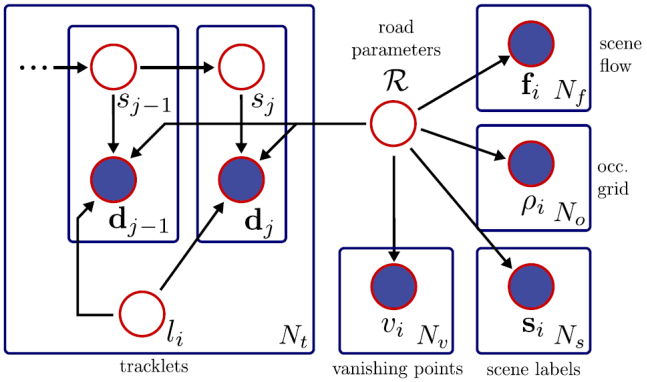


Figure 155: **Bayesian network** containing road parameters like scene flow, occupancy grid, scene labels and vanishing points. It also contains tracklets [**?**].

In the paper, this is formulated in a complex probabilistic graphical model, which is illustrated in Fig. 155 at the high level. This is a Bayesian network and it comprises observations of the scene flow, the occupancy, some semantic labels and vanishing points. Moreover, it comprises **tracklets**, which are comparable to a hidden Markov model that is inside the Bayesian network. In this approach, it is attempted to predict the state of each individual vehicle wrt. the jointly inferred scene understanding. So in short, the problem is solved by using inference techniques in graphical models.

Experimental results. Given the stereo input, calculating the distances is very hard, but by observing how the vehicles move, and by putting them into context with a certain traffic situation, the uncertainty of the measurements can be narrowed down. That´s the advantage of holistic modeling.

PnPNet: Perception and Prediction with Tracking in the Loop. Another way of integrating additional information into the tracking problem is to use maps. The idea is to use HD maps in combination with lidar to support the object detector and the tracker and to do joint perception tracking and trajectory estimation. Once you have tracked objects and you have a good motion model, then it becomes much easier to predict the motion of an object. That´s directly relevant to the behaviour modeling stage that follows the prediction stage and the planner stage that must execute the behaviour plans that have been determined [**?**].

Argoverse: 3D Tracking and Forecasting With Rich Maps. There is an additional tool for evaluating these algorithms besides KITTI benchmark. Argoverse is a recent data set for 3D tracking and forecasting with rich HD maps that is publicly released for autonomous driving. The data set contains sequences of lidar measurements and RGB video as well as accurate localization. This data set is nowadays frequently used for evaluating tracking and motion prediction techniques and it also maintains a leaderboard [**?**].

12 Decision Making and Planning

The following sections focus on the last component of the modular pipeline, namely *Decision Making and Planning*. The lecture is structured into four units. First we will introduce the decision making and planning in the subsection 12.1, and where it is located in the self-driving pipeline. Following, we will look at the three planning problems. In subsection 12.2 we describe route planning where a system needs to find a route, like a GPS systems are doing, then in the subsection 12.3 we show how different situations can be decomposed into, typically, graph-based representations. Finally, in 12.4 we describe motion planning, where the goal is to plan vehicle trajectory which is then fed to controller for execution.

12.1 Introduction

The goal of the decision making and path planning module is to find and follow path from current location to destination. It is supposed to take infrastructure and dynamic objects into account. The input for this stage is the vehicle and environment state, as perceived through the perception stack. The output is then a path or trajectory as input to vehicle controller. As it turns out, a number of challenges can arise in the driving situations, thus impose difficulties when modeled as a single optimization problem. For example picture an intersection and a vehicle coming into it. What are the possible situations and actions that a car should take? Some of them may include, stay stopped - if the traffic lights are red, decelerate to stop, stop because of a yield sign, emergency stop, go forward etc. As it can be seen, there is a lot of possibilities even without including actions of other traffic participants.

Therefore, the idea is to not model the entire block of decision making stack, but to break planning problem into a *hierarchy of simpler* problems, as shown in the figure 156. Each problem can than be tailored to its scope and level of abstraction. Earlier in the hierarchy represents higher level of abstraction. Decomposed in this way, every optimization problem will have constraints and objective functions.

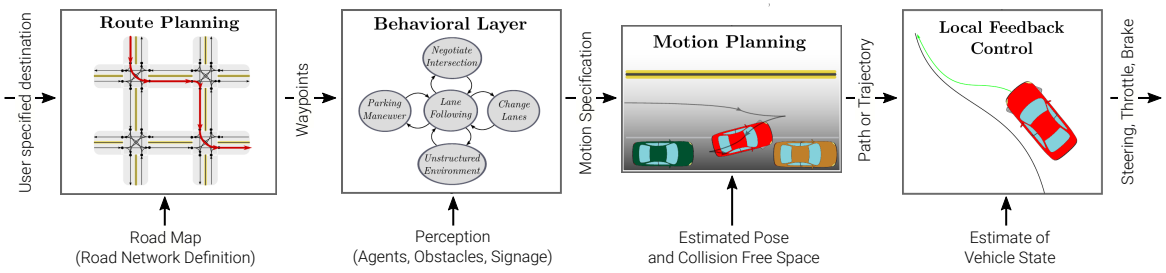


Figure 156: **Planning module decomposition**

When a destination is passed to a route planer, it generates a route through the road network obtained from the map. Following, a behavioral layer then reasons about the environment and generates a motion specification such that the vehicle progress along the selected route. Motion planer then finds a feasible motion accomplishing the specification provided by the behavioral layer. Finally, a feedback control adjusts actuation variables to correct errors in the execution of the reference path.

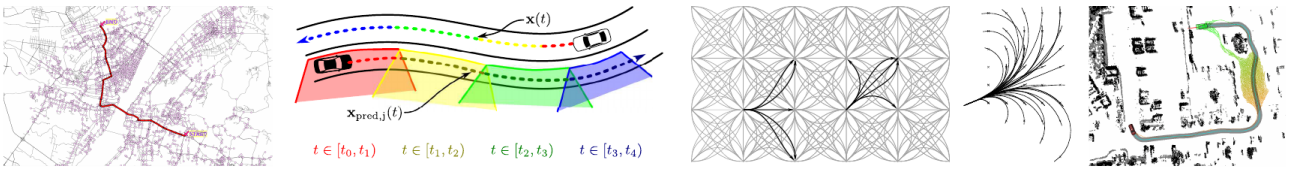
Route Planning is the first step in the hierarchy. Here we represent the road network as **directed graphs**, that comprises nodes and directed edges. In such graphs, edges weights correspond to road segment length or travel time. When we have such representations, the problem is translated into a *minimum-cost* graph network problem. We can solve such problems by running inference algorithms, e.g. Dijkstra's or A*.

Behavior Planning selects *driving behavior* based on the current vehicle and environment state, e.g. at stop line: observe the traffic, stop, traverse etc. This layer of abstraction is often modeled via **finite state machines**, that model transitions governed by perception. They are deterministic, but can also be modeled probabilistically, e.g. using Markov Decision Processes.

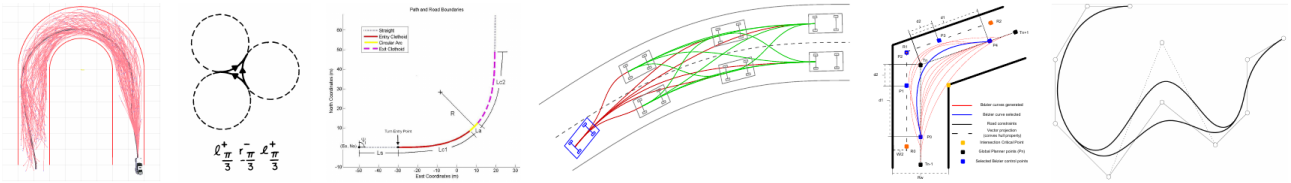
Motion Planning is the responsible for finding a feasible, comfortable, safe and fast vehicle trajectory. In this process much difficulties can arise, since exact solutions are in most cases computationally intractable. Often **numerical approximations** are employed to overcome the computational intractability. Some of the methods used are: variational methods, graph search and incremental tree-based methods.

Local Feedback Control finally execute the path/trajectory from the motion planner. This controller also corrects errors due to inaccuracies of the vehicle model. It is particularly important that this component is robust and stable, and functions such that the driving is comfortable. This type of control was discussed in the lectures 5 and 6.

Some of the planning methods use in literature are depicted in the figure 157.



(a) Dijkstra [29] (b) FunctionOptimization [38] (c) Lattices [39] (d) A* [36]



(e) RRT [40] (f) Line&Circle [41] (g) Clothoid [42] (h) Polynomial [43] (i) Bezier [44] (j) Spline [45]

Figure 157: **Overview** on some of the planning algorithms.

12.2 Route Planning

The goal of the *route planning* is to navigate from point A to point B on a road map, using the *most efficient path*, defined in certain terms, e.g. time, distance. The route planning itself is a high-level planing, where low-level details are abstracted away. Sometimes it is also referred to as *mission planning*.

Road networks can be represented as graphs. The figure 158 illustrates such a road network on the example of a cross section. If we model a road network by a directed graph, different paths through the graph represent different paths we could take on a road. We can use such representations on both lane level and road level, depending on the needed granularity.

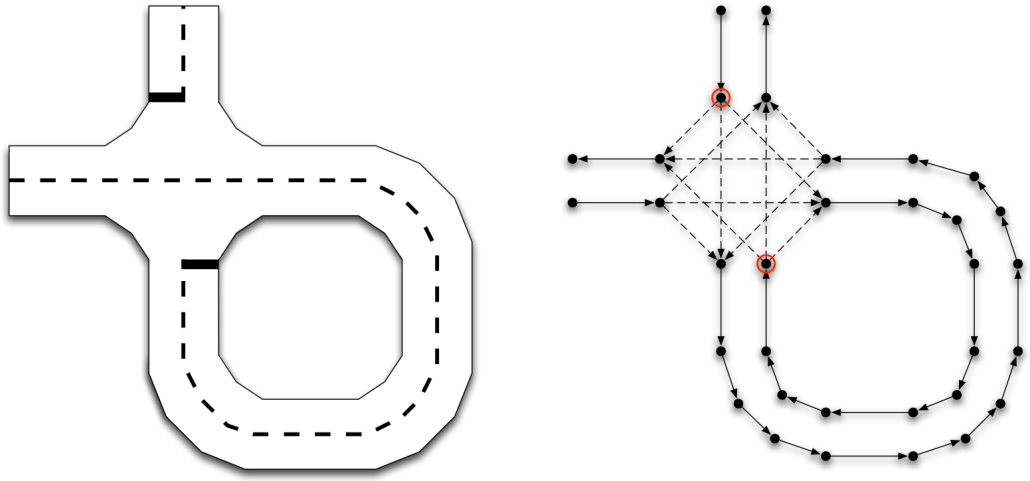


Figure 158: **Road Network** An illustration of a road network.

Generally in self-driving, we have HD maps that allows for using directed graphs on the lane level. Such graph is shown on figure 159. Formally, we define such a graph as follows: $G = (V, E)$ with vertices $v \in V$ and directed edges $(u, v) \in E$. Edges correspond to road or lane segments. In such graphs, s denotes the source from which we want to travel, and t denotes the sink - destination. As mentioned, we are interested in finding the *shortest path*. In the example graph, shortest path can be the least edges, whereas in a weighted graph that can be the path with the least sum of weights.

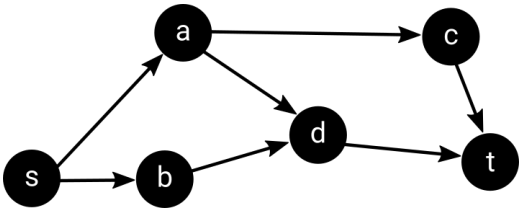


Figure 159: **Directed graph.**

12.2.1 Breadth First Search

One of the algorithms used for finding the shortest path is **Breadth First Search** (BFS). First we introduce a data structure that will enable such search.

Queue is a collection of sequential entities, that functions in a *First-in-first-out (FIFO)* manner. Two operations are defined: *enqueue* - addition at the end, and *dequeue* - removal from the front of the queue. This structure is often implemented via linked lists, in that way both operations have time complexity $O(1)$.

Breadth first search is defined as follows in the pythonic pseudo-code bellow. Queue *open* stores all the elements we need to work on, the variable *closed* is the set of the nodes that are already processed, the variable *predecessors* collects the lists needed for backtracking and finding the exact set of visited nodes along the shortest path.

Algorithm 1 BFS(G,s,t)

```
1:  open ← Queue(), closed ← Set(), predecessors ← Dict()
2:  open.enqueue(s)
3:  while !open.isEmpty() do
4:      u ← open.dequeue()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed and v ∉ open then
9:              open.enqueue(v)
10:             predecessors[v] ← u
11:     closed.add(u)
```

This algorithm works for unweighted graphs, and returns the optimal solution in $O(|V| + |E|)$. Here we give an explanation of the algorithm. We start from s and first enqueue it to *open* queue. Then we ask if first element u in the queue (in the first iteration that is node s), is equal to the sink node t, if it is, we found our node and we extract path using predecessors. If the element isn't equal to the sink, for each it's successor v we enqueue it into the open queue if it's not contained in neither *closed* or *open* queues, and set it's predecessor to u. After processing all u's successors we add u to the set of *closed* nodes. A virtual run of the algorithm can be seen here.

As said, the algorithm works for unweighted graphs, which restricts our representation. With such a restriction, we are not able to represent road segments of variable length. If we chunk road network into small segments of equal length, we can then run this algorithm, but with a potentially substantial increase in computation. One solution for this problem is given in the next subsection.

12.2.2 Dijkstra's Algorithm

In order to more reliably represent road networks, we can use **weighted graphs**. An edge in such directed graph carries a weight corresponding to road segment length or travel time. We can employ a better algorithm for finding the shortest paths on weighted graphs, called **Dijkstra's algorithm**.

Min Heap is a specialized tree-based data structure that is needed for this algorithm. It is a efficient implementation of priority queue. Min Heapit satisfies the so called *Min heap property*: The value of any node in the tree is smaller than all its children. Therefore, the smallest ellement is stored at root node. As a common implementation binary tree is used, and insertion and removal have time complexity of $O(\log n)$.

The easiest way to understand it, is to defined it as same as the BFS. Differences between the two are highlighted in red.

Algorithm 1 BFS(G,s,t)

```
1:  open ← Queue(), closed ← Set(), predecessors ← Dict()
2:  open.enqueue(s)
3:  while !open.isEmpty() do
4:      u ← open.dequeue()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed and v ∉ open then
9:              open.enqueue(v)
10:             predecessors[v] ← u
11:     closed.add(u)
```

Algorithm 2 Dijkstra(G,s,t)

```
1:  open ← MinHeap(), closed ← Set(), predecessors ← Dict()
2:  open.push(s,0)
3:  while !open.isEmpty() do
4:      u, uCost ← open.pop()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed then
9:              uvCost ← edgeCost(u,v)
10:             if v ∈ open then
11:                 if uCost + uvCost < open[v] then
12:                     open[v] ← uCost + uvCost
13:                     predecessors[v] ← u
14:             else
15:                 open.push(v,uCost+uvCost)
16:                 predecessors[v] ← u
17:     closed.add(u)
```

The algorithm returns optimal solution in $O(|V|\log|V|+|E|)$. Unlike in BFS, here we use Min Heap as the structure for *open* nodes, in order to efficiently obtain the first node that is to be processed. More precisely, together with node, in *open* minheap we collect node's distance from the sink, and use that as a number by which we rank the nodes saved in that minheap. As before, we select the first node u in our prioritized queue *open*, if it is the sink, we are done and we can extract the path. If not, we analyse all u's successors. For a successor v, if it is not a *closed* node, and if it is an *open* node, we ask if the cost of getting to node u plus the cost from u to v is less than the cost to get to node v, if yes, we set the cost of the node v to the mentioned sum, and update it's predecessor as node u. If the node v wasn't an open node, we push it to the *open* nodes heap, and set it's price to mentioned sum, and set predecessor to the parent node u. Intuitively, we ask ourselves: if the node v is not closed (already processed), and if it is open (still needs to be processed), is it better for us to come to that particular node v following the path coming through the node u, or to use the path previously found for the node v. A detailed explanation with an example algorithm run can be found here.

The problem is that Dijkstra's algorithm also doesn't scale to large graphs, as it becomes computationally very expensive and thus slow to run. For example, the map of New York City would comprise of 54,837 vertices and 140,497 edges. To check all the nodes in this graph would be very slow. We need some kind of guidance during our search, and one such guiding could be defined using the so called **planning heuristics**. As an example of the heuristic, we can use **Euclidean planning heuristics**. The heuristic exploits structure of planar graphs. Namely, straight lines between vertices in a graph are useful estimate of the distance along the path. In the example on figure 160 Dijkstra's algorithm would expand from s to a, where in contrast the heuristic-guided search (e.g. A* algorithm) would choose b as it is more likely to be on the shortest path since it is closer to the target t.

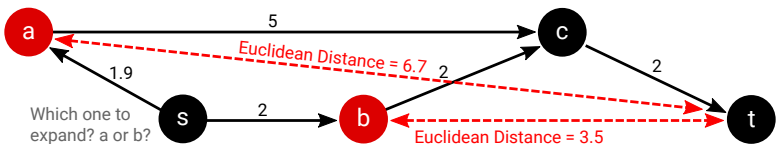


Figure 160: **Euclidean Heuristic** shown on an example graph

12.2.3 A* Algorithm

The **A* Algorithm** is a guided-search algorithm that uses planning heuristics to find a shortest path from the node A to the node B in a graph. It is shown that no matter which heuristic is used, it always returns the optimal solution. As for the past algorithm, comparison with the Dijksta's algorithm will be shown.

Algorithm 2 Dijkstra(G,s,t)

```
1:  open ← MinHeap(), closed ← Set(), predecessors ← Dict()
2:  open.push(s,0)
3:  while !open.isEmpty() do
4:      u, uCost ← open.pop()
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed then
9:              uvCost ← edgeCost(u,v)
10:             if v ∈ open then
11:                 if uCost + uvCost < open[v] then
12:                     open[v] ← uCost + uvCost
13:                     predecessors[v] ← u
14:             else
15:                 open.push(v,uCost + uvCost)
16:                 predecessors[v] ← u
17:     closed.add(u)
```

Algorithm 3 A*(G,s,t)

```
1:  open ← MinHeap(), closed ← Set(), predecessors ← Dict()
2:  open.push(s,0,h(s))
3:  while !open.isEmpty() do
4:      u, uC, uHeur ← open.pop() cost+heuristic
5:      if u = t then
6:          return extractPath(u,predecessors)
7:      for all v ∈ u.successors() do
8:          if v ∉ closed then
9:              uvC ← edgeCost(u,v)
10:             if v ∈ open then
11:                 if uC + uvC + h(v) < fullCost(v) then
12:                     open[v] ← uC + uvC, h(v)
13:                     predecessors[v] ← u
14:             else
15:                 open.push(v,uC + uvC, h(v))
16:                 predecessors[v] ← u
17:     closed.add(u)
```

The most important difference compared to Dijkstra's algorithm is that now when we are asking ourselves whether the path that we are on (coming from node u) into node v is better than the one we previously knew, we now take into account also the heurstic value of the node v. Particularly, for a node v that is a successor of the node u (u is popped from *open* minheap as in Dijkstra's), if it's not closed, and it is open, and if the cost of the node u plus the cost from u to v PLUS the heuristic value of v (could be a euclidean distance from v to target node t) is smaller than the full cost of the node v (heuristic + actual cost to get to that node), we update the cost of node v as $uCost + uvCost$, and we set heuristic value as $h(v)$. If the node v is not open, we push it the *open* minheap and set it's predecessor to the parent node u. Both A* and Dijkstra's algorithm find an optimal solution, it is very likely that A* would find that solution faster. The simulation of the algorithm run can be found here. An example search from a publicly available software https://github.com/kevinwang1975/PathFinder is shown as an example in figure 161. It can clearly be seen how the usage of heuristics can guide the search towards the target, and not spend much time exploring the space that's not leading towards an optimal solution.

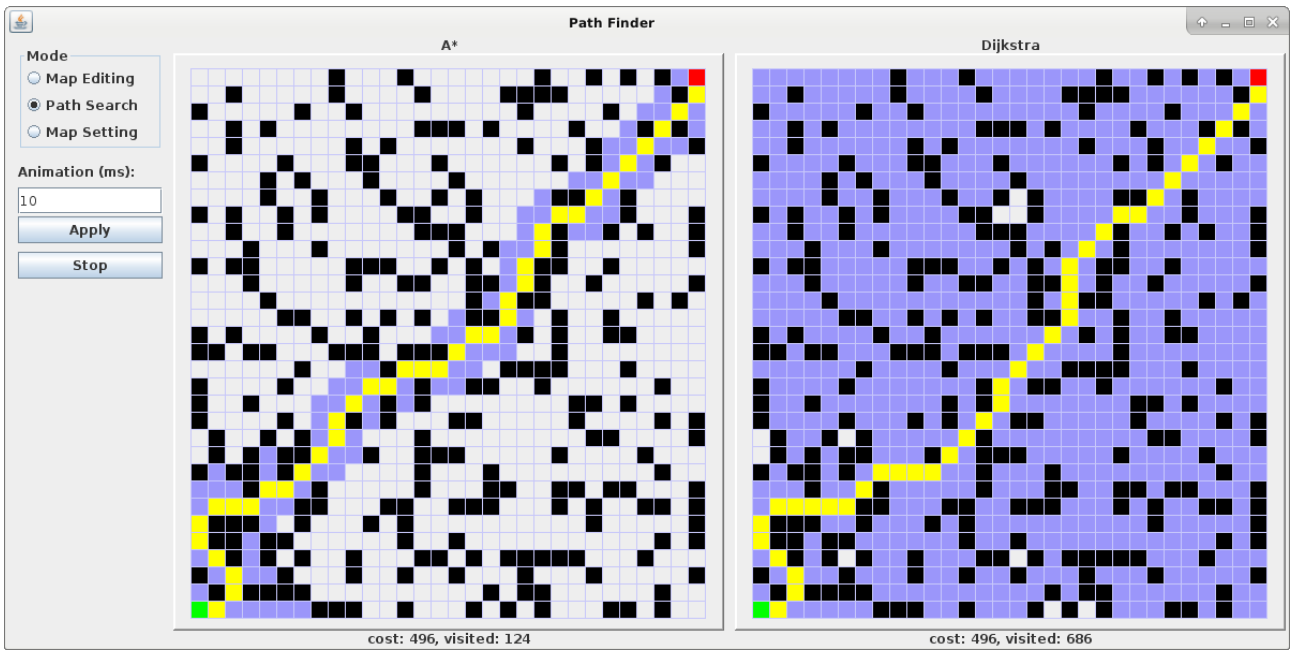


Figure 161: **Path Finder** A* vs Dijkstra's algorithm

12.3 Behavior Planning

Given a planned route we need to follow that path. That task is then splitted into two, behavior planning and motion planning.

To follow a planned route, the vehicle must conduct various maneuvers as mentioned, e.g. speed tracking, car following, stopping etc. It is very difficult to design and execute a motion planner for all maneuvers jointly. Thus, the behavior planning stage discretizes the behaviors into simpler (atomic) maneuvers. Now each of the atomic behaviors can be addressed with a dedicated motion planner. Input for this planner is a high-level route plan and the output of perception stack. As output, the planner computes constraints for motion planner, i.e. corridor, objects, speed limits, target etc. Some of the frequently used models include:

- Deterministic: Finite State Machines (FSMs) and variants
- Probabilistic: Markov Decision Processes (MDPs) and Partially Observable Markov Decision Processes (POMDPs)

Finite State Machine (figure 162) is defined by quintuple $(\Sigma, \mathcal{S}, \mathcal{F}, s_0, \delta)$

- Σ is the input alphabet
- $\mathcal{S}$ is a non-empty set of states
- $\mathcal{F} \subseteq \mathcal{S}$ is the (possibly empty) set of final states
- $s_0 \in \mathcal{S}$ is the initial state
- $\delta : \mathcal{S} \times \Sigma \to \mathcal{S}$ is the state transition function

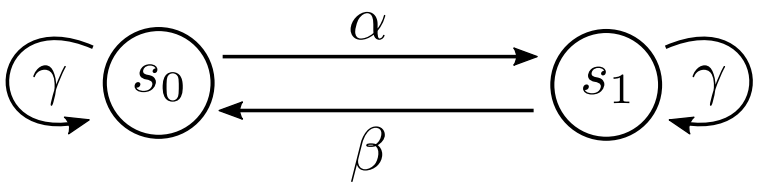


Figure 162: **Finite State Machine** An example.

Applied on the problem of self-driving cars, a FSM for a simple vehicle behavior could me modeled as in figure Fig. 163.

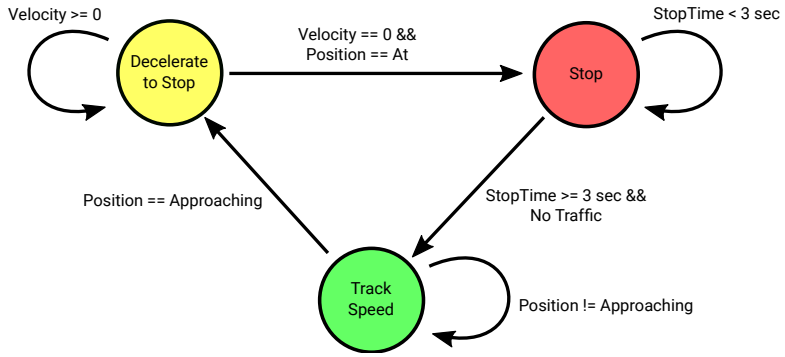


Figure 163: **Finite State Machine** A self-driving example.

If the vehicle is in the state Track Speed, and the position is equal to approaching (an intersection or a stop sign) we change the state to Decelerate to Stop, if the position is not approaching, then we continue to track speed. If in the state Decelerate to Stop, the speed is larger or equal to zero, the state should stay the same, until we don't approach the full stop. If we approach the zero velocity, and we approach a Stop, we should hang there until our StopTime is larger than 3 sec, and it there is no traffic, we should switch to state Track Speed.

More advanced FSM from the Darpa Challenge is shown in the figure 164.

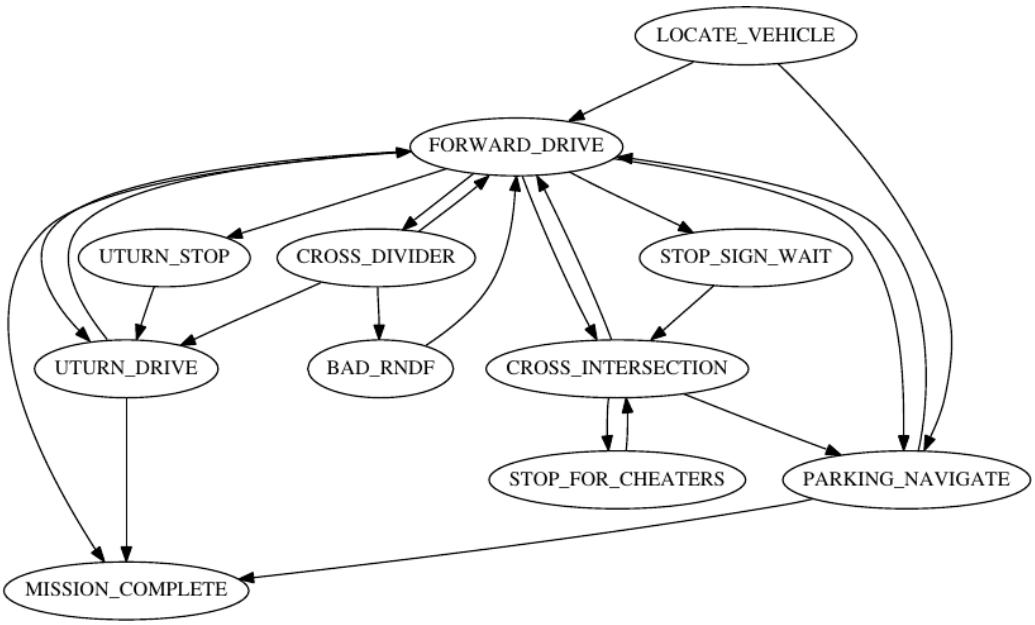


Figure 164: **Finite State Machine** Stanford Junior's FSM.

In order to handle multiple scenarios, like intersections, highway driving etc., finite state machines can be extended to **Hierarchical State Machine (HSM)**. Each scenario can then have a *meta-state*, and we can transition between those meta-states. Furthermore, inside each of them, we can have states that can also be transitioned between. As an advantage, we see that HSMs are more simpler to describe, and more efficient, but as a disadvantage some of the rules within the meta-states can duplicate. Figure 165 depicts one such HSM.

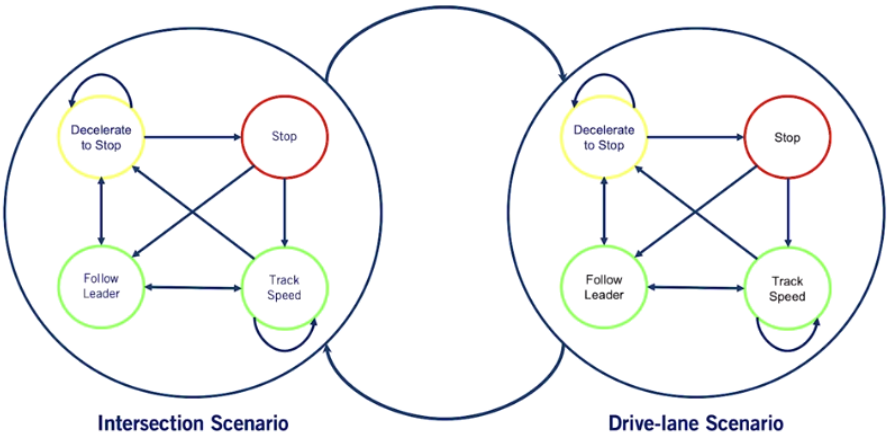


Figure 165: **Hierarchical State Machine** An example.

12.4 Motion Planning

Motion planning is the most fine-grained stage of all the previous ones. The goal of motion planning is to compute safe, comfortable and feasible trajectory from the vehicle's current configuration to the goal based on the output of the behavioral layer. This planning is often done locally, e.g. few meters ahead. Takes as input static and dynamic obstacles around vehicle and generates collision-free trajectory. There are two types of output representations:

- Path: $\sigma(l) : [0, 1] \rightarrow \mathcal{X}$ (does not specify velocity)
- Trajectory: $\pi(t) : [0, T] \rightarrow \mathcal{X}$ (explicitly considers time)
- ... with $\mathcal{X}$ the configuration space of the vehicle and T the planning horizon

Main formulations of these problems are: Variational Methods, Graph Search Methods and Incremental Search Techniques.

12.4.1 Variational Methods

Variational methods **minimize a functional**:

$$\begin{aligned} \underset{\pi}{\text{argmin}}\ J(\pi) &= \int_0^T f(\pi)\ dt \\ \text{s.t.} \pi(0) &= \mathbf{x}_{init} \quad \wedge \quad \pi(T) \in \mathbf{X}_{goal} \end{aligned}$$

The functional $f(\cdot)$ integrates soft constraints, e.g. spatial, verolicty, jerk, etc. Additional hard constraints can be formulated, such as minimun turn radius etc. Most of this kind of problems are solved using numerical optimization. Since it is often non-linear problem it converges to a local minimum.

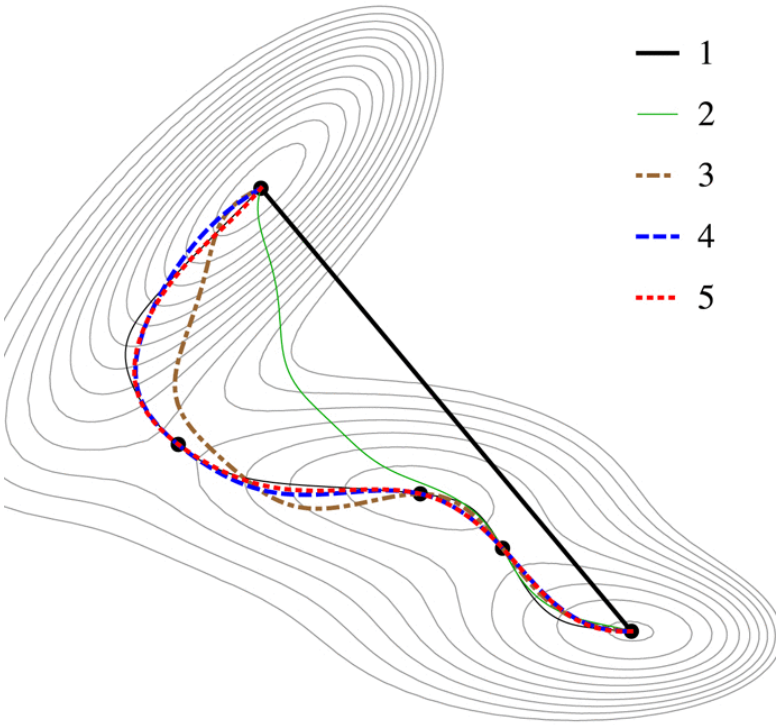


Figure 166: **Variational Optimization** An example.

12.4.2 Graph Search Methods

An alternative to variational methods is to use **graph search methods**, such as discussed before Dijkstra, A*, etc. The idea is to discretize configuration space $\mathcal{X}$ into graph $G = (V, E)$. To obtain a graph representation, we can employ various algorithms for constructing graphs. Thus, the problem somes down to a search based, which we are familiar with.

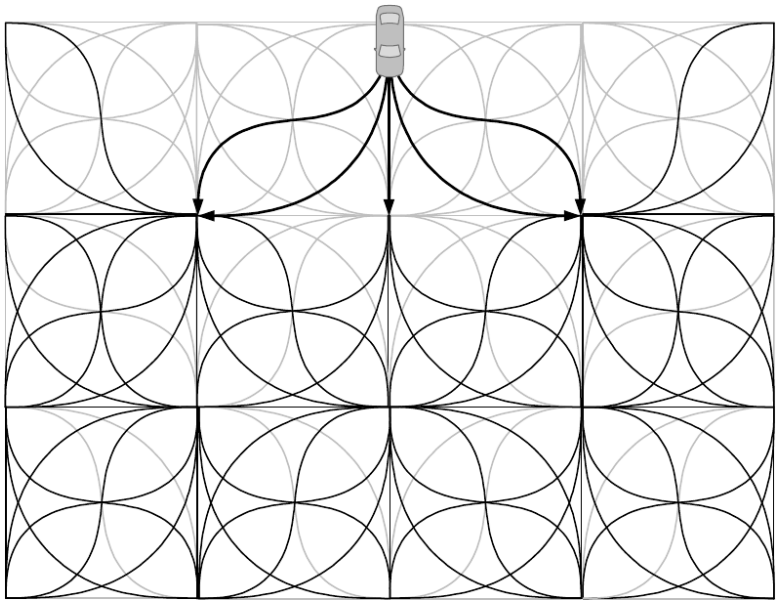


Figure 167: **Graph Search** An example.

12.4.3 Incremental Search Techniques

The idea behind **incremental search methods** is to incrementally build increasingly finer discretization of configuration space $\mathcal{X}$. These methods are guaranteed to provide feasible path given enough computation time. However, computation time can be unbounded. Prominent example is: Rapidly exploring random trees (RRTs).

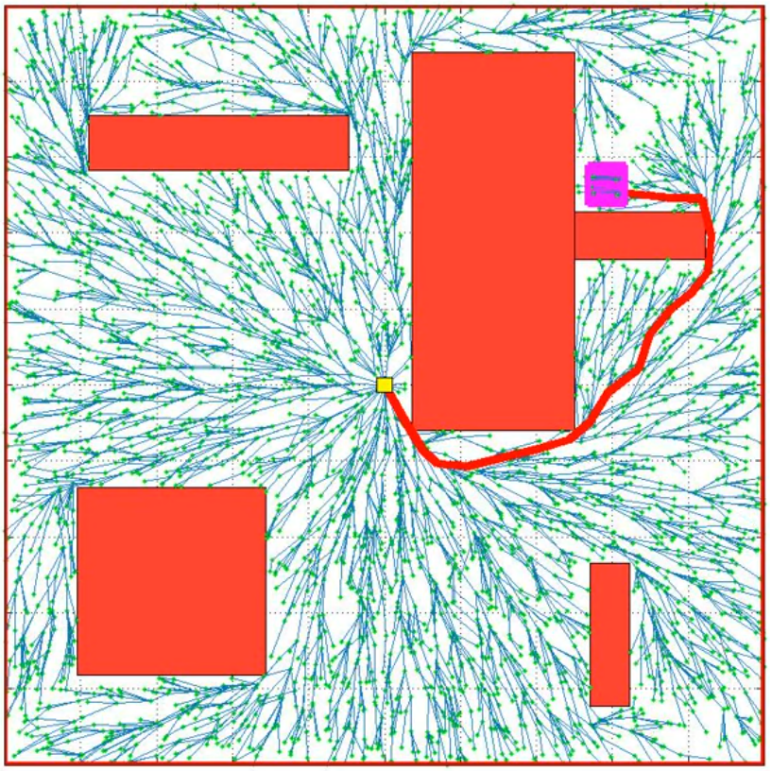


Figure 168: **Incremental Search Technique** Rapidly exploring random trees.

12.4.4 Hybrid A* Path Planning

Hybrid A* is an A* variant that guarantees kinematic feasibility of the path. It was used in Stanley vehicle. The demonstration of the working system can be seen here.

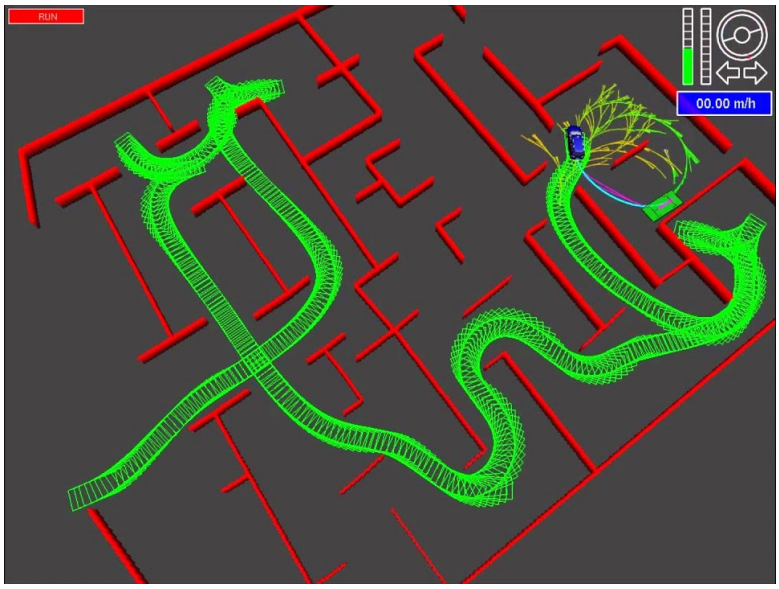


Figure 169: **Hybrid A* Algorithm**

12.5 Summary

Short summary of the lecture points:

- Driving situations and behavior are very complex, thus hard to model
- We overcome the complexity by breaking the problem into hierarchy of simpler problems: route planning, behavior planning and motion planning
- Each problem is then tailored to its scope and level of abstraction
- Road networks can be represented as weighted directed graphs, thus we can process them with inference algorithms and graph based search techniques
- Dijkstra's algorithm finds the shortest path in such a graph
- A* exploits planning heuristics to improve efficiency
- Behavior planning can be implemented using finite state machines
- Motion planning is often solved using variational and graph search methods

Here we show an overview on different used methods. There is still no consensus on which algorithm works best, so the method is often chosen as problem dependent.

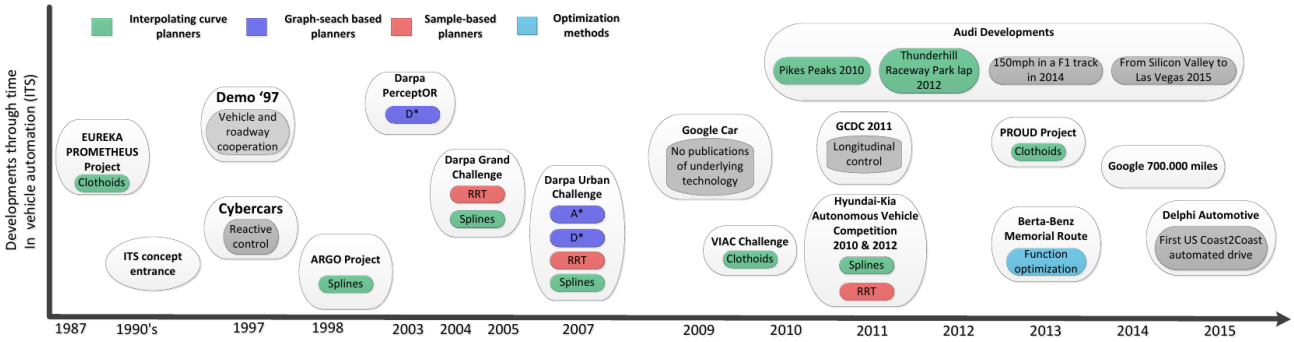


Figure 170: **Algorithms used in self-driving**

References

[1] H. Badino, U. Franke, and R. Mester. Free space computation using stochastic occupancy grids and dynamic programming. In *Proc. of the IEEE International Conf. on Computer Vision (ICCV) Workshops*, 2007.

[2] Hernan Badino, Uwe Franke, and David Pfeiffer. The stixel world - a compact medium level representation of the 3d-world. In *Proc. of the DAGM Symposium on Pattern Recognition (DAGM)*, 2009.

[3] Hernán Badino, Daniel F. Huber, and Takeo Kanade. Real-time topometric localization. In *IEEE International Conference on Robotics and Automation, ICRA 2012, 14-18 May, 2012, St. Paul, Minnesota, USA*, pages 1635–1642. IEEE, 2012.

[4] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI)*, 39(12):2481–2495, 2017.

[5] Aseem Behl, Kashyap Chitta, Aditya Prakash, Eshed Ohn-Bar, and Andreas Geiger. Label efficient visual abstractions for autonomous driving. In *Proc. IEEE International Conf. on Intelligent Robots and Systems (IROS)*, 2020.

[6] Mariusz Bojarski, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Larry D. Jackel, Urs Muller, and Karol Zieba. Visualbackprop: visualizing cnns for autonomous driving. *CoRR*, abs/1611.05418, 2016.

[7] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *arXiv.org*, 1604.07316, 2016.

[8] Chenyi Chen, Ari Seff, Alain L. Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proc. of the IEEE International Conf. on Computer Vision (ICCV)*, pages 2722–2730, 2015.

[9] Kashyap Chitta, Aditya Prakash, and Andreas Geiger. Neat: Neural attention fields for end-to-end autonomous driving. In *International Conference on Computer Vision (ICCV)*, 2021.

[10] Laura A. Clemente, Andrew J. Davison, Ian D. Reid, José Neira, and Juan D. Tardós. Mapping large loops with a single hand-held camera. In *IN PROC. ROBOTICS: SCI. SYST*, 2007.

[11] Felipe Codevilla, Antonio M. Lopez, Vladlen Koltun, and Alexey Dosovitskiy. On offline evaluation of vision-based driving models. In *Proc. of the European Conf. on Computer Vision (ECCV)*, 2018.

[12] Felipe Codevilla, Matthias Miiller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *Proc. IEEE International Conf. on Robotics and Automation (ICRA)*, 2018.

[13] Felipe Codevilla, Eder Santana, Antonio M. López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. In *Proc. of the IEEE International Conf. on Computer Vision (ICCV)*, 2019.

[14] Jinggang Huang, Ann B. Lee, and David Mumford. Statistics of range images. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2000.

[15] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. *arXiv.org*, 1612.01925, 2016.

[16] Christian Kerl, Jürgen Sturm, and Daniel Cremers. Dense visual SLAM for RGB-D cameras. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*, pages 2100–2106. IEEE, 2013.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[18] Abhijit Kundu, Yin Li, Frank Dellaert, Fuxin Li, and JamesM. Rehg. Joint semantic segmentation and 3d reconstruction from monocular video. In *Proc. of the European Conf. on Computer Vision (ECCV)*, 2014.

[19] Nikolaus S. Lang, Michael Rüßmann, Jeffrey Chua, and Xanthi Doubara. Making autonomous vehicels a reality. 2017.

[20] Michael Montemerlo, Jan Becker, and Sebastian Thrun. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics (JFR)*, 25(9):569–597, September 2008.

[21] Matthias Müller, Alexey Dosovitskiy, Bernard Ghanem, and Vladlen Koltun. Driving policy transfer via modularity and abstraction. In *Proc. Conf. on Robot Learning (CoRL)*, 2018.

[22] Dean Pomerleau. ALVINN: an autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems (NeurIPS)*, 1988.

[23] Aditya Prakash, Aseem Behl, Eshed Ohn-Bar, Kashyap Chitta, and Andreas Geiger. Exploring data aggregation in policy learning for vision-based urban autonomous driving. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[24] Axel Sauer, Nikolay Savinov, and Andreas Geiger. Conditional affordance learning for driving in urban environments. In *Proc. Conf. on Robot Learning (CoRL)*, 2018.

[25] Deqing Sun, Stefan Roth, and Michael J. Black. Secrets of optical flow estimation and their principles. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2010.

[26] Urmson et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[27] S. Vedula, P. Rander, R. Collins, and T. Kanade. Three-dimensional scene flow. *IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI)*, 27(3):475–480, 2005.

[28] Brady Zhou, Philipp Krähenbühl, and Vladlen Koltun. Does computer vision matter for action? *Science Robotics*, 4(30), 2019.