Unit -1 AIAV

Autonomous Driving :

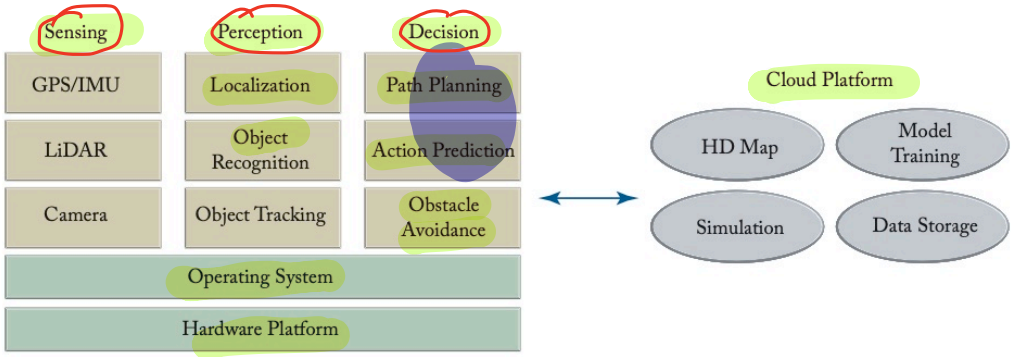


Figure 1.1: Autonomous driving system architecture overview.

→ sense → localization
→ percept → object
→ decision → obstacle.
path action

HD, Model, simulation, Data st.

Sensing

- **GPS/IMU:** These devices provide positional data and inertial measurements (like acceleration and orientation) crucial for determining the vehicle's exact location and movement.
- **LiDAR:** Uses laser beams to create a high-resolution 3D map of the vehicle's surroundings, essential for navigating and identifying obstacles.
- **Camera:** Captures visual data that aids in detecting road signs, lane markings, other vehicles, pedestrians, and more.

Perception

- **Localization:** Combines data from GPS, IMU, and sometimes other sensors to pinpoint the vehicle's position within a few centimeters on a map.
- **Object Recognition:** Analyzes sensor inputs to identify and classify objects in the vehicle's environment (e.g., cars, pedestrians).
- **Object Tracking:** Continuously monitors and predicts the paths of detected objects to avoid collisions and adjust the route dynamically.

Decision

- **Path Planning:** Determines the optimal path or route the vehicle should take to reach its destination safely and efficiently.
- **Action Prediction:** Forecasts the actions of other road users (like a car merging into the lane or a pedestrian crossing the street) to make informed driving decisions.
- **Obstacle Avoidance:** Identifies potential obstacles from the perception module's data and calculates maneuvers to avoid them while maintaining a safe and legal driving path.

Cloud Platform

- **HD Map:** Provides highly detailed maps that offer precise road details unlike conventional maps; essential for localization and navigation.
- **Model Training:** Uses collected data to train machine learning models that improve the system's decision-making capabilities.
- **Simulation:** Tests and validates the vehicle's software in virtual scenarios to ensure it performs as expected in real-world conditions.
- **Data Storage:** Stores vast amounts of data from vehicle sensors and operations, essential for analysis, model training, and refining system performance.

→ location & orientation
→ Global position
→ accurate
→ slow update

IMU
→ frequent update

localization

position of vehicle

Radar sonar
→ best min safety
→ in memory processing
→ rapid response

Perception :

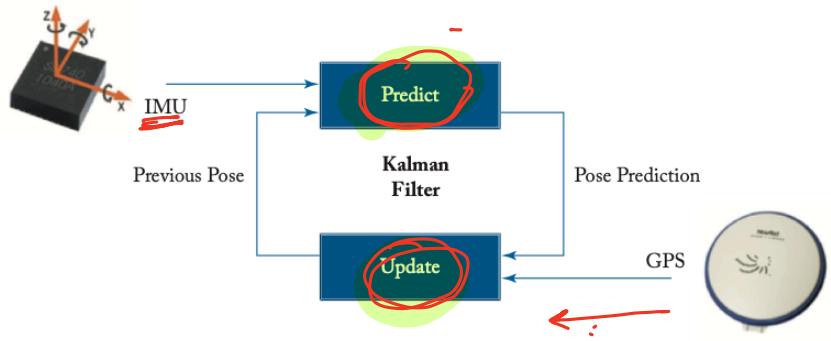
① Localization : precise position of vehicle.

i) GPS/IMU → Kalman Filter

slow accurate — fast errors accumulate over time

correction!

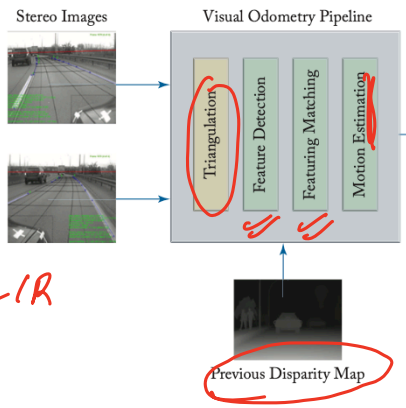
→ Tunnel localization



IMU predicts
GPS updates

Figure 1.2: GPS/IMU localization.

ii) vision based (Stereo localization) → 2D → 3D



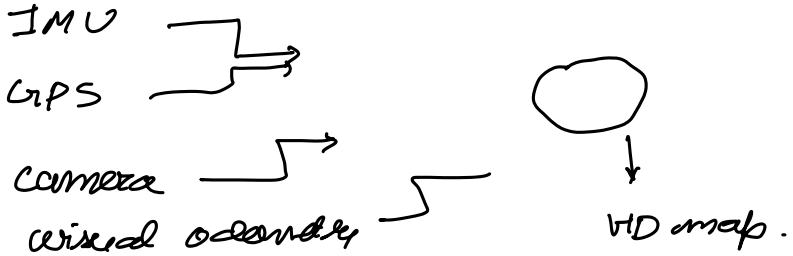
① → Δ (2D)
→ perceive depth (disparity map)
② → feature detection.
③ → feature matching (track)
→ estimate movement.
→ vehicle pose to correct
→ integration with HD map

LIDAR

iii) LIDAR
→ particles
→ high precision localization
→ Kalman Filter

10cm

Sensor fusion



② Object detection

→ camera & LiDAR

→ RADAR (relative speed & distance)

↳ immediate response.

CNN

- convolution
- activation
- Pooling
- F.C

autoencoders

③ Object Tracking

→ monitor detected objects

path planning, collision avoidance

Decision

→ Action prediction

→ Path planning

→ Obstacle avoidance

Localization :

- GNSS ✓
- LIDAR
- HD map
- Visual odometry
- Dead Reckoning sensors

find position BRO!!

① GNSS :

→ global navigation satellite system

GPS ⇄ (x) (y) (z)

→ precise
→ 4 satellites accuracy
→ 6 orbital planes

→ continuous availability

GPS

① satellite to GPS receivers. Biphase modulation
② navigation info
③ sent back calculated position...

Space fixed coordinate → no change
Earth fixed " → rotation axis & Greenwich

GNSS error Analysis :

1. satellite clock : 10 ns ⇒ 3m inaccuracy
2. orbit error : ± 2.5m
3. ionosphere delay : charged particles
4. troposphere : temperature & humidity
5. multipath : buildings – bouncing signals

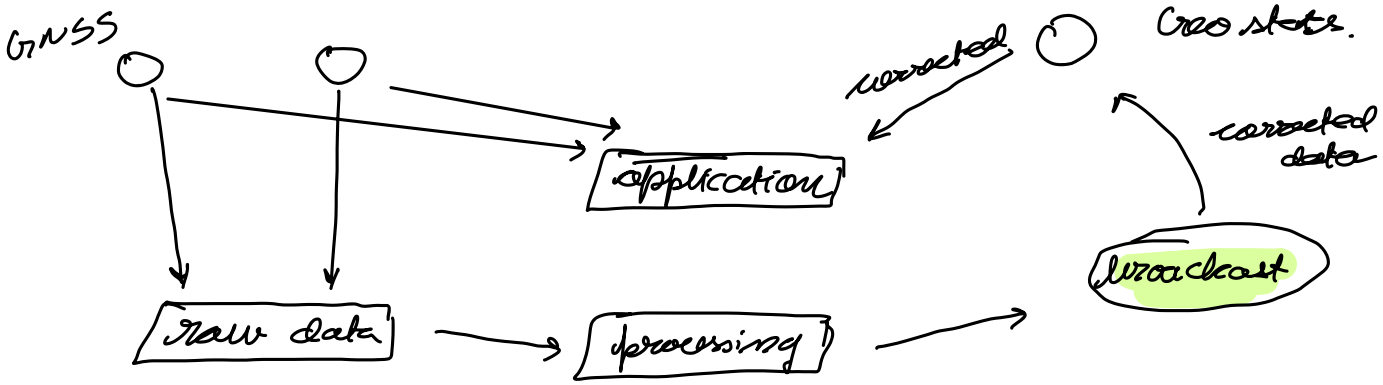
Corrections

- Satellite Based Augmentation System (SBAS)
- Real time Kinematic & differential GPS
- Precise Point Positioning
- GNSS + INS inertial navigation system → GPS + IMU

SBAS	RTK differential GPS	PPP	GNSS + INS
→ additional satellite data & ground station	→ signal errors corr in real time	→ no base station	→ tunnels
→ limited coverage	→ requires base station (10-20 km)	→ coverage km ↑	→ complex + cost
→ reliable for aviation (m)	→ cm level accuracy	→ consistent accuracy	→ NO GNSS signal high accuracy
	→ survey, construction, agriculture		

① SBAS (Satellite Based Augmentation Systems)

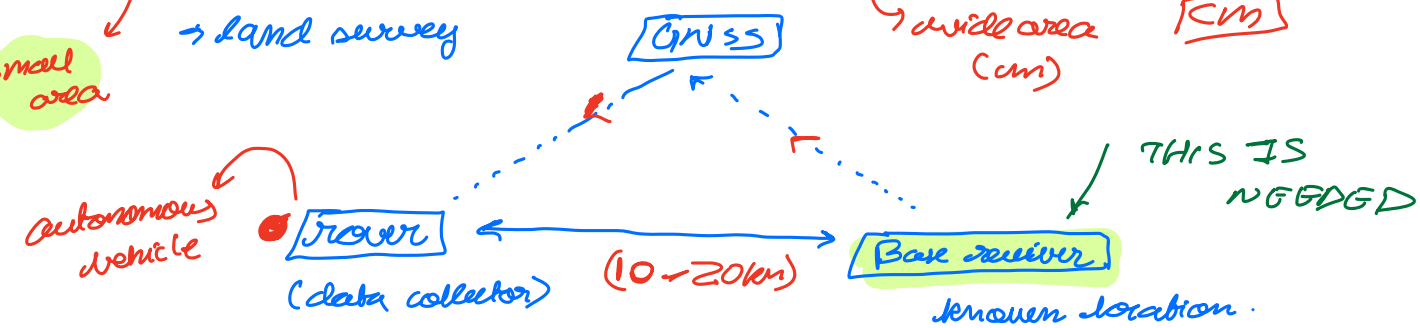
→ ground based correction using extra satellites
→ limited coverage
→ reliable for aviation (commercial guarantee)



→ GAGAN (India)
→ interoperability (7.6 m)
USA → 1m - 1.5m

② Real Time Kinematics & differential GPS (RTK-d GPS)

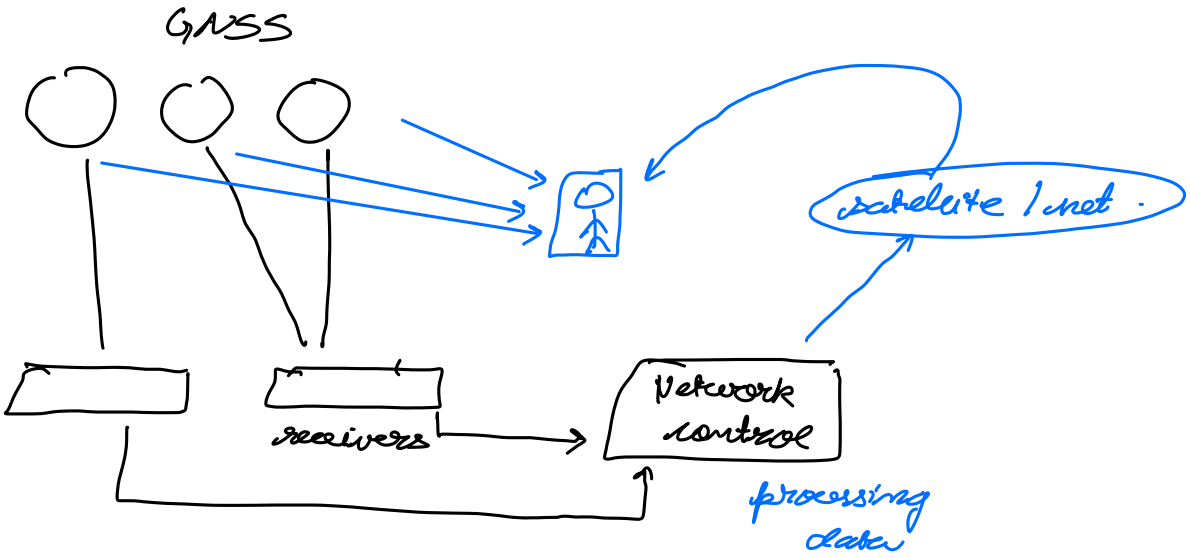
→ land survey
small area
wide area (cm)
cm



③ PPP (Precise Point Positioning)

→ NO BASE STATION REQUIREMENT

→ RTK is impractical
→ worldwide reference station



---

④ GNSS (Inertial Navigation System)

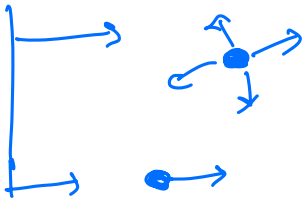
→ Kalman filter
→ GPS / IMU

Overview of GNSS-INS Integration:

1. **Combining Technologies**: GNSS provides absolute positioning using satellite signals, while INS offers relative positioning based on internal sensors that measure linear and angular movements.
2. **Operational Mechanics**: An INS system uses accelerometers and gyroscopes to track the vehicle's motion. These sensors determine the acceleration and rotation, which, when integrated over time, provide velocity and displacement from an initial known position.
3. **High Update Rate of INS**: INS systems operate at high frequencies (typically around 1 kHz), offering rapid updates on position and orientation. This is crucial for dynamic environments and when GNSS signals are temporarily lost (e.g., in tunnels or urban canyons).
4. **Error Accumulation in INS**: While INS can quickly calculate changes in position and orientation, its accuracy degrades over time due to error accumulation in the integration process. This drift is inherent in INS due to small errors in measurement being compounded as they are continuously integrated.
5. **Correction with GNSS**: To correct the drift from the INS, GNSS provides periodic absolute position updates. These updates reset the accumulated errors in the INS data, anchoring the INS calculations to a known position.
6. **Kalman Filter for Fusion**: A Kalman Filter is commonly used to fuse data from GNSS and INS. This statistical method optimally combines the high-frequency updates from INS with the less frequent but accurate position fixes from GNSS. It estimates the true vehicle state by considering the uncertainties in data from both sources.
7. **Redundancy and Reliability**: When GNSS signals are unavailable or unreliable, the INS can maintain localization for a short period with reasonable accuracy, thus ensuring continuity of navigation. Conversely, when INS data starts to drift, GNSS can recalibrate the system.
8. **Application Scenarios**: This integration is particularly beneficial in automotive navigation for autonomous vehicles, where maintaining continuous and accurate localization is critical for safety and effective operation.

② LIDAR (localization) → vehicle kidnap problem??

→ Light detection & Ranging
⇒ illuminate reflect
→ creation of 3D (map)



Light source more energy

Doplex effect complex system efficient

High energy LASER
→ cloud property
→ temperature

Low energy LASER
→ AIAV

5 params

→ 1 laser beam → 2 angles set
→ 3 to point of origin

→ cost function.
⇒ data segmentation

③ HD Maps :
→ super detailed
→ road lanes ..... ± info height. (cm)

Localization of LIDAR & HD MAPS → fuse different sensors

① LIDAR current
② prebuilt HD maps → GNSS/INS

→ compare & match.

live data + context

Workflow:

① initial
- → rough initial pos (GNSS + INS + Odometry)
- → defines search area
- → cloud of particles (ones) is placed.

Particle filter Technique

predict + update

② motion update
- → each particle is moved based on vehicle's speed
- → + noise

③ LiDAR scan
- → LiDar is compared to HD maps
- → each particle weighted

④ convergence
- → high weight → closeness

Visual Odometry :

→ estimating position & motion using camera images only
→ visual changes b/w frames
→ changes in environment

VO pipeline :

✓ ① capture image sequence
✓ ② feature detection
✓ ③ feature matching (across frames)
↔ ④ motion estimation (rel. camera movement)
↓ ⑤ trajectory update (combine with old frames)
✓ ⑥ error minimization (local optimization)
Bundle adjustment.

→ struggles low light
→ requires texture
→ blur / occlusions

① Stereo Visual odometry (stereo & depth) 3D

→ 2 cameras

→ measure depth & vehicle motion (estimation)

working

1) capture images
2) calculate disparity (diff. in both pixel) to get depth
3) extract features
4) match features
5) estimate motion
6) optimize

② Monocular Visual → assumptions & triangulation

③ Visual Inertial Odometry

camera + IMU (GPS+IMU)

Dead Reckoning & Wheel Odometry: → localization.

Dead Reckoning: navigational method to calculate current position by using previously determined position and advancing based upon speed & elapsed time.

Wheel odometry: → measuring rotational of the wheels to estimate changes in position over time.
→ robot navigational

Wheel encoders:

→ sensors attached to wheels
→ mechanical into electrical signals

Optical: → laser light – block light – photodetector
→ high precision
→ susceptible to dust

magnet

→ callibration
→ sensor fusion
→ env. adaptation

Working

① attached to wheel
② rotate – pulse
③ converted to digital
④ analysis.

Errors:

→ integration motion over time, errors accumulate exp. over long distances.

Systematic error (repeatable)

→ unequal size
→ distance b/w front & rear
→ misalignment
→ calibration of steering angle
→ air pressure

non systematic

→ wheel slip
→ rough terrain
→ sudden collision
→ fast turn / break
→ force
→ small object

Ways to reduce:

① Calibration
② thin wheels
③ deploy on smooth surface
④ ensure turning of steering

→ sensor fusion
→ multiple robots
→ mutual referencing
→ detect terrain & adjust params

Sensor Fusion

→ integration of data from multiple sensor source

local + map
obstacle

1) GPS + IMU : basic localization + orientation
2) LIDAR : 3D maps
3) Radar : detecting objects in poor visibility (speed + distance)
4)
5) Camera – visual → lane, sign, behaviour

→ Data integration

① Kalman Filter.
- → estimate x, v over time
- → accumulate over time
- → update using other data

keeps track of best guess

② Particle Filter
- → estimate state in non linear
- → set of points to represent initial state (bunch of friends to find)

→ Challenges
1. data align
2. computational
3. env. factors

→ Sol & Tech :
1. advanced fusion : DL / probabilistic
2. redundancy
3. real time processing hardware

Unit-2

Goal: sense surrounding & build reliable & detailed representation

Perception in Autonomous Driving

Introduction

Autonomous vehicles operate in complex environments where accurate perception is crucial. Sensors such as cameras, LiDAR, radar, and ultrasonic sensors help in gathering environmental data. Among these, cameras and LiDAR are the most informative. Computer vision plays a key role in processing visual data for autonomous driving. Since the 1980s, significant advancements have been made in perception, yet it remains a major challenge.

most info: LiDAR & camera

perception

Datasets

Large datasets are essential for improving perception algorithms in autonomous vehicles. These datasets help in quantitative evaluation, exposing weaknesses, and enabling fair comparisons. Common datasets for computer vision tasks include those for image classification, semantic segmentation, optical flow, stereo vision, and tracking.

For autonomous driving, key datasets include:

- KITTI
- cityscapes

- **KITTI**: A dataset created by Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago, featuring real-world street scenes collected using multiple sensors, including GPS, LiDAR, and cameras. It contains data for stereo vision, optical flow, visual odometry, object detection, object tracking, and road parsing.
- **Cityscapes**: A dataset focused on urban scene segmentation.

KITTI

→ Cityscapes

Newer datasets include:

- **Audi Autonomous Driving Dataset (A2D2)**
- **nuScenes**
- **Berkeley DeepDrive**
- **Waymo Open Dataset**
- **Lyft Level 5 Open Data**

These datasets provide high-precision 3D geometry, real-world scenarios, and diverse perception tasks, making them essential for advancing autonomous driving research.

KITTI:

1. stereo data: binocular, left & right (perceive depth)
2. optical flow: snapshot @ t_1, t_2 & compare pixel distance
3. visual odometry: perceive self movement using change in environment (Train main backbone network)
4. Object detection & orientation
5. object tracking
6. Road parsing.

→ high quality
→ real world
→ different obstacles

occulsion: partially / fully hidden objects?

Detection and Segmentation in Autonomous Driving

Detection

Autonomous vehicles must detect various objects on the road, including cars, pedestrians, obstacles, and lane markers. Object detection involves three main stages:

1. **Preprocessing** of input images
2. **Region of interest detection**
3. **Classification** of detected objects

Challenges in detection include variations in position, size, shape, orientation, and appearance. Additionally, detection must be performed in real time for safe navigation.

Key Detection Methods:

- **Histogram of Oriented Gradients (HOG) + Support Vector Machine (SVM)** (Dalal & Triggs, 2005): Uses sliding windows to extract HOG features and classify objects with a linear SVM.
- **Deformable Part Model (DPM)** (Felzenszwalb et al.): Splits objects into smaller parts to handle non-rigid shapes and uses latent SVM for detection.
- **LiDAR-based Detection:** While LiDAR performs well for car detection, it struggles with pedestrians and cyclists, highlighting the need for sensor fusion.

Pedestrian detection is particularly critical for safety, as human behavior is unpredictable and often occluded. Modern detectors rely on convolutional neural networks (CNNs), which are discussed in the next chapter.

Segmentation

Semantic segmentation enhances object detection by assigning a class label to each pixel, providing a structured understanding of the environment.

Traditional Approach:

- **Conditional Random Fields (CRF):** A graphical model where nodes (pixels) are assigned labels based on extracted features, ensuring spatial smoothness and object coherence.
- **Challenges:** CRF struggles with long-range dependencies and computational efficiency.

Advancements:

- Fully connected CRFs with pairwise potentials improve inference speed.
- Algorithms incorporating object class co-occurrence enhance accuracy.
- Deep learning approaches (discussed in the next chapter) improve segmentation performance using multi-scale features and contextual reasoning.

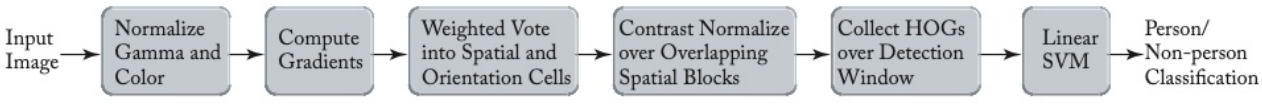


Figure 3.5: HOG+SVM. Adapted from Dalal and Triggs [12].

feature extraction → classification

HOG + SVM : Histogram of orientation Gradient

Detection → HOG + SVM
→ DPM
→ LIDAR + sensor fusion (CNN)

→ object detection

image → resize pixel → gradient magnitude & direction → feature vector → binning (direction as basis for)

$|x_c - x_l|$

↓ also normalize BUT OKAY....

[1, 2, 3, 4, ...]

DPM – deformable Part Model

such as part position.

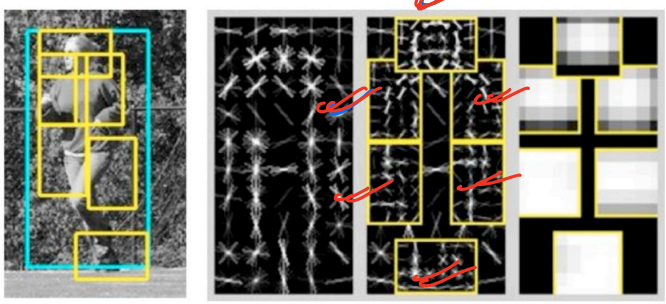


Figure 3.6: Deformable part model. Adapted from Felzenszwalb et al. [13], used with permission.

HOG feature

convolve with human template

↓

output to SVM

→ part (Body)

→ HOG + detector for individual part.

penalty score

ISSUE: → 3 legs.....

→ same person (limitation)

Segmentation

→ enhance object detection by assigning class label to each pixel

local classifier : each pixel classify

context understand ← Regional Features, Global Features → scene/overall structure.

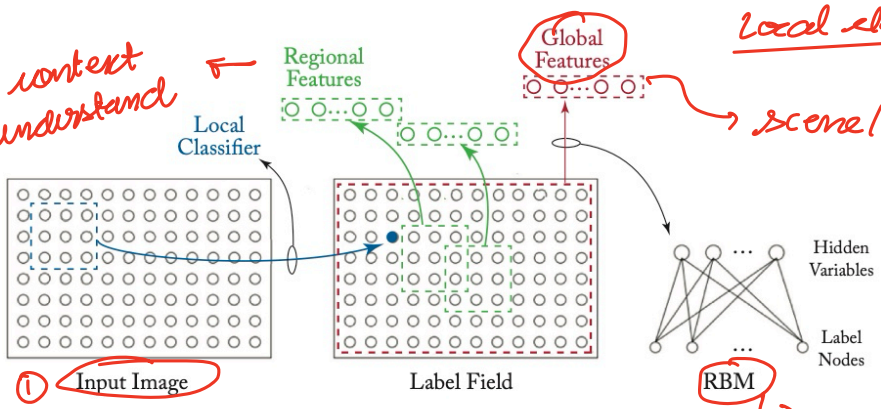


Figure 3.8: Graphical model representation of He et al. [17].

interaction b/w pixels

Restricted Boltzman Machine

Stereo, Optical Flow, and Scene Flow in Autonomous Driving

Stereo and Depth Perception

Autonomous vehicles require **3D spatial information** for navigation. While LiDAR provides **precise but sparse** depth data, stereo cameras offer **dense** visual information.

- **Stereo vision** mimics human binocular vision by capturing images from two slightly different angles and solving a **correspondence problem** to estimate depth.
- **Feature-based methods** use distinctive features (e.g., SIFT, SURF) for matching but provide sparse results.
- **Area-based methods** use spatial smoothness constraints to compute **dense** disparity maps but require more computation.
- **Global methods** (e.g., **Semi-Global Matching (SGM)**) optimize disparity estimation by minimizing energy functions, improving accuracy and efficiency.
- **Deep learning-based methods** now achieve the best performance in stereo matching (discussed in the next chapter).

The depth (z) of an object is derived using the formula:

$$z = \frac{B d}{f}$$

where **B** is the camera baseline, **d** is the disparity, and **f** is the focal length.

Optical Flow

Optical flow estimates **2D motion** by tracking intensity changes between consecutive images. Unlike stereo vision, which captures images simultaneously, optical flow must account for:

- **Motion variations** due to lighting changes, reflections, and transparency.
- **The aperture problem**, where motion ambiguity arises due to limited local observations.

To improve robustness, **alternative cost functions** have been introduced to replace the quadratic penalty in classical methods.

Scene Flow

Autonomous vehicles need **3D motion estimation** rather than just 2D optical flow. **Scene flow** extends optical flow by using **two consecutive stereo image pairs** to estimate both:

- **3D positions** of points.
- **3D motion** between time intervals.

The **KITTI Scene Flow 2015 benchmark** evaluates methods for accurate **3D motion estimation**, crucial for vehicle navigation and obstacle avoidance.

Aspect	Scene Flow	Stereo Vision
Definition	3D motion estimation of every pixel in the scene over time	Depth estimation from two cameras using disparity (difference in viewpoints)
Input Data	Sequence of stereo image pairs or RGB-D video	Stereo image pair (left & right images)
Output	3D position **and** 3D velocity (motion vector) of each point	3D position (depth) of each point
Temporal Information	Yes – tracks motion over time	No – captures only spatial information
Use Case	Useful for detecting **moving objects** and understanding **dynamic scenes**	Ideal for reconstructing **static 3D scenes**
Complexity	Higher computational cost due to motion tracking	Comparatively simpler and faster
Accuracy in Motion Detection	High – can estimate object speed and direction	Low – cannot estimate motion directly
Applications	Path prediction, obstacle motion, scene understanding	Lane detection, 3D mapping, obstacle avoidance
Limitations	Heavier on processing, more complex algorithms	Can struggle in low-light or texture-less areas
Suitability	Better for **dynamic environments**	Better for **static environment depth estimation**

ONLY

NO

OFFLINE?

Object Tracking in Autonomous Vehicles

Tracking Overview

Tracking estimates an object's **location, speed, and acceleration** over time, allowing autonomous vehicles to maintain safe distances and predict movement. This is particularly challenging for pedestrians and cyclists due to sudden direction changes.

Challenges in tracking:

- **Occlusion** (partial/full obstruction of objects).
- **Appearance similarity** among objects of the same class.
- **Variability in appearance** due to lighting, pose, and articulation.

Bayesian Filtering Approach

Tracking is traditionally modeled as a **sequential Bayesian filtering** problem with two main steps:

1. **Prediction:** The object's state is estimated based on past motion.
2. **Correction:** The state estimate is refined using new sensor observations.

A commonly used method is the **Particle Filter**, but its recursive nature makes recovery from missed detections difficult.

Alternative Approaches

- **Energy minimization:** Finds the optimal object trajectory by enforcing motion smoothness and appearance consistency. However, the large number of possible object hypotheses makes this approach computationally expensive.
- **Tracking-by-detection:** Detects objects in consecutive frames and links them, but faces **missed detections** and **false positives** from object detectors.

Markovian Decision Process (MDP) for Tracking

MDP-based tracking defines object states and transitions:

- **Active:** Object detected.
- **Tracked:** Object confirmed as valid.
- **Lost:** Object temporarily undetected but might reappear.
- **Inactive:** Object lost for too long, removed from tracking.

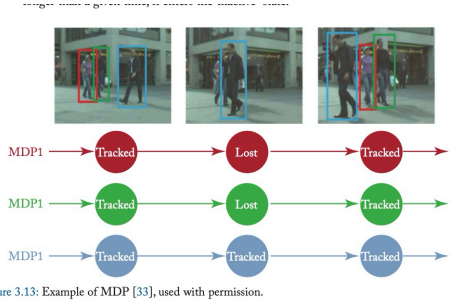


Figure 3.13: Example of MDP [33], used with permission.

MDP Tracking Algorithm:

- Uses an **SVM classifier** to validate detections.
- Applies a **tracking-learning-detection model** to maintain appearance consistency.
- Continuously updates the object's **bounding box template** for re-identification.
- ...tes based on learned **transition and reward functions**.

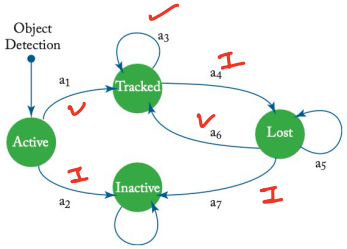


Figure 3.12: MDP formulation of tracking (based on [33]).

Part-2

CNNs and Object Detection in Autonomous Driving

4.1 Convolutional Neural Networks (CNNs)

CNNs are a type of deep neural network that use **convolution** as the primary computational operation. They were first introduced by LeCun et al. in 1988, inspired by the **visual cortex's** structure. CNNs excel in **computer vision** tasks due to:

- **Local connectivity:** Neurons only connect to nearby neurons within a **receptive field**.
- **Weight sharing:** Spatially shared weights reduce the number of parameters, making CNNs efficient.
- **Translation invariance:** CNNs learn patterns irrespective of their location in the image.

CNNs revolutionized computer vision, with models like **AlexNet (2012)** leading to state-of-the-art **autonomous driving perception systems**.

4.2 Object Detection in Autonomous Driving

Traditional Object Detection

Early detection methods relied on **hand-crafted features** and structured classifiers, but these struggled with **large data volumes and object variations**.

CNN-Based Object Detection

Girshick et al. introduced **R-CNN**, proving CNNs outperform traditional methods. Faster R-CNN improved detection by using a **Region Proposal Network (RPN)** for generating potential object locations.

Faster R-CNN Pipeline:

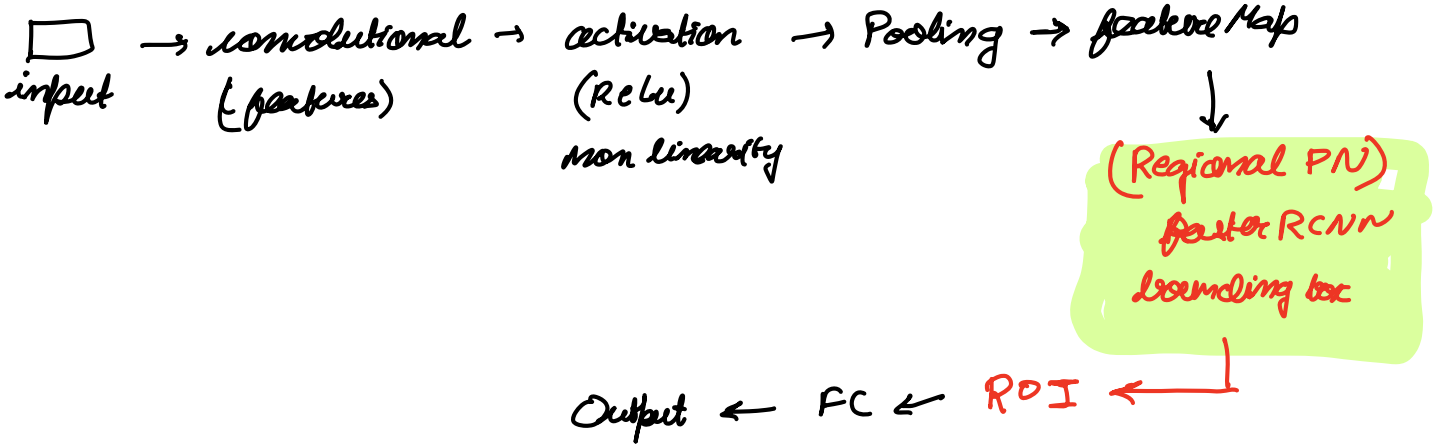
1. **RPN generates region proposals** by scanning feature maps using anchor boxes of different sizes (e.g., 128×128, 256×256, 512×512).
2. **ROI pooling refines proposals**, mapping them to a fixed-size feature map.
3. **Final classification and bounding box regression** predict object type and precise location.

Proposal-Free Algorithms

Some models avoid the region proposal step for real-time performance:

- **SSD (Single Shot MultiBox Detector):** Uses multiple convolutional layers to detect objects of varying sizes in a single pass.
- **YOLO (You Only Look Once):** Directly predicts object locations and classes in one forward pass, achieving high speed.

CNN


input
convolutional
(features)
activation
(Relu)
non linearity
Pooling
feature Map
(Regional PN)
faster RCNN
bounding box
ROI
FC
Output

Semantic Segmentation in Autonomous Driving

4.3 Semantic Segmentation → pixel level identification

Semantic segmentation is **crucial** in autonomous driving perception, as it helps identify **road surfaces, obstacles, and other scene elements** at the pixel level.

Fully Convolutional Networks (FCN)

- FCNs transform traditional CNNs (e.g., VGG-19) by **removing the softmax layer** and replacing fully connected layers with **1×1 convolutions**.
- They allow **input of any size** and predict **per-pixel labels** for segmentation.
- However, **small object segmentation** is challenging due to dominant larger receptive fields.

Pyramid Scene Parsing Network (PSPNet)

To address **global-local feature integration**, Zhao et al. proposed **PSPNet**, which enhances FCNs using a **pyramid pooling module**.

PSPNet Workflow:

1. **Feature extraction:** A CNN (ResNet) extracts feature maps from the input image.
2. **Pyramid pooling:** Multi-level pooling (1×1, 2×2, 3×3, 6×6) aggregates **contextual information**.
3. **Feature compression:** Feature maps are passed through **1×1 convolutions** for dimensional reduction.
4. **Upsampling & fusion:** Pooled features are upsampled and concatenated with original feature maps for **final pixel-wise classification**.

Key Findings from PSPNet Experiments:

- **Average pooling** performs better than max pooling.
- **Multi-level pyramid pooling** improves segmentation over global-only pooling.
- **Feature dimensionality reduction** helps maintain efficiency.
- **Auxiliary loss** aids deep network optimization.

PSPNet won **1st place** in the **ImageNet Scene Parsing Challenge 2016** and achieved **state-of-the-art** performance on **PASCAL VOC 2012** and **Cityscapes datasets**.

Conclusion

Deep learning, particularly **FCN-based architectures like PSPNet**, has significantly advanced **semantic segmentation** in autonomous driving, ensuring more **precise road scene understanding** for safer navigation.

Stereo and Optical Flow in Autonomous Driving

4.4 Stereo and Optical Flow

Stereo vision and optical flow are **key techniques** for depth estimation and motion analysis in autonomous driving. Both involve **matching corresponding points** between two images.

4.4.1 Stereo Vision

- **Content-CNN (Siamese Architecture)**:
  - Uses two CNN branches (for left and right images) with **shared weights**.
  - Outputs are merged via an **inner-product layer** to estimate pixel disparity.
  - **Disparity estimation** is treated as a **classification problem** over possible disparity values.
  - Achieves **fast processing** on the KITTI Stereo 2012 dataset.
  - **Post-processing** (e.g., local smoothing, semi-global matching) enhances accuracy and 3D depth estimation.

4.4.2 Optical Flow

- **FlowNet (Encoder-Decoder Architecture)**:
  - **FlowNetSimple**: Stacks images and applies convolution layers, but is computationally heavy.
  - **FlowNetCorr**: Extracts features separately, merges via a **correlation layer**, and applies convolutions.
  - Uses **"up-convolution"** to restore resolution after compression.
  - FlowNet achieves **competitive performance** on KITTI with **0.15-sec GPU inference time**.
  - **SpyNet** refines optical flow estimation using a **coarse-to-fine spatial pyramid** approach.
  - SpyNet achieves **state-of-the-art performance** with a **lightweight model**, ideal for mobile applications.

4.4.3 Unsupervised Learning for Dense Correspondence

- **Challenge**: Ground truth dense correspondence is **expensive and difficult** to collect.
- **MonoDepth & MonoDepth2** use **unsupervised learning** from video frames.
  - **Loss components**:
    1. **Appearance Matching Loss** – Assumes corresponding pixels in two views are visually similar.
    2. **Disparity Smoothness Loss** – Enforces local smoothness with occasional depth discontinuities.
    3. **Left–Right Disparity Consistency Loss** – Ensures disparity consistency between left and right views.
  - Uses an **encoder-decoder** structure with skip connections.
  - Performs **better than traditional methods**, with improvements by increasing training data.

UNIT-3

Prediction & Routing ⇄ Planning & control, Traffic Prediction, Lane level Routing

Planning and Control in Autonomous Vehicles

The **Planning and Control** architecture in autonomous vehicles integrates multiple modules to ensure safe and reliable operation.

Architecture: Planning and Control in a Broader Sense

- The **Map and Localization** module processes raw sensor data (e.g., point cloud, GPS) to determine the vehicle's position.
- The **Perception** module detects surrounding objects.
- Other modules, including **routing, traffic prediction, behavioral decision, motion planning, and feedback control**, contribute to predicting environmental behavior and determining the vehicle's movement.

All modules operate within a synchronized clock cycle, where they fetch, compute, and publish data for downstream consumption. Collaboration between **perception and planning/control** is critical, ensuring minimal module scope and problem complexity.

The **divide-and-conquer** approach structures functional modules according to data flow, categorizing **prediction and routing** under planning and control as dependency modules. Traditional decision-making, planning, and control aspects are discussed in the next chapter.

While conventional techniques dominate current systems, AI-driven **end-to-end solutions** are becoming increasingly viable. These solutions, covered in a later chapter, aim to seamlessly process sensor inputs and map data to execute vehicle maneuvers.

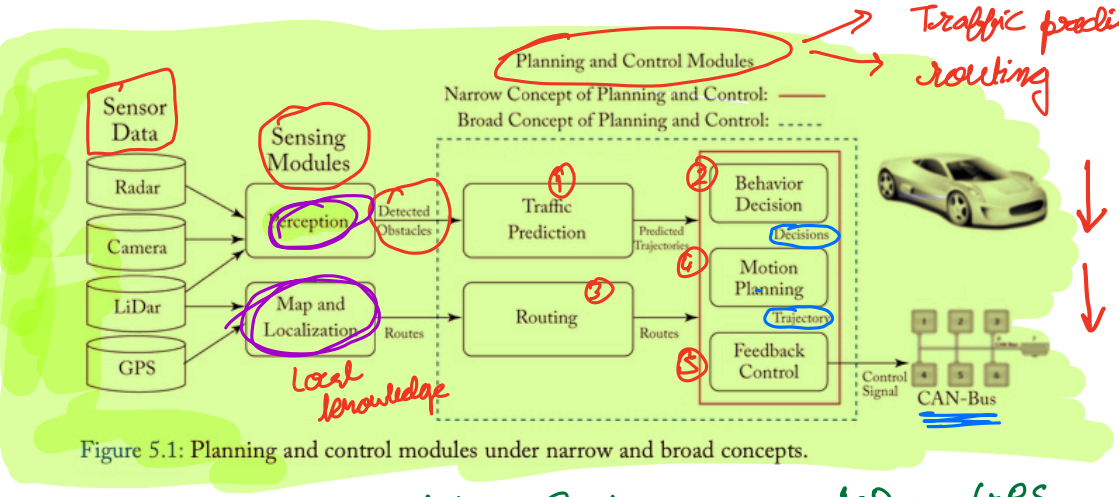


Figure 5.1: Planning and control modules under narrow and broad concepts.

Traffic prediction, routing

sensor data → sensing modules → Plan & control

Adv
① ↑ safety
② efficiency
③ scalability

① sensor data - Radar, camera, LiDar, GPS

② sensing Modules - Perception, Map & localization

detection, location knowledge

(traffic pred.) (Routing)

① behavioural
② Motion → Trajectory
③ Feedback → steering, a↑↓

decisions

PLAN & CONTROL

Improvement
① vehicle to everything
② AI algo
③ sensor robustness

Disadvantage
① cost
② complexity
③ ethical & legal issues

Scope of Each Module in Autonomous Vehicle Planning and Control

The **planning and control** system in autonomous vehicles consists of multiple interdependent modules, each solving a specific sub-problem to improve efficiency and modularity.

Key Modules and Their Functions

1. **Routing Module**
   - Issues top-level navigation commands.
   - More complex than traditional map services, relying on HD maps tailored for autonomous driving.
2. **Traffic Prediction Module**
   - Processes perceived objects (e.g., vehicles, pedestrians) and predicts their future trajectories.
   - Initially implemented as a software library but now a dedicated module for better real-time performance.
   - Inputs are used by the behavioral decision module.
3. **Behavioral Decision Module**
   - Acts as a "co-pilot," determining how the vehicle should respond based on routing and traffic prediction.
   - Generates commands such as lane-following, stopping, or yielding.
   - Converts individual behavioral decisions into constraints for motion planning.
4. **Motion Planning Module**
   - Computes an optimized trajectory for the vehicle based on constraints from the behavioral decision module and goals from the routing module.
   - Ensures smooth and executable trajectories while maintaining consistency across planning cycles.
   - Uses redundant mapping and perception data as a backup for failed predictions or newly detected obstacles.
5. **Feedback Control Module**
   - Converts planned trajectories into drive-by-wire signals for vehicle control (braking, steering, acceleration).
   - Ensures the actual path conforms to the planned trajectory while considering physical constraints.

Collaboration and Design Philosophy

- Modularization simplifies software development by dividing complex problems into smaller, manageable tasks.
- The system follows a **data stream hierarchy**, where downstream modules rely on upstream computations.
- Conflicts are resolved at the upstream level whenever possible, ensuring consistency across modules.

This structured approach provides a **comprehensive, coherent solution** to autonomous vehicle planning and control, balancing modular design with real-world adaptability.

CAN ≡ Controller Area Network

Traffic Prediction

Objective

Traffic prediction aims to forecast the behavior of detected objects (e.g., vehicles, pedestrians) in the near future by computing spatial-temporal trajectory points. The prediction process considers object attributes such as position, velocity, heading, and acceleration, along with behavioral and environmental factors.

Problem Formulation

Traffic prediction is divided into two sub-problems:

1. **Classification Problem:** Determines categorical behaviors, such as lane changes or pedestrian crossing.
2. **Regression Problem:** Predicts precise trajectory paths, including speed and timing.

Behavior Prediction as Classification

- Vehicle behavior is more predictable than pedestrian or cyclist behavior.
- A classification model predicts whether a vehicle will follow a particular lane sequence.
- Instead of defining categories based on specific intersections, behavior is classified based on lane sequences.
- Lane sequence classification enables scalable and structured learning.

Feature Design for Vehicle Behavior Prediction

Three key categories of features:

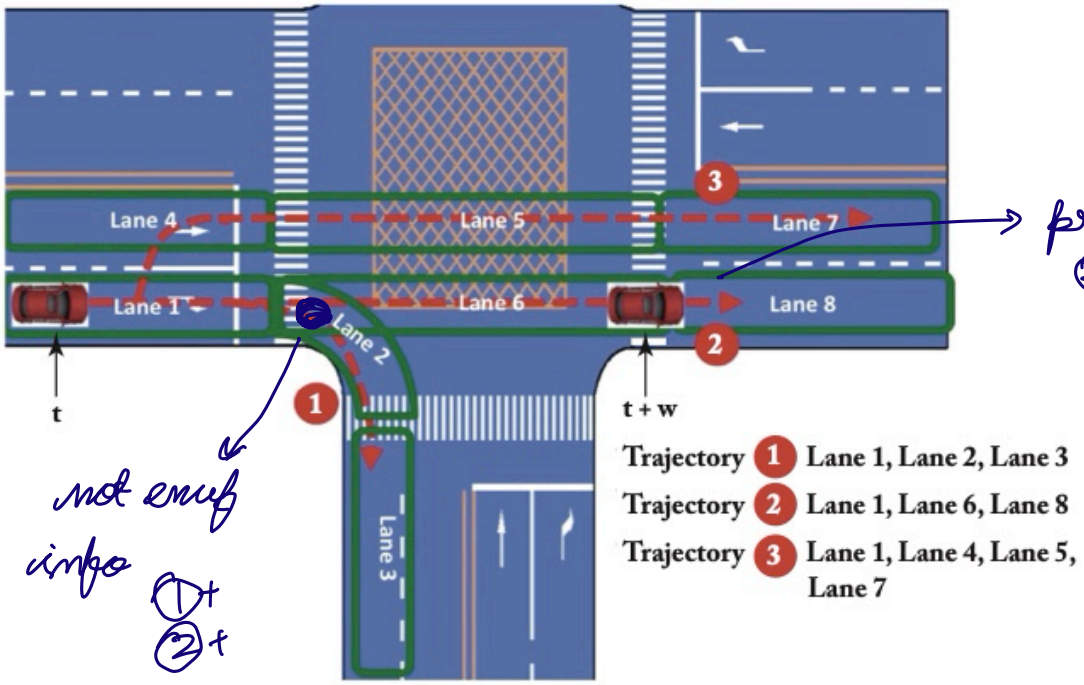
1. **Vehicle History Features:** Includes absolute and relative positions of a vehicle over a historical window.
2. **Lane Sequence Features:** Captures lane properties such as curvature, heading, and boundary distance.
3. **Surrounding Object Features:** Includes distances between a vehicle and adjacent lane vehicles.

Model Selection

1. **Memory-less Models (SVM, DNN):** Require historical data encoding; suitable for simpler environments.
2. **Memory Models (LSTM, RNN):** Retain past information; better for complex traffic scenarios.

Performance Metrics

- **Precision:** Measures how many predicted trajectories match actual vehicle movements.
- **Recall:** Measures how many actual behaviors were successfully predicted.



→ pretty obvious ② +ve.

not enuf info ① + ② +

Figure 5.3: Formulating a behavioral traffic prediction problem as a binary classification problem with regard to lane sequences.

→ lane seq is effective

This is trajectory classification.

behavioural classification problem → Trajectory problem

↓

feature engineering from raw data

- Vehicle history
  - → frames
  - → previous lane
  - → future lane
  - → long / lat
  - → info params
  - → relative distance
- lane properties
  - → road / curve
  - → distance C/R
  - → turning
- surroundi
  - → rear vehicle
  - → long / lat
  - → etc.

① behaviour is determined – high level next frame decision.

② vehicle trajectory – execute the decisions - Motion Planning

→ Kalman Filter

Vehicle Trajectory Generation in Traffic Prediction Systems

The section discusses vehicle trajectory generation within traffic prediction systems. After determining a vehicle's behavior, the system must generate a spatial-temporal trajectory following the predicted lane sequence. A Kalman-Filter is used to track the vehicle's position along the lane's centerline in (s, l) coordinates, where **s** represents longitudinal distance and **l** represents lateral deviation. The motion transformation matrix governs trajectory prediction, with the speed of approximation to the centerline controlled by the parameter βt.

Machine learning-based approaches, such as regression models, could also be used for trajectory generation by leveraging historical trajectory data. However, the authors argue that behavior prediction is more critical than trajectory modeling.

The prediction problem is divided into two phases:

1. **Behavior Prediction** – Modeled as a binary classification problem for each lane sequence.
2. **Trajectory Generation** – Uses motion planning techniques or rule-based Kalman filtering.

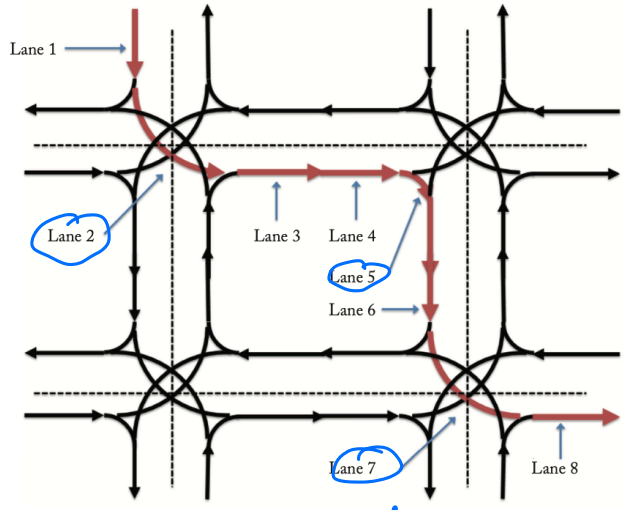
Interactions among vehicles are not explicitly modeled but may be implicitly considered with high-frequency predictions. The extracted features for prediction include:

- **Vehicle History Features**: Position, speed, heading, acceleration, lane boundary distance, and vehicle dimensions.
- **Lane Features**: Lane position, curvature, heading, boundary distances, and lane type.
- **Surrounding Obstacle Features**: Positions, speeds, and headings of the closest front and rear vehicles.

These features help classify whether a vehicle will follow a specific lane sequence, improving traffic prediction accuracy.

Lane Level Routing

→ exact sequence of lanes
→ HD map
→ output is lane sequence
→ acts as input for motion planning



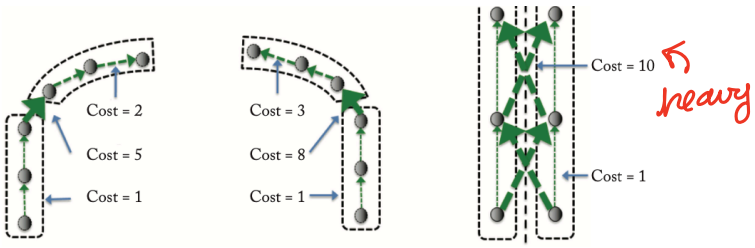
Virtual

→ virtual lanes (turns)

(start_lane, start_point, till)

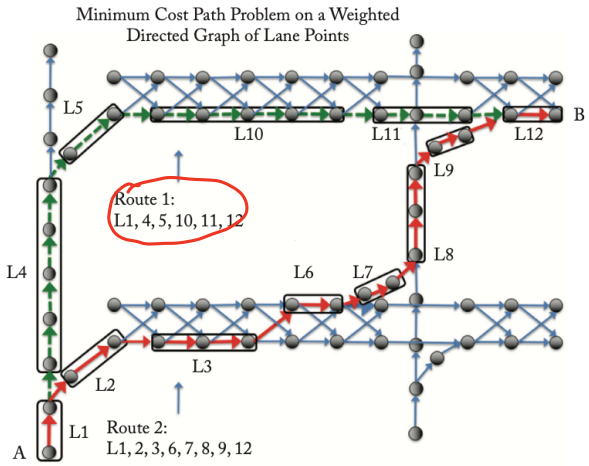
1 0 end

2 0 end ...



heavy

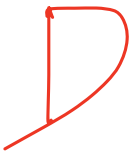
Figure 5.6: Costs of edges connecting lane points under scenarios of Right Turn, Left Turn, and Switch Lanes.



→ low execution difficulty
→ safe & smooth

5.7: Routing as a minimum cost path problem on lane points connected weighted graph.

Routing algo → Dijkstra
→ A*



1. Set distance to all nodes as ∞, distance to source = 0
2. Mark all nodes as unvisited
3. While there are unvisited nodes:
   a. Select the unvisited node with the smallest distance
   b. For each neighbor of this node:
      - Calculate new distance
      - If new distance < old distance, update it
   c. Mark the current node as visited
4. Reconstruct path using previous node map

Initialize:
    dist[v] = ∞ for all nodes v
    dist[source] = 0
    prev[v] = None
    unvisited = all nodes

While unvisited is not empty:
    u = node with smallest dist[u] in unvisited

    For each neighbor v of u:
        alt = dist[u] + cost(u, v)
        If alt < dist[v]:
            dist[v] = alt
            prev[v] = u

    Remove u from unvisited

Return path by backtracking from destination using prev[]

1. Set g(n) = ∞ for all nodes, g(source) = 0
2. Set f(n) = ∞ for all nodes, f(source) = g + heuristic
3. Add source to open set
4. While open set is not empty:
   a. Select node with lowest f-score
   b. If node is destination, stop and reconstruct path
   c. For each neighbor:
      - Calculate tentative g = current g + cost
      - If tentative g < old g, update g, f and previous node
      - Add neighbor to open set if not already there

A*

```
Initialize:
    g[v] = ∞, f[v] = ∞ for all nodes v
    g[source] = 0
    f[source] = g[source] + h(source, goal)
    open_set = {source}
    prev[v] = None

While open_set not empty:
    current = node in open_set with lowest f[current]

    If current == goal:
        return reconstruct path using prev[]

    For each neighbor v of current:
        tentative_g = g[current] + cost(current, v)

        If tentative_g < g[v]:
            prev[v] = current
            g[v] = tentative_g
            f[v] = g[v] + h(v, goal)
            Add v to open_set
```

Q-learn

```
Initialize Q(s, a) = 0 for all states and actions

For each episode:
    Initialize state s

    While not terminal:
        Choose action a using ε-greedy policy
        Take action a → observe reward r and new state s'

        Update Q-value:
        Q(s, a) = Q(s, a) + α * [r + γ * max Q(s', a') - Q(s, a)]

        s = s'
```

DQN

Initialize Q-network with weights θ
Initialize target network with weights $\theta^- = \theta$
Initialize replay buffer D

For each episode:
  Initialize state s

  While not terminal:
    Select action a using ε-greedy policy:
      $a = \text{argmax } Q(s, a; \theta)$

    Take action a $\rightarrow$ observe reward r and next state s'

    Store (s, a, r, s') in D

    Sample mini-batch from D:
      For each (s, a, r, s') compute target y:
        $y = r + \gamma * \max Q(s', a'; \theta^-)$

    Update Q-network by minimizing loss:
      $L = (y - Q(s, a; \theta))^2$

    Every C steps:
      $\theta^- \leftarrow \theta$

Typical Routing Algorithms for Autonomous Vehicles

The **Routing problem** in autonomous driving can be solved using shortest path algorithms. Two commonly used algorithms are **Dijkstra's Algorithm** and the *A Algorithm**

1. Dijkstra's Algorithm for Autonomous Vehicle Routing

Dijkstra's algorithm finds the shortest path between a **source node** and a **destination node** on a weighted graph. It is applied to lane-point-based routing as follows:

Steps in Dijkstra's Algorithm:

1. **Graph Construction:**
   - Extract lane graph data from the **HD map** within a given radius.
   - Define **lane points** as graph nodes.
   - Set the source lane point (closest to the vehicle's location) and destination lane point.
   - Initialize all nodes' costs to **infinity**, except the source node (**cost = 0**).
2. **Processing Nodes:**
   - Set the **current node** to the source.
   - Maintain an **unvisited set** of lane points and a **prev_map** to track predecessors.
3. **Updating Distances:**
   - For each **adjacent lane point**, compute a **tentative distance** based on edge weights.
   - If the tentative distance is smaller than the current distance, update it.
4. **Iteration:**
   - Mark the current lane point as **visited** and remove it from the unvisited set.
   - Continue until the destination is reached or no valid path exists.
5. **Path Reconstruction:**
   - If a path is found, reconstruct it using the **prev_map** and return a sequence of lane segments **{(lane, start_position, end_position)}**.

Complexity Analysis:

- Using a **priority queue**, the **time complexity** is **O(|E| + V log V)**, where V = number of lane points and E = number of edges.

*2. A Algorithm for Autonomous Vehicle Routing**

The *A algorithm** is an optimized search method that extends Dijkstra's algorithm by incorporating a **heuristic function (h(v))**. It balances **known path costs (g(v))** with an **estimated cost (h(v))** to reach the destination.

Formula for Node Cost:

f(v) = g(v) + h(v)

- **g(v):** Cost from the source to node v.
- **h(v):** Estimated cost from node v to the destination.

Steps in A Algorithm:*

1. **Initialize:**
   - Define **openSet** (nodes to explore).
   - Set source lane point **cost = 0** and all others to infinity.
2. **Node Expansion:**
   - Extract the node with the **lowest f(v) value** (best estimated path).
   - Update distances for adjacent lane points and adjust their **g(v)** and **h(v)** values.
3. **Heuristic Function (h(v)):**
   - In autonomous vehicle routing, **h(v)** can be defined as the **Mercator distance** between two lane points.
   - If **h(v) is admissible** (never overestimates the true cost), A* guarantees the shortest path.
4. **Path Reconstruction:**
   - Similar to **Dijkstra's Algorithm**, the shortest path is reconstructed and returned as a sequence of lane segments.

Comparison with Dijkstra's Algorithm:

- *A is more efficient** because it uses **heuristics** to guide the search.
- If **h(v) = 0**, A* reduces to **Dijkstra's Algorithm** (i.e., pure shortest path search).

Conclusion:

- **Dijkstra's Algorithm** guarantees the shortest path but explores all possible paths.
- *A Algorithm** optimizes the search by using heuristics, making it more efficient when a good heuristic is available.
- Both algorithms are crucial for **autonomous vehicle lane-level routing**, ensuring **safe and optimal navigation** in complex road networks.

Routing Graph Cost: Weak vs. Strong Routing

In **autonomous vehicle routing**, the **configuration of costs** between lane points is more critical than the choice of algorithm. Costs can be dynamically adjusted based on **traffic conditions, road closures, or lane preferences**.

Dynamic Cost Adjustments:

- **High cost** for congested or restricted roads to avoid them.
- **Infinite cost** for blocked roads to prevent routing through them.
- **Ad-hoc graph construction:** If a route is unavailable within a **small radius**, the system can **reload a larger radius** and recompute the route.

Types of Routing Requests:

1. **Passenger-Initiated Routing:** When the vehicle starts its journey, a route is computed from the source to the destination.
2. **System-Initiated Routing:** Modules like **Behavior Decision or Motion Planning** may request re-routing based on real-time driving conditions.

Strong vs. Weak Routing

1. Strong Routing

- The **vehicle strictly follows** the computed route lane-by-lane.
- If deviations are required (e.g., due to obstacles), a **re-routing request** is triggered.

Example: If a slow-moving vehicle is ahead, the autonomous car will **reduce speed** and follow it instead of switching lanes.

2. Weak Routing

- The vehicle can **deviate from the planned route** when necessary.
- Allows **lane changes** to overtake slow vehicles, similar to human driving behavior.

Example: Instead of slowing down behind a slow vehicle, the car may **switch lanes**, overtake, and merge back.

Safety Considerations

Regardless of strong or weak routing, in **emergency situations**, the vehicle prioritizes **safety first** and may request **urgent re-routing** to handle critical scenarios.

Behavioral Decisions in Autonomous Vehicles

The **behavior decision module** acts as the "co-driver," processing multiple data sources, including:

1. **Routing output** – Lane sequences and positions.
2. **Vehicle attributes** – Location, speed, heading, and target lane.
3. **Historical data** – Previous decisions (follow, stop, turn, switch lanes).
4. **Obstacle information** – Objects' locations, speeds, and predicted paths.
5. **Traffic & map data** – Lane relationships, traffic signals, and legal lane switches.
6. **Local traffic rules** – Speed limits, right-turn laws, etc.

Given the complexity of integrating these data sources, **rule-based systems** are widely used for behavioral decision-making in autonomous vehicles. Many successful systems, like Stanford's "Junior" and CMU's "Boss," use **Finite-State Machines (FSMs)** with cost functions. However, **Bayesian models**, including **Markov Decision Processes (MDP)** and **Partially Observable MDPs (POMDPs)**, are increasingly applied in academic research.

Markov Decision Process (MDP) Approach

An **MDP** is defined by:

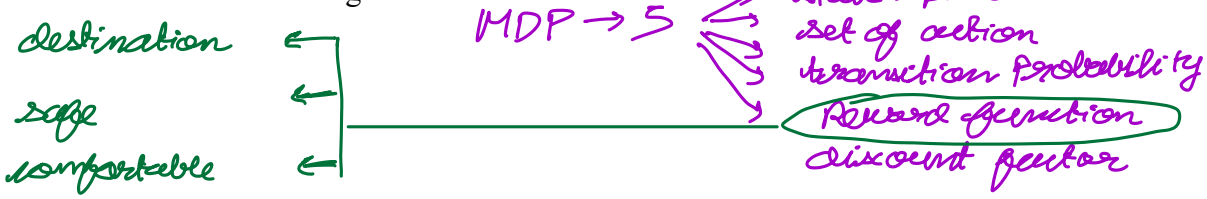
1. **State Space (S):** Vehicle's position, lanes, and road environment.
2. **Action Space (A):** Possible decisions (e.g., Follow, Switch Lane, Turn, Yield, Stop).
3. **Transition Probability (Pa):** Probability of moving from one state to another given an action.
4. **Reward Function (Ra):** Evaluates actions based on safety, comfort, and goal achievement.
5. **Discount Factor (γ):** Prioritizes short-term rewards over long-term ones.

The goal is to find an **optimal policy (π)** that maps states to actions to **maximize the accumulated reward** using **Dynamic Programming** or **Bellman's Value Iteration**.

Key Considerations for a Good Reward Function:

1. **Reaching the destination** – Incentivize actions that follow the routing plan.
2. **Safety & collision avoidance** – Penalize high-risk moves; reward safer paths.
3. **Comfort & smoothness** – Prefer gradual speed changes over abrupt maneuvers.

Balancing these factors is crucial to designing an effective MDP-based decision system for autonomous driving.



Scenario-Based Divide and Conquer Approach in Behavioral Decision-Making for Autonomous Vehicles

The **Scenario-Based Divide and Conquer** approach decomposes the vehicle's surroundings into multiple independent **scenarios**, each with its own rules to determine **Individual**

Divide & conquer

Decisions for objects. These individual decisions are then consolidated to form a **Synthetic Decision** that dictates the vehicle's behavior.

Key Concepts:

1. **Synthetic Decision**: The final, top-level decision for the autonomous vehicle, considering all road objects and scenarios.
   - Examples: **Follow**, **Turn**, **Change Lane**, **Stop**, Cruise
   - Parameters include lane position, speed limits, and distances.
2. **Individual Decision**: Decisions assigned to individual objects based on scenario-specific rules.
   - Examples: **Follow**, **Stop**, **Overtake**, **Yield**, **Attention**.
   - Parameters include the object's ID, distances, and time gaps.
3. **Scenario Construction**: The surrounding world is divided into **layered scenarios**, where:
   - Objects typically belong to one scenario.
   - Lower layers provide information to upper layers.
   - Routing intentions and previous computations are shared.
4. **Decision Merging & Conflict Resolution**:
   - Some objects may appear in multiple scenarios with conflicting decisions.
   - A merging layer ensures safety and coherence.
   - The **final synthetic decision** is then computed.
5. **System Framework (Figure 6.4)**:
   - **Layer 0**: Constructs scenarios (e.g., Master Vehicle, Lanes, Traffic Lights).
   - **Layer 1**: Computes individual decisions within each scenario.
   - **Layer 2**: Merges decisions for consistency.
   - **Final Step**: Generates the **synthetic decision**, which is passed to the motion planning module.

common

individual

distance, current, target, speed limit, car id

By using this structured approach, autonomous vehicles make logical and safety-compliant driving decisions while optimizing motion planning.

Synthetic Decision

Synthetic Decision	Parametric Data
Cruise	➢ Current lane ➢ Speed limit of the current lane
Follow	➢ Current lane ➢ *id* for the vehicle to follow ➢ Speed to reach minimum of current lane speed limit and speed of the vehicle to follow ➢ Not exceeding 3 m behind the vehicle in front
Turn	➢ Current lane ➢ Target lane ➢ Left or right turn ➢ Speed limit for turning
Change Lane	➢ Current lane ➢ Target lane ➢ Change lane by overtaking and speed up to 10 m/sec ➢ Change lane by yielding and speed down to 2 m/sec
Stop	➢ Current lane ➢ *id* for any object to stop, if any ➢ Stop by 1 m behind the object to stop

Figure 6.1: Synthetic decision with its parameters in behavioral decision.

Individual Decision		Parametric Data
Vehicle	Follow	➢ *id* for the vehicle to follow ➢ Speed to reach for following the vehicle ➢ Distance to keep for following the vehicle
	Stop	➢ *id* for the vehicle to stop ➢ Distance to stop behind the vehicle
	Attention	➢ *id* for the vehicle to stop ➢ Minimum distance to keep while paying attention to the vehicle
	Overtake	➢ *id* for the vehicle to overtake ➢ Minimum distance to keep for overtaking ➢ Minimum time gap to keep for overtaking
	Yield	➢ *id* for the vehicle to yield ➢ Minimum distance to keep for yielding ➢ Minimum time gap to keep for yielding
Pedestrian	Stop	➢ *id* for the pedestrian to stop ➢ Minimum distance to stop by the pedestrian
	Swerve	➢ *id* for the pedestrian to swerve ➢ Minimum distance to keep while swerving around

Figure 6.2: Individual decisions with parameters in behavior decision module.

Overview

Aspect	Synthetic Decision	Individual Decision
Definition	A comprehensive decision that dictates the overall behavior of the autonomous vehicle itself, based on the consolidation of all individual decisions.	Decisions made for specific elements or objects in the vehicle's environment, such as other vehicles, pedestrians, traffic signs, etc.
Scope	Broad, considering the collective input from multiple individual decisions to guide the vehicle's overall behavior.	Narrow, focusing on specific objects or scenarios, assessing how the vehicle should interact with or react to each individually.
Output	High-level commands like "change lanes," "accelerate," "decelerate," "follow the vehicle ahead," or "stop at the traffic light."	More detailed actions associated with specific entities, like "follow this vehicle," "yield to that pedestrian," or "overtake this bicycle."
Dependency	Depends on the aggregation and interpretation of multiple individual decisions.	Each decision is relatively independent, typically associated with one particular object or aspect of the environment.
Purpose	To provide a final, overarching behavioral strategy that the vehicle will execute.	To handle specific interactions with discrete elements within the environment.
Example in Use	If the synthetic decision is to change lanes, this command would be based on individual decisions regarding vehicles in the adjacent lanes and traffic conditions.	An individual decision might be to slow down because a pedestrian is detected crossing the road ahead, or to prepare for a lane change based on a clear lane.
Impact on Motion Planning	Guides the overall trajectory and strategic actions of the vehicle.	Individual decisions feed into the synthetic decision, impacting specific tactical adjustments before being integrated into the broader strategy.
Formulation	Formulated at a higher abstraction level, synthesizing insights from various individual decisions to ensure safety and compliance with traffic norms.	Typically formulated based on real-time data from sensors and pre-defined rules focusing on immediate surroundings.

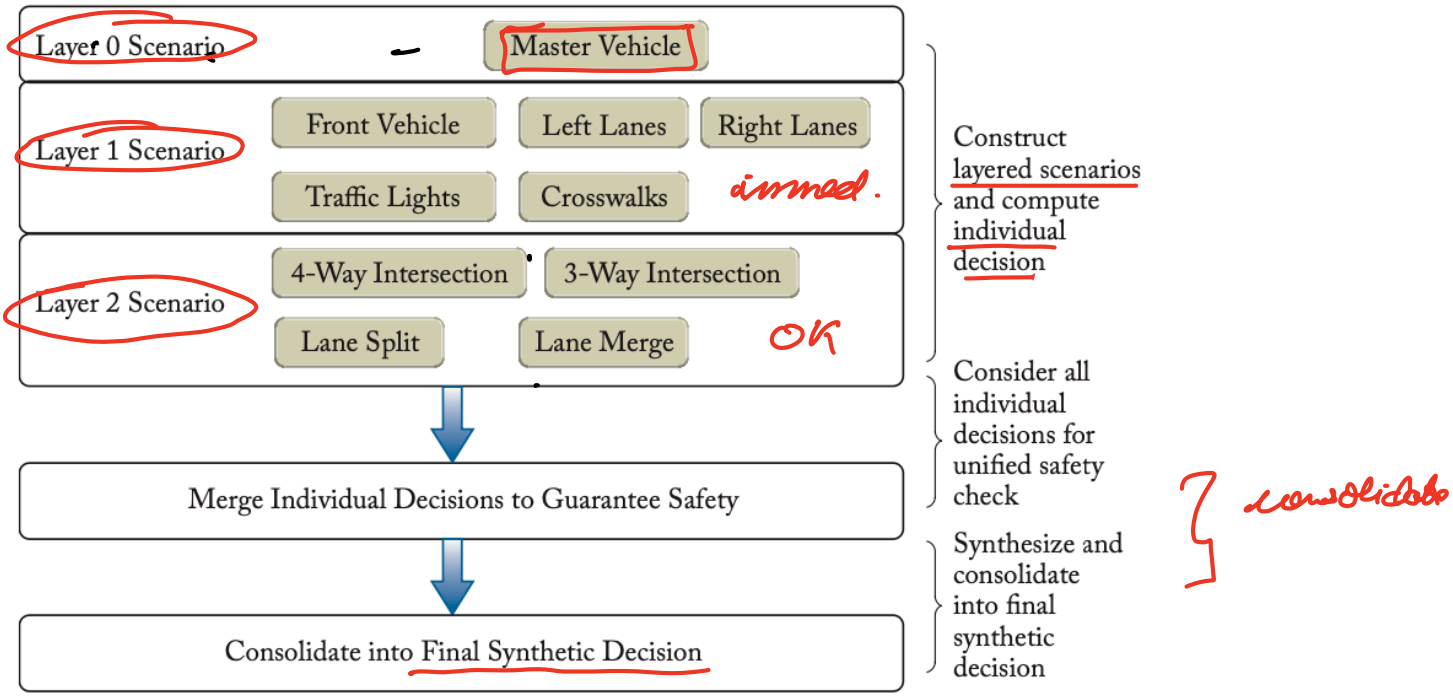


Figure 6.4: Architecture of a rule-based behavior decision system containing layered scenarios.

Motion Planning

The **motion planning** module is responsible for generating a trajectory for an autonomous vehicle and sending it to the feedback control system for execution. It defines a **spatial-temporal trajectory** as a sequence of points containing attributes like position, time, speed, and curvature.

Motion Planning as an Optimization Problem

Motion planning in autonomous vehicles is simpler than general robotics due to predefined road networks and constraints on vehicle movement (throttle, brake, steering). The optimization problem consists of:

1. **Optimization Object:** Finding the trajectory with minimal cost while ensuring collision avoidance and adherence to road shape.
2. **Constraints:** Ensuring smooth movement within feasible curvature and acceleration limits.

Approaches to Motion Planning

Two major approaches exist:

1. **Path and Speed Planning (Divide and Conquer):**
   - Path planning determines a trajectory shape on the 2D plane.
   - Speed planning determines how the vehicle should traverse the path.
   - Works well for **urban low-speed** driving.
2. **SL-Coordinate System-Based Planning:**
   - Considers **longitudinal (s-direction) and lateral (l-direction) movements** separately.
   - Naturally integrates speed into trajectory shape.
   - More suitable for **high-speed highway** driving.

Vehicle and Road Modeling

A **vehicle's pose** is represented as:

$\bar{x} = (x, y, \theta, \kappa, v)$

where:

- (x, y) is the position,
- θ is the direction,
- κ is the curvature,
- v is the speed.

The **SL-coordinate system** defines the road as a set of longitudinal (s) and lateral (l) positions using a **high-definition map (HD-map)** and lane reference lines.

Key Equations

The pose variables satisfy:

Vehicle :

$x, y \rightarrow$ 2D

θ : direction of motion

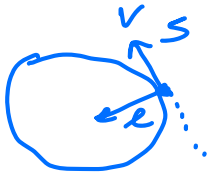
κ : curvature $\left(d\theta/ds\right)$

v : tangential velocity

- $\dot{x} = v\cos\theta$
- $\dot{y} = v\sin\theta$
- $\dot{\theta} = v\kappa$

SL coordinate :

high speed driving



l : lateral ($\perp$)

s : Longitude.

$$x(s,l) = x_{ref}(s) + l\cos\left(\theta_{ref}(s) + \pi/2\right)$$

$$y(s,l) = y_{ref}(s) + l\sin\left(\theta_{ref}(s) + \pi/2\right)$$

HD map + reference lines

Path & Speed Planning

divide & conquer

1. Path Planning

- Defines a vehicle's path as a continuous mapping from the range [0,1][0,1] to a set of vehicle poses.
- The goal is to find an optimal path from an initial pose to a desired end pose while minimizing cost and satisfying constraints.
- Uses **piece-wise polynomial splines** (cubic or quintic spirals) to connect sampled trajectory points.
- Candidate paths are generated by segmenting the lane into grids, with **256 possible paths** (in a 4×4 grid).
- **Dynamic Programming** is applied to find the minimum cost path by treating the problem as a search on a directed weighted graph.
- **Cost Function Considerations:**
  - **Road Alignment**: Paths close to the lane's central reference line have lower costs.
  - **Obstacle Avoidance**: Collision-prone grid points have high costs.
  - **Smoothness**: Ensures minimal curvature changes for driving comfort and control feasibility.

2. Speed Planning with ST-Graph

- Determines the speed at which a vehicle moves along the planned path while respecting physical and safety constraints.
- Uses an **ST-Graph** (Speed-Time Graph), where:
  - **S-axis** represents the longitudinal distance traveled.
  - **T-axis** represents time.
- Obstacles (e.g., other vehicles) are mapped onto the ST-Graph as time-dependent occupied regions.
- **Three speed planning strategies:**
  1. **Speed Plan 1**: Yielding—staying behind all vehicles.
  2. **Speed Plan 2**: Overtaking one vehicle while yielding to another.
  3. **Speed Plan 3**: Overtaking all vehicles.
- **Behavioral decision-making** influences cost assignment to different regions of the ST-Graph, guiding the search algorithm to favor safer, feasible paths.
- **Cost Considerations for Speed Planning:**
  - **Collision avoidance**: High costs assigned to risky grid areas.
  - **Acceleration constraints**: High costs for rapid acceleration changes to ensure control stability.

This two-step approach—path planning followed by speed planning—enables safe, efficient, and optimal motion planning for autonomous vehicles.

Motion Planning with Longitudinal and Lateral Planning

Instead of separating path planning and speed planning, motion planning can be approached using **longitudinal and lateral planning** within the SL-coordinate system.

1. Lateral Planning

- Focuses on movement in the **l-dimension** (sideways movement).
- Uses a **quintic polynomial** to ensure smooth motion by minimizing jerk.
- Start state P0=[d0,d˙0,d¨0]P_0 = [d_0, \dot{d}_0, \ddot{d}_0] is determined by the previous frame's end state.
- Candidate lateral end states d1d_1 are evaluated with **zero velocity and acceleration**, ensuring the vehicle moves parallel to the lane.
- **Cost Function Considerations:**
  - Penalizes slow convergence.
  - Prefers minimal lateral offset from the center of the lane.
  - Ensures consistency with behavioral decisions and collision-free paths.
- **Low-Speed Adjustments:**
  - At low speeds, lateral planning must depend on longitudinal motion due to vehicle constraints.
  - The trajectory is then optimized as a function of longitudinal position ss instead of time tt.

2. Longitudinal Planning

- Focuses on movement in the **s-dimension** (forward motion).
- The target trajectory starget(t)s_{\text{target}}(t) depends on the driving goal:
  - **Following a vehicle**: Maintains a safe distance and time gap from the lead vehicle.
  - **Lane changing (yielding/overtaking)**: Adjusts trajectory to avoid obstacles or overtake another vehicle.
  - **Stopping**: Brings the vehicle to a halt at a fixed position with zero speed and acceleration.
- Uses a **quartic or quintic polynomial** depending on whether speed or position is the priority.

3. Selecting the Optimal Trajectory

- **Combines** longitudinal and lateral candidate trajectories into all possible motion plans.
- Filters out **infeasible trajectories** based on:
  - Behavioral decision constraints.
  - Collision risks.
  - Control feasibility.
- The final trajectory is chosen based on **minimum weighted cost** across both dimensions.

This method provides **smooth, efficient motion planning** while maintaining flexibility for different driving scenarios.

Feedback Control in Autonomous Driving

6.3 Feedback Control

Feedback control

Feedback control in autonomous driving is similar to general mechanical control, where the system continuously tracks the difference between the actual and desired trajectory. Existing research has incorporated obstacle avoidance and route optimization into feedback control. Since traditional vehicle pose feedback control is well-established, this section focuses on two key concepts: the **Bicycle Model** and **PID Feedback Control**.

6.3.1 Bicycle Model

The **Bicycle Model** is a simplified vehicle model used in feedback control. It assumes:

- The vehicle is a rigid body with a front and rear axle.
- The **front wheels can rotate**, while the **rear wheels remain parallel** to the body.
- Vehicles **cannot move laterally without forward movement** (non-holonomic constraint).
- Inertial and tire slip effects are neglected at low speeds but become significant at high speeds.

The model represents the vehicle's pose using:

- Position: (x,y)(x, y)
- Heading angle: θ\theta
- Steering wheel angle: δ\delta

Equations describe the relationship between the front and rear wheel velocities, constrained by the vehicle's geometry. A **Unicycle Model** is often used as a further simplification, reducing control to steering angle change rate ω\omega and throttle/brake percentage.

6.3.2 PID Control

PID (Proportional-Integral-Derivative) Control is widely used for trajectory tracking. The **error function** e(t)e(t) represents the difference between the actual and desired vehicle pose. The PID system consists of:

- **P (Proportional) Control:** Direct response to error (KpK_p)
- **I (Integral) Control:** Corrects accumulated past errors (KIK_I)
- **D (Derivative) Control:** Predicts future errors (KDK_D)

Two PID controllers manage:

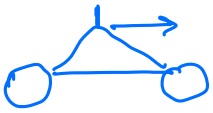
1. **Steering Wheel Angle (δ\delta)** – Adjusts based on heading error (θe\theta_e) and lateral error (lel_e).
2. **Forward Speed (VsV_s)** – Adjusts throttle/brake based on speed error.

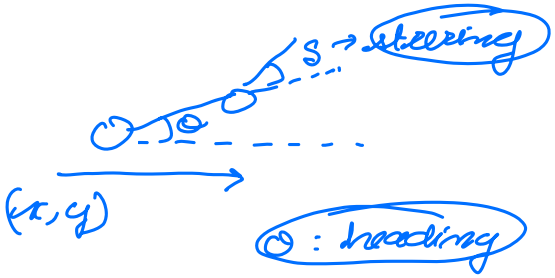
These controllers help autonomous vehicles closely follow a planned trajectory. More advanced control systems are needed for smoother motion and better passenger experience.

① Bicycle Model :

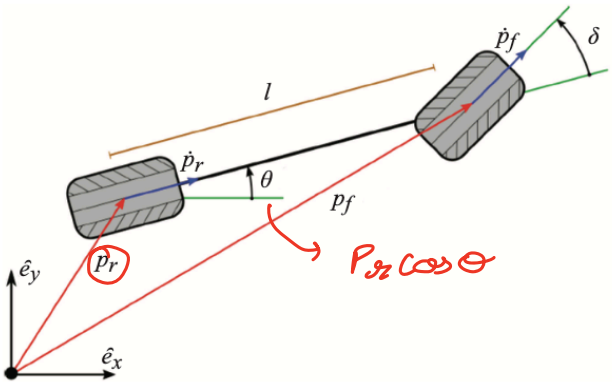
→ control of vehicle dynamics
→ vehicle into 2D

① vehicle as bicycle





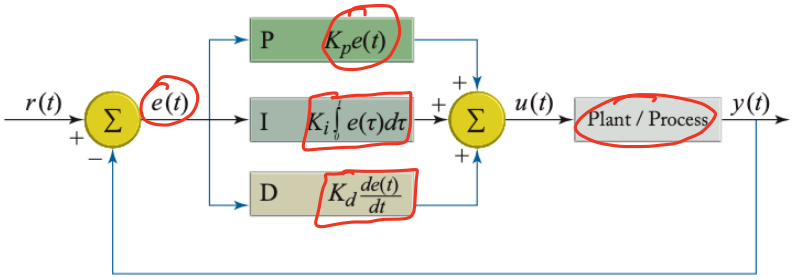
① no side side movement
② have to move →
- rotational motion
- fwd motion



→ ~~effect of~~ front wheel on rear wheel

② PID (Proportional, integral & derivate)

→ automatic control system
→ error value b/w desired & processed



$e(t) = y(t) - r(t)$

P K_P
I K_I
D K_D

P: → present error
→ ⇢ adjust.

↗ speed
↘ steering

I: → sums up past errors
→ accumulate correction (long term consistency)

D: → rate of change of error
→ strong corrective action to stabilize vehicle quickly

Unit-4 :

Reinforcement Learning-Based Planning and Control

Traditional planning and control for autonomous driving rely on optimization-based methods, including routing, traffic prediction, behavioral decision-making, motion planning, and feedback control. While these methods have been effective, learning-based approaches, particularly reinforcement learning (RL), are gaining interest.

Three key reasons highlight the need for RL in autonomous driving:

1. **Early-Stage Autonomous Driving** – Current systems are tested in controlled environments, and optimization-based methods may not handle real-world, unrestricted urban scenarios effectively.
2. **Limited Use of Historical Data** – Traditional optimization approaches rely on heuristics and do not fully utilize vast amounts of driving data, whereas RL can leverage this data to handle complex scenarios.
3. **Human Learning Analogy** – Human drivers learn through feedback from experience rather than direct optimization, making RL a more natural fit for autonomous driving.

The chapter covers the fundamentals of RL, including Q-learning and Actor-Critic methods, and explores RL applications in various layers of autonomous vehicle planning and control. While RL has shown promise in different driving scenarios, fully solving autonomous vehicle planning in unrestricted urban environments remains an open challenge.

Reinforcement Learning

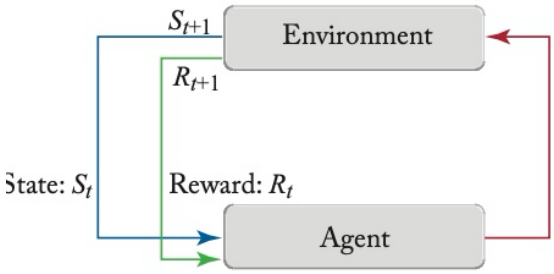
Reinforcement Learning (RL) is an interactive learning process where an **agent** interacts with an **environment**, taking actions, sensing states, and receiving rewards iteratively. The **goal** of the agent is to learn a policy that maximizes cumulative rewards over time.

Key Concepts:

1. **Agent and Environment** – The agent makes decisions based on the state (**St**) of the environment and receives rewards (**Rt**) as feedback.
2. **Reward Function** – Determines the desirability of state transitions, helping the agent optimize its long-term behavior.
3. **Returns** – The sum of rewards over time, which can be finite (episodic tasks) or infinite (continuing tasks).
4. **Discount Factor (γ)** – A parameter ($0 \leq \gamma \leq 1$) that determines how future rewards are valued. A higher γ gives more importance to future rewards.
5. **Policy (π)** – A strategy mapping states to actions to maximize expected returns.
6. **Value Function (Vπ)** – Measures the expected return from a given state when following a policy.
7. **Q-Value Function (Qπ)** – Measures the expected return of taking a specific action in a given state.
8. **Bellman Equation** – A recursive equation used to compute the value function efficiently in a Markov Decision Process (MDP).

The chapter sets the foundation for understanding RL by explaining its mathematical framework and principles, which are essential for optimizing decision-making in dynamic environments like autonomous driving.

→ closed loop learning process
→ define all params



agent takes A_t in env.
- $S_t \rightarrow S_{t+1}$
- $R_t \rightarrow R_{t+1}$

① agent – decision maker
② env
③ state – current situation, → State Space : all states
④ action – affects the state of env. (transition)
⑤ Reward (R) – profit / value for some goodwill / badwill
⑥ Policy (π) – decides which action to be taken in a state
probabilistic, deterministic, set of rules
⑦ Value function – How good is my current state, considering future rewards
⑧ Action value Q (function) – worth of taking particular action in current state + future + current state
⑨ return – cummulative reward expected, starting from current state, (G_t)
– factors immediate rewards & discounted
– long term decision making
⑩ discount factor – importance given to future rewards
⑪ learning rate – overriding old info with new
⑫ exploration – ε-greedy

Q learning Method +DQN

Actor-Critic Method

Q-learning

→ type of RL algo

→ how to act optimally in a given environment by learning [ACTION-VALUE] function

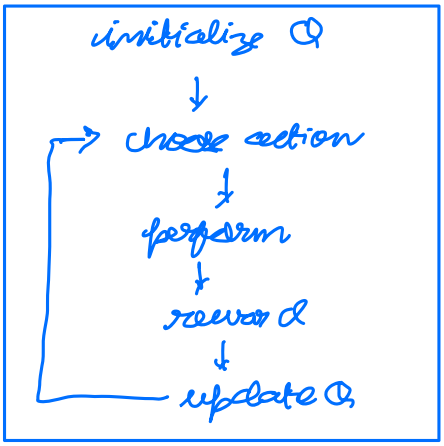
state-action to rewards

$$Q(s,a) = r + \gamma(\max_{a'} Q(s',a'))$$

- r: immediate reward
- $\max_{a'} Q(s',a')$: future reward

predict best action a in state s to maximize cumulative reward

→ update using bellman equations.



Action →

State	0	1	2	3	4
0					
1					
2					
3					
4					

R =

→ it is not explorative

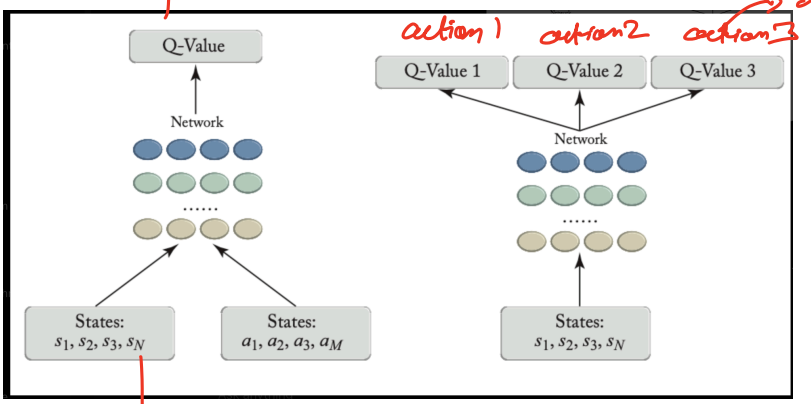
→ at some take random action

DATA IS TOO LARGE

↓

DQN (deep q-learning)

single Q-value



→ multiple Q-values

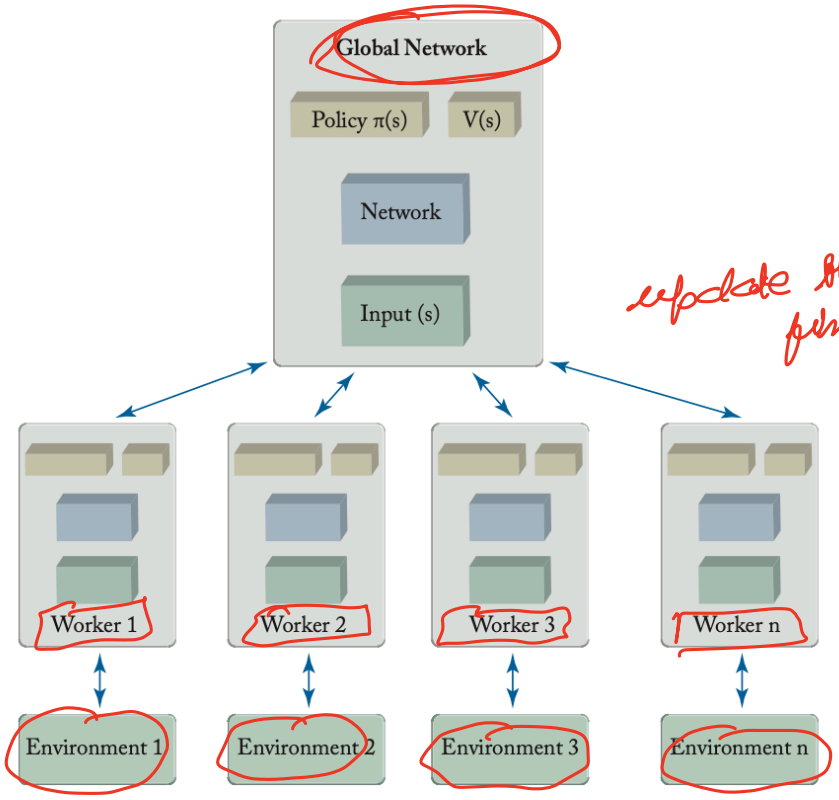
complex decision multiple actions

→ how good different actions on state are.

states

→ decision in binary 0/1

Aspect	Q-Learning	Deep Q-Learning (DQN)
Basic Concept	A reinforcement learning algorithm that uses a tabular method for learning the optimal policy.	An extension of Q-learning that uses a deep neural network to approximate the Q-value function instead of a table.
Data Structure	Utilizes a Q-table where each entry represents a state-action pair and its corresponding Q-value.	Uses a neural network to approximate the Q-values for all actions given a state, without storing them explicitly in a table.
Scalability	Works well with small, discrete state and action spaces due to the tabular approach.	Scales efficiently to problems with large or continuous state spaces due to the function approximation capability of neural networks.
Computation	Computationally less intensive for smaller problems but becomes infeasible as the state-action space grows.	Computationally demanding due to the need for training deep neural networks, but handles larger state spaces effectively.
Generalization	Limited generalization; each state-action pair must be visited and updated.	Better generalization; can predict Q-values for unseen states through learned representations.
Efficiency	Can be slow to converge in environments with many states because it must update the Q-table iteratively.	Potentially faster convergence in complex environments by generalizing over similar states.
Practical Usage	More suited to smaller, well-defined problems.	More suited for complex problems with high-dimensional state spaces, such as video games or autonomous driving.
Memory Usage	Requires memory proportional to the number of state-action pairs, which can be substantial.	Requires substantial memory for the neural network, but does not scale with the number of state-action pairs directly.
Learning Process	Every state-action pair is updated individually based on experience.	Global updates are applied to the network parameters based on batches of experience, impacting all state-action evaluations.



→ multiple agents learn simultaneously unique environment.

update their findings

Actor: best action

Critic: evaluator / feedback mechanism

Figure 7.6: Asynchronous Advantage Actor-Critic Framework (based on [11]).

Actor-Critic Methods (A3C Algorithm)

Overview of A3C

The **Asynchronous Advantage Actor-Critic (A3C)** algorithm, introduced by DeepMind, improves upon Deep-Q-Learning (DQN) by using multiple independent learning agents (worker agents) that train asynchronously in their own environments while sharing a **global network**. This approach enhances learning efficiency, stability, and robustness.

Key Steps in A3C Training:

1. **Initialize** a global network and sync all worker agents to it.
2. Each **worker agent** interacts with its **own environment**, gathering experience.
3. Each **worker computes its loss function** separately.
4. Workers **update their gradients** based on computed losses.
5. Workers **synchronously update the global network**, and training continues.

Actor-Critic Framework:

A3C uses two separate but interconnected **neural network branches**:

- **Actor (Policy Network π(s))**: Estimates the best action to take in a given state.
- **Critic (Value Network V(s))**: Estimates the expected reward (value function) for a given state.

The **critic** guides the **actor** by estimating how much better an action is than expected, using the **advantage function**:

$$A(s,a) = Q(s,a) - V(s)$$

where $Q(s,a)$ is the expected return, approximated by the discounted return R.

Loss Functions in A3C:

- **Value loss** (for critic): $L_{value} = \sum (R - V(s))^2$
- **Policy loss** (for actor): $L_{policy} = A(s) * \log(\pi(s)) + H_{\pi} * \beta$ where **Hπ** is the entropy term to balance **exploration** (high entropy) and **exploitation** (low entropy).

Advantages of A3C over DQN:

- **Multiple parallel agents** speed up learning.
- **More diverse experiences** improve generalization.
- **Better sample efficiency** through asynchronous updates.
- **Handles continuous action spaces** more effectively.

A3C is widely used in **robotics, autonomous control, and game AI** due to its ability to efficiently learn optimal policies in complex environments.

Learning-Based Planning and Control in Autonomous Driving

Overview

Reinforcement Learning (RL) has been applied to various levels of **autonomous driving planning and control**, including:

1. **Behavioral Decision** – High-level decision-making for traffic scenarios.
2. **Motion Planning** – Determining spatial-temporal trajectories.
3. **Feedback Control** – Low-level execution of control signals.

While **end-to-end deep learning** methods (sensor data to direct control signals) exist, they suffer from **complexity and lack of explainability**. RL-based approaches offer an alternative, focusing on structured decision-making in different planning layers.

7.3.1 Reinforcement Learning for Behavioral Decision

Traditional **rule-based** behavioral decision-making struggles with unpredictable, **long-tail traffic scenarios**. RL provides a more adaptable, human-like decision-making system by learning from driving experiences.

Key Aspects of RL-based Behavioral Decision:

- **Action Space (Desires):**

  D=[0,vmax]×L×{g,t,o}nD = [0,v_{max}] \times L \times \{g, t, o\}^n

  where:

  - vmaxv_{max} = target speed
  - LL = lateral lane positions
  - g,t,og, t, o = yield (give way), overtake, maintain offset distance (nudge/attention)
- **State Space:**
  - **"Environment model"** created from sensory data.
  - Kinematic history of surrounding vehicles.
- **Key Contribution:**
  - **Does not require Markov property**, enabling policy gradient optimization without strict state transition assumptions.
  - Initialized using **imitation learning** and refined through **iterative policy gradient updates**.

Although implementation details remain proprietary, RL-based behavioral decision-making is promising for **handling complex real-world traffic interactions** beyond rigid rule-following approaches.

Aspect	Behavioral Decision	Motion Planning and Control
Objective	To enable the vehicle to make smart decisions in complex, dynamic traffic situations.	To calculate and execute a safe and efficient path or trajectory for the vehicle.
What it Involves	Deciding actions like changing lanes, adjusting speed, or stopping based on traffic conditions and other road users.	Determining the specific route and maneuvers the vehicle should take from point A to B, considering obstacles.
Challenges	Requires understanding a wide range of possible traffic interactions and environmental variables.	Involves continuous assessment of vehicle dynamics and environmental conditions to make precise movements.
State Space	Includes data about surrounding vehicles, road conditions, traffic signals, and vehicle's current state.	More detailed; includes not only immediate environmental data but also the vehicle's planned future states.
Action Space	Typically includes discrete decisions like lane changes, acceleration changes, or braking.	Involves more granular controls such as steering angles, speed adjustments over time, and navigating through turns.
Use of RL	RL helps the vehicle learn from past driving experiences, improving decision-making in similar future scenarios.	RL is used to optimize the path dynamically, learning to adjust the planned trajectory in response to real-time changes.
Example	An RL system might learn to change lanes in heavy traffic to maintain optimal speed and safety.	An RL system might plan and adjust a path through a busy intersection, considering the timing of traffic lights and the movement of other vehicles.
Benefits	Enhances adaptability to unpredictable driving environments and improves safety and efficiency.	Ensures smooth, efficient, and safe navigation, adjusting dynamically to new obstacles and changes in the road layout.
Data Dependency	Heavily relies on real-time and historical data to predict and react to dynamic scenarios effectively.	Requires precise, real-time data processing to continuously refine vehicle trajectory and movements.
Feedback Loop	Uses feedback from the vehicle's actions to refine future decisions, enhancing responsiveness to similar situations.	Employs feedback from vehicle dynamics and environmental interactions to refine path calculations continuously.
Learning Speed	Faster initial learning due to discrete nature of decision-making but requires ongoing refinement as scenarios evolve.	Slower initial learning due to complexity of inputs and required precision, but highly effective once optimized.
Complexity	Lower computational complexity compared to motion planning, but high complexity in scenario classification.	Higher computational complexity due to the need for continuous data integration and response adjustments.
Autonomy Level	Critical for achieving high levels of autonomy where vehicles must make independent complex decisions.	Essential for full autonomy, enabling vehicles to navigate without human intervention even in dynamic environments.
Safety Impact	Directly impacts the immediate safety of the vehicle and its surroundings by preventing potential collisions.	Influences long-term route efficiency and safety, optimizing overall travel time and energy consumption.

Reinforcement Learning on Planning and Control in Autonomous Driving

Key Challenges in RL-based Planning and Control

- **State Space Design:**
  - Must incorporate **autonomous vehicle state + surrounding environment**.
  - Large multi-dimensional continuous space needs efficient representation.
  - **Cell-mapping techniques** help discretize state space to simplify control problems.

Reinforcement Learning Approaches

1. Cell-Mapping Q-Learning for Car-Like Vehicles (CLV) [2]

- **State Space (Before Cell-Mapping):** Includes **velocity, Cartesian coordinates (x, y), and orientation (θ)**.
- **Action Space:** Includes **traction motor voltage** and **steering angle**.
- **Approach:**
  - Uses **Control-Adjoining-Cell-Mapping (CACM)** to reduce state space complexity.
  - Implements **Q-learning with a table-based approach**, maintaining a **Q-table** and a **model-table** (for averaging local transitions).
  - Focuses purely on **exploration**, making exploitation unnecessary.
  - **Obstacle avoidance** is handled via a separate safety function rather than as a state variable, limiting robustness against dynamic obstacles.

2. RNN-based Adaptive Cruise Control & Traffic Circle Merging [1]

- **Predicts short-term future using supervised learning**:
  - Learns a function N^(st,at)≈st+1N̂ (s_t, a_t) \approx s_{t+1} for state transitions.
- **Uses reinforcement learning (RNN-based policy function)** to handle unpredictable environmental changes.
- **Approach balances prediction with adaptability** to varying traffic conditions.

ML + RNN (RL)

Unsolved Problems & Challenges

1. **State Space Representation:**
   - **Including dynamic obstacles and road structure** in a structured manner.
   - Possible solution: **SL-coordinate grid-based state representation** (inspired by motion planning).
2. **Reward Function Design:**
   - Categorical behavior decisions are easier to reward than **continuous motion planning and control signals**.
   - Motion planning reward design should **consider goal-reaching, obstacle avoidance, and ride comfort**.
   - Designing rewards at the **feedback control signal level** is easier but shifts the problem from **optimal trajectory generation to trajectory execution**.
3. **Balancing RL and Optimization-Based Control:**

- **Trajectory execution** (low-level control) can be solved using traditional optimization.
- **Motion planning** (high-level decision-making) benefits more from RL.
- Future research should explore **RL for motion planning** to improve trajectory optimization.

Conclusion

Reinforcement learning in **planning and control** is promising but faces challenges in **state space representation, reward function design, and integration with optimization techniques**. More research is needed, particularly in **RL-based motion planning for spatial-temporal trajectory generation**.

Client systems - 8 Chapter 4 P2.

Autonomous Driving as a Complex System

Overview

Autonomous driving is a highly **complex system** that integrates multiple technologies to enable real-time decision-making in urban environments. It involves:

- **Sensing**: GPS/IMU, LiDAR, and cameras collect environmental data.
- **Perception**: Object recognition and tracking analyze sensor data.
- **Localization**: Determines the vehicle's position.
- **Planning & Control**: Includes prediction, path planning, and obstacle avoidance.

Client System for Autonomous Driving

1. Hardware Platform

- Follows a **sensing → perception → action computing** paradigm.
- Sensors collect real-world data, which is processed for decision-making.
- Action plans are executed via control systems.

2. Role of the Operating System

- Manages **communication and resource allocation** among components.
- Ensures real-time processing, e.g., **camera delivering 60 FPS (≤16 ms per frame)**.
- **Resource contention issues**:
  - Large LiDAR data bursts can overwhelm the CPU, causing **dropped camera frames**.
  - The OS must **allocate resources efficiently** to prevent failures.

Conclusion

Autonomous driving **relies on seamless integration** of hardware and software. A well-designed **operating system** is crucial for managing real-time tasks and **optimizing resource allocation** to ensure reliable performance in unpredictable urban environments.

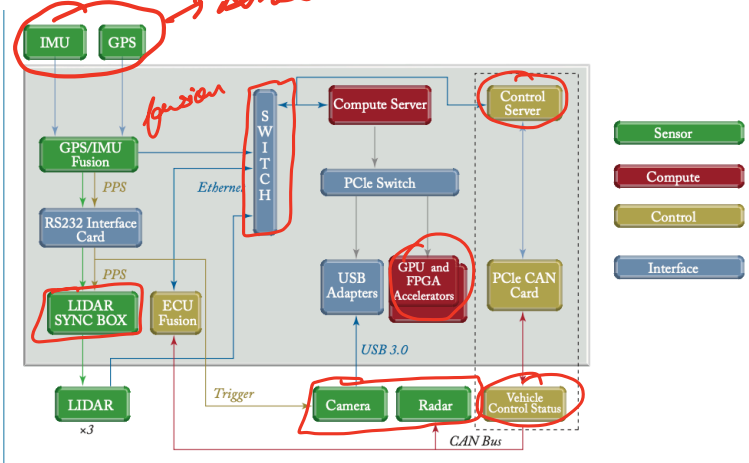
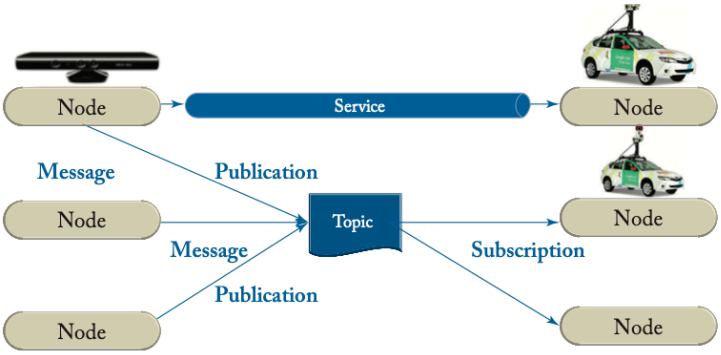


Figure 8.2: Hardware platform for autonomous driving.

ROS: → manage all components
→ communication
→ resource allocation } →
→ set of software libraries

3 → master
→ node
→ service

ROS2 → DDS middleware
reliable communication
↘ shared memory



→ nodes
→ service
→ master
→ Topic → Pub
→ sub.
→ msg

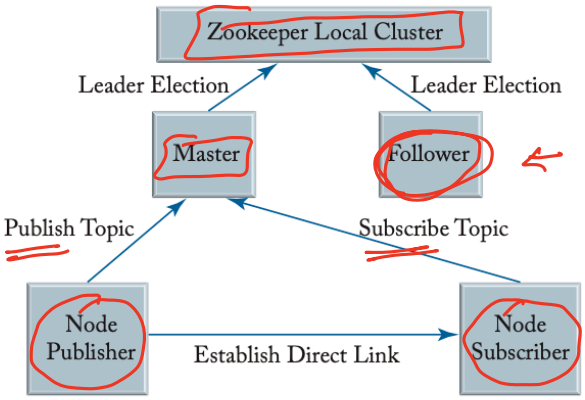
Figure 8.4: ROS communication mechanisms.

① Master: coordinator, establish communication, find each other
② Nodes: computation → sensing,
→ actuation
decision. } msg.
③ service: special type node to node
↳ immediate feedback

Communication Standards

① Publish / subscribe – unidirection, 1-many
② req / response – direct, synchronous, 1-1

System Reliability



① decentralization of master (Zookeeper)
② node monitoring
③ state management

← decentralize.

① use shared memory
② multicast > broadcast
③ lightweight serialization

① use of LXC } resource & security management
② sandbox

Computing Platforms for Autonomous Driving

Overview of Computing Platform

Autonomous vehicles generate up to 2 GB of raw sensor data every second, which needs to be processed in real time for perception and action planning. The computing platform plays a crucial role in meeting real-time performance and ensuring system robustness. Key challenges in platform design include cost, power consumption, and heat dissipation.

Computing Platform Implementation

An existing autonomous driving computing solution includes two compute boxes, each with:

- **Intel Xeon E5 processor (12 cores)** delivering 400 GOPS/s and consuming 400 W of power.
- **Nvidia K80 GPUs** providing 8 TOPS/s per GPU at 300 W power consumption. The system delivers 64.5 TOPS/s at 3,000 W power consumption, connected to 12 cameras and a LiDAR unit. To ensure reliability, a second compute box mirrors the tasks of the first. However, the high power consumption (over 5,000 W) and cost ($20,000–$30,000 per box) make this solution expensive and impractical for average consumers.

Existing Computing Solutions

1. **GPU-Based Solution**:
   - **Nvidia PX 2**: Composed of two Tegra SoCs and two Pascal GPUs, capable of performing 24 trillion deep-learning calculations per second. It processes 2,800 images/s with optimized I/O and DNN acceleration.
2. **DSP-Based Solution**:
   - **Texas Instruments TDA**: A SoC with DSP cores and Vision Accelerators designed for vision processing, offering significant acceleration with less power consumption compared to ARM Cortex-15 CPUs.
   - **CEVA XM4**: A DSP-based solution for energy-efficient computer vision, requiring less than 30 mW for 1080p video at 30 fps.
3. **FPGA-Based Solution**:
   - **Altera Cyclone V SoC**: Used in Audi products for sensor fusion, optimizing the integration of multiple sensors for reliable object detection.
   - **Zynq UltraScale MPSoC**: Achieves high efficiency in CNN tasks (14 images/sec/Watt) and object tracking (60 fps in live 1080p video).
4. **ASIC-Based Solution**:
   - **MobilEye EyeQ5**: Features four programmable accelerators optimized for computer vision, signal processing, and machine learning tasks. It enables system expansion with multiple EyeQ5 devices connected via PCI-E ports.

Conclusion

Various computing solutions, ranging from GPUs to FPGAs and ASICs, are available for autonomous driving, each offering trade-offs in terms of performance, power efficiency, and cost. The goal is to balance these factors to ensure the vehicle can process sensor data efficiently and reliably.

Computer Architecture Design for Autonomous Driving

Matching Workloads to Computing Units

Experiments on an ARM mobile SoC with a CPU, GPU, and DSP were conducted to determine the best computing units for various tasks in autonomous driving:

- **Convolution (Object Recognition/Tracking)**: The **GPU** proved to be the most efficient, completing tasks in 2 ms with low energy consumption (4.5 mJ).
- **Feature Extraction (Localization)**: The **DSP** was the most efficient for this task, completing in 4 ms and consuming 6 mJ.

For control-heavy tasks like localization and path planning, the **CPU** was used as it was not suitable for GPUs and DSPs.

Autonomous Driving on a Mobile Processor

A vision-based autonomous driving system was implemented on the ARM mobile SoC, utilizing:

- **DSP** for sensor data processing (e.g., feature extraction and optical flow).
- **GPU** for deep learning tasks (e.g., object recognition).
- **CPU** threads for localization, path planning, and obstacle avoidance.
- The system demonstrated decent performance:
  - **Localization**: 25 images/s.
  - **Deep Learning**: 2-3 object recognition tasks/s.
  - **Planning/Control**: Path planning within 6 ms.
  - Power consumption: 11 W.

This setup allowed the vehicle to drive at 5 mph without loss of localization. With more resources, the system could handle higher speeds and more complex tasks.

Design of Computing Platform

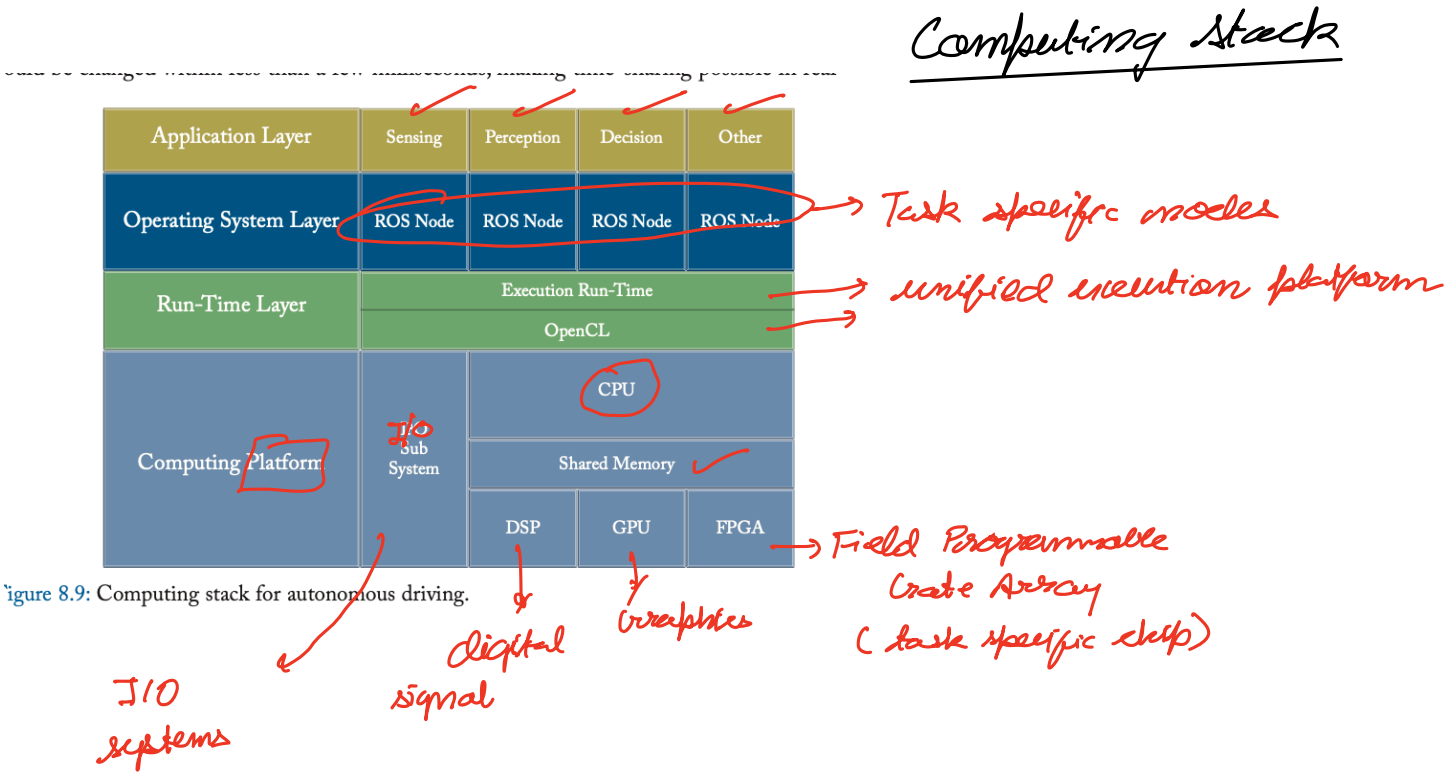
The success on the ARM mobile SoC was attributed to utilizing its heterogeneous computing resources optimally. However, not all tasks could be handled due to limited resources. Tasks like object tracking, traffic prediction, and data uploading were identified as non-continuous and could be time-shared on an **FPGA** for efficiency.

Proposed Computing Platform Design

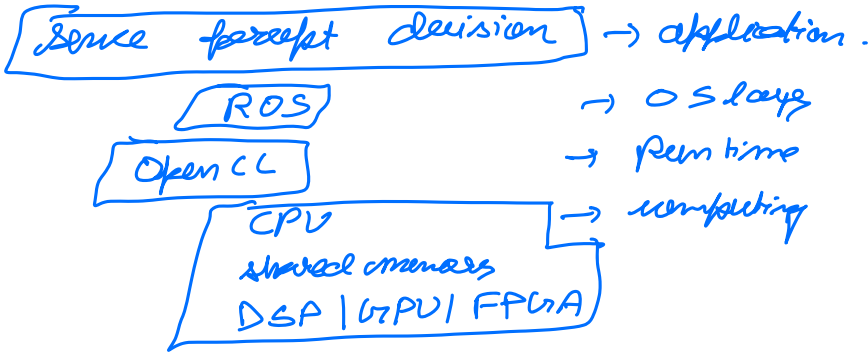
The proposed computing stack for autonomous driving includes:

- **SoC Architecture**: Includes components such as a DSP, GPU, multi-core CPU, and FPGA. These components handle tasks like feature extraction, object recognition, planning, and control.
- **FPGA**: Time-shared for tasks like object tracking and traffic prediction using Partial-Reconfiguration techniques.
- **Shared Memory**: Allows efficient communication between the different computing units.
- **Run-Time Layer**: Uses OpenCL to map workloads to the appropriate computing unit and schedules tasks at runtime.
- **Operating System Layer**: Built on the Robot Operating System (ROS), enabling a distributed system where each node handles a specific task in autonomous driving.

This design aims to balance performance, energy efficiency, and flexibility, ensuring that the platform can adapt to real-time needs in autonomous driving systems.



igure 8.9: Computing stack for autonomous driving.



1

Introduction to Deep Q-Learning (2 Marks)

Deep Q-Learning (DQL) is a model-free reinforcement learning algorithm where an agent learns an optimal policy by interacting with the environment and maximizing a reward function. It is useful for autonomous vehicle control as it enables decision-making in real-time traffic conditions.

- Uses a neural network to approximate the Q-value function.
- Learns through experience using a replay buffer to store past experiences.
- Utilizes an epsilon-greedy strategy for exploration and exploitation.

Steps for Implementing Deep Q-Learning in Autonomous Vehicles (7 Marks)

Step 1: Define the Environment and State Representation

- The environment is modeled using a simulator (e.g., CARLA or SUMO).
- The state includes sensor inputs (LIDAR, cameras, radar), velocity, lane position, and surrounding vehicles' locations.

Step 2: Define Actions and Reward Function

- **Actions**: Steering angle, acceleration, braking, lane changes.
- **Rewards**:
  - Positive rewards for staying in the lane, maintaining safe distances, and smooth driving.
  - Negative rewards for collisions, abrupt braking, or violating traffic rules.

Step 3: Train the Deep Q-Network (DQN)

- Use a Convolutional Neural Network (CNN) to process visual and sensor data.
- Train using a replay buffer and experience replay to improve learning stability.
- Use a target network to avoid instability in Q-value updates.

Step 4: Policy Optimization and Deployment

- Fine-tune the model using transfer learning from simulation to real-world scenarios.
- Deploy the trained model in real-world environments with additional safety constraints.

Real-World Example (1 Mark)

A real-world example is **Autonomous Highway Merging**.

7(a)	Evaluate the application of reinforcement learning (RL) in autonomous vehicle planning and control. Specifically, assess its effectiveness in scenarios such as adaptive route planning, decision-making under uncertain traffic conditions, and obstacle avoidance. Consider the advantages and limitations of using RL compared to traditional planning methods. Propose potential strategies to enhance RL algorithms for autonomous vehicles, focusing on real-time performance, safety, and the ability to generalize to diverse driving environments.	

8(a)	Examine the challenges and propose solutions for integrating client systems in autonomous driving platforms.	08

Hybrid Models: Combining RL with traditional methods can help mitigate some of its weaknesses. For example, basic rules can govern straightforward scenarios while allowing RL to handle complex situations, balancing performance with reliability.¶

Improved Exploration Techniques: Enhancing exploration methods such as epsilon-greedy or using advanced strategies like entropy maximization can help improve the learning speed and quality of the RL models.¶

Real-time Performance Optimization: Investing in computational optimizations and leveraging edge computing can help deploy RL algorithms that require intensive computations to run efficiently in real-time.¶

Safety Protocols: Implementing rigorous safety checks and fallback mechanisms to handle potential failures of RL systems is crucial. These protocols ensure that the vehicle can safely revert to a conservative mode of operation if unexpected behavior occurs.¶

Generalization Techniques: Developing techniques to improve the generalization of RL models across different environments is essential. Transfer learning and domain randomization are examples of approaches that can help RL systems perform well in diverse driving conditions.¶

Conclusion While RL holds substantial promise for enhancing autonomous vehicle planning and control, addressing its limitations through strategic improvements and hybrid approaches will be key to realizing its full potential in practical applications.¶

Evaluation of Reinforcement Learning in Autonomous Vehicle Planning and Control

1. Effectiveness in Adaptive Route Planning and Obstacle Avoidance

Adaptive Route Planning: Reinforcement learning (RL) excels in adaptive route planning by continuously learning from the vehicle's experiences to optimize paths in real-time. RL algorithms can dynamically adjust routes based on current traffic conditions, road closures, and other environmental variables. This ability is particularly valuable in scenarios where road conditions change unpredictably, requiring the vehicle to adapt quickly.

Obstacle Avoidance: In obstacle avoidance, RL can effectively process sensor data to make split-second decisions. By training on a variety of scenarios, an RL-equipped vehicle can learn to identify and react to obstacles such as pedestrians, animals, or unexpected debris on the road. The real-time processing capability of RL allows autonomous vehicles to learn from near-miss incidents to improve their detection and evasion strategies.

2. Advantages of RL Over Traditional Planning Methods

Adaptability: RL provides superior adaptability compared to traditional methods, which often rely on predefined rules and cannot easily adapt to new situations without reprogramming or manual updates.

Learning from Interaction: Unlike traditional methods that typically operate under programmed instructions, RL improves through trial and error across diverse scenarios, enhancing its decision-making capabilities over time based on direct interactions with the environment.

Optimization Capabilities: RL continuously refines its strategies to maximize the cumulative reward, potentially leading to more efficient and safer driving strategies than those derived from static algorithms.

3. Limitations of RL Compared to Traditional Methods

Data and Computation Intensive: RL requires significant amounts of data and computational resources to train effective models, which can be a limitation in resource-constrained environments.

Convergence and Stability Issues: RL algorithms can suffer from slow convergence and stability issues during the training phase, especially in environments with high variability.

Lack of Interpretability: RL decisions are often based on complex models like neural networks, which can be difficult to interpret compared to rule-based systems where decisions are traceable and explainable.

4. Strategies to Enhance RL Algorithms for Autonomous Vehicles

Integration with Simulation Environments: Using advanced simulation environments can help in training RL algorithms under a wide range of scenarios, including rare and dangerous situations that are not frequently encountered in real life.

Challenges

1. **Compatibility and Interoperability:**
   - **Issue:** Different client systems may use varied communication protocols and data formats, complicating their integration into a unified autonomous driving platform.
   - **Impact:** This lack of standardization can lead to integration issues, reducing the system's overall efficiency and performance.
2. **Real-Time Data Processing:**
   - **Issue:** Client systems in autonomous vehicles often require real-time data processing to react promptly to road conditions.
   - **Impact:** Delays in data processing can lead to missed cues or delayed reactions, compromising vehicle safety.
3. **Scalability:**
   - **Issue:** As the autonomous driving ecosystem grows, the platform must efficiently scale to accommodate more client systems without performance degradation.
   - **Impact:** Poor scalability can limit the expansion and adoption of autonomous driving technologies.
4. **Security and Privacy:**
   - **Issue:** Integrating multiple client systems increases the risk of cyber threats and data breaches.
   - **Impact:** Security vulnerabilities can expose sensitive user data and critical system operations to attackers, posing safety risks.
5. **Software and Hardware Updates:**
   - **Issue:** Ensuring that all integrated client systems are consistently updated with the latest software and hardware specifications is challenging.
   - **Impact:** Outdated components may not function optimally, leading to system inefficiencies or failures.

Proposed Solutions

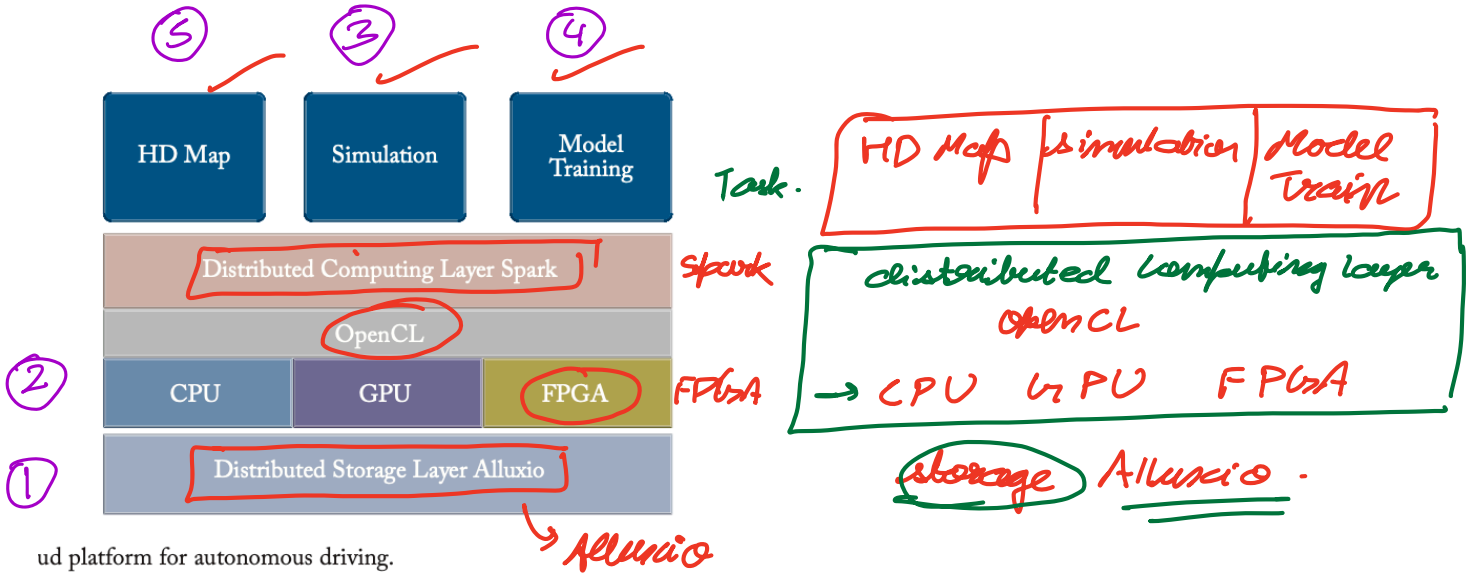
1. **Developing Universal Standards and Protocols:**
   - **Approach:** Work with industry groups to develop and adopt universal communication protocols and data standards for client systems in autonomous vehicles.
   - **Benefit:** Ensures that different systems can communicate seamlessly, enhancing compatibility and easing integration.
2. **Enhanced Real-Time Processing Capabilities:**
   - **Approach:** Invest in advanced computing hardware and optimize software algorithms to handle real-time data processing efficiently.

   - **Benefit:** Reduces latency and accelerates response times, improving vehicle safety and operational efficiency.
3. **Modular Architecture:**
   - **Approach:** Design the autonomous driving platform using a modular architecture where client systems can be plugged in or updated independently.
   - **Benefit:** Facilitates easier scalability and maintenance, allowing for the smooth integration of new technologies and upgrades.
4. **Robust Security Framework:**
   - **Approach:** Implement multi-layered security protocols, including encryption, secure boot, and intrusion detection systems, specifically tailored for integrated client systems.
   - **Benefit:** Enhances data security and privacy, protecting against unauthorized access and cyber threats.
5. **Regular Software and Firmware Updates:**
   - **Approach:** Establish a centralized management system for regular and automated updates across all client systems.
   - **Benefit:** Ensures all components remain up-to-date with the latest security patches and performance upgrades, maintaining system integrity.
6. **Continuous Testing and Validation:**
   - **Approach:** Regularly conduct comprehensive testing and validation of integrated systems to ensure compatibility and performance under various conditions.
   - **Benefit:** Identifies potential integration issues early, allowing for timely adjustments and ensuring reliable system performance.

Unit-5 Cloud P

Introduction

1. **Introduction to Autonomous Driving Cloud Infrastructure**:
   - Autonomous vehicles generate over 2GB of raw sensor data per second.
   - Efficient cloud infrastructure is necessary for storing, processing, and utilizing this data.
   - Cloud infrastructure supports tasks like simulation tests, offline deep learning model training, and HD map generation.
2. **Key Cloud Computing Applications**:
   - Applications include simulation tests, HD map generation, and offline model training.
   - These require distributed computing and storage support.
3. **Challenges with Tailored Infrastructures**:
   - Lack of dynamic resource sharing: Each infrastructure tailored to one application limits the ability to interchange resources.
   - Performance degradation: Data sharing across applications is inefficient without a unified infrastructure, causing performance overhead.
   - Management overhead: Specialized infrastructures require extensive engineering efforts to maintain.
4. **Unified Cloud Infrastructure Solution**:
   - The infrastructure is unified to provide both distributed computing and storage.
   - A heterogeneous computing layer using GPUs and FPGAs improves performance and energy efficiency.
5. **Components of the Cloud Platform**:
   - **Distributed Computing Layer**: Uses Spark.
   - **Heterogeneous Computing Layer**: Uses OpenCL for GPU/FPGA acceleration.
   - **Distributed Storage Layer**: Uses Alluxio for in-memory storage.
6. **Benefits**:
   - Reliable, low-latency, and high-throughput cloud platform for autonomous driving.



ud platform for autonomous driving.

Spark: low latency, high throughput, RT tasks

openCL, CPU, GPU, FPGA → Heterogeneous computing

Alluxio: in memory storage - best I/O

Application.

9.2.1 Distributed Computing Framework

- **Options for Distributed Computing**: The choice was between Hadoop MapReduce and Apache Spark.
- **Spark's Advantages**: Spark is an in-memory distributed computing framework with low latency and high throughput, using RDDs (Resilient Distributed Datasets) to store data across clusters. It avoids MapReduce's disk-based structure, improving performance significantly.
- **Reliability Testing**: A 1,000-machine Spark cluster was stress-tested for three months, identifying bugs and ensuring reliability.
- **Performance Comparison**: Spark outperformed MapReduce by 5X in processing SQL queries, with Spark completing tasks much faster (150s vs 1,000s for MapReduce).

9.2.2 Distributed Storage

- **Options for Storage**: The decision was between Hadoop Distributed File System (HDFS) and Alluxio.
- **Alluxio's Memory-Centric Approach**: Alluxio uses memory as its default storage medium for faster read/write performance and supports tiered storage (memory, SSD, HDD) to handle larger data volumes.
- **Co-location with Compute Nodes**: Alluxio is co-located with compute nodes to exploit spatial locality, achieving a 30X speedup compared to using HDFS alone.

9.2.3 Heterogeneous Computing

- **Computing Substrates**: Spark typically uses CPUs, but GPUs and FPGAs can offer better performance for specific tasks (e.g., GPUs for image convolution tasks, FPGAs for low-power vector computation).
- **Challenges**:
  - Dynamically allocating computing resources (CPU, GPU, FPGA) for different workloads.
  - Seamlessly dispatching workloads to these resources.
- **Solution**:
  - **YARN and LXC**: YARN handles resource management and scheduling, while Linux Containers (LXC) enable resource isolation and virtualization with minimal overhead (less than 5% CPU overhead).
  - **Heterogeneous Computing RDD**: Developed to dispatch tasks from Spark to native spaces (GPU/FPGA) using the Java Native Interface (JNI).
  - **OpenCL for GPU/FPGA Tasks**: OpenCL is used to dispatch workloads to GPUs and FPGAs, optimizing performance for heterogeneous computing.

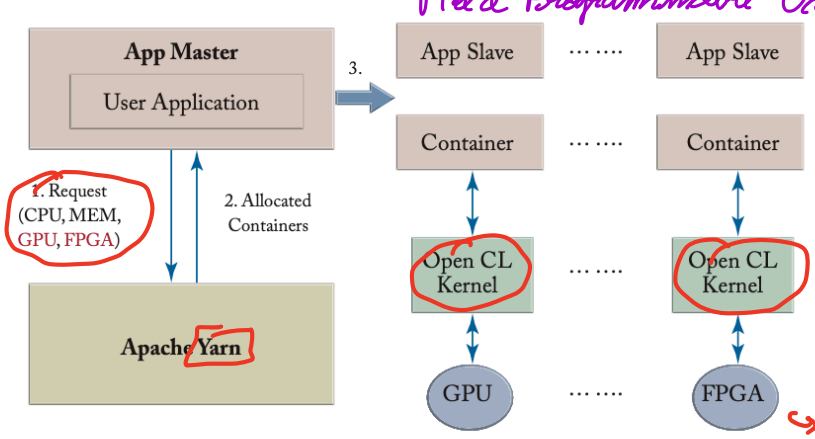


Figure 9.2: Distributed heterogeneous computing platform.

9.3 Simulation

- **Purpose**: Distributed simulation is used for testing new algorithms before deployment in real vehicles to minimize costs and ensure thorough testing.
- **Simulation with ROS**: The Robot Operating System (ROS) is used to replay data and verify algorithms. Simulation tests need to be scalable to cover enough data and reduce time.
- **Distributed Simulation with Spark**: The Spark infrastructure is used to build a distributed simulation platform, deploying algorithms on multiple compute nodes and aggregating the results.

9.3.1 BinPipeRDD

- **Problem**: Spark traditionally consumes structured text data, but ROS generates multimedia binary data (e.g., sensor readings, obstacle bounding boxes).
- **Solution**: BinPipeRDD was created to allow Spark to consume binary input streams. The process involves encoding, serialization, and deserialization of binary data into a format that Spark can process, while maintaining flexibility for user-defined tasks.
- **How It Works**: Binary files are processed into byte streams, which are then decoded, processed by user logic, and serialized back into byte streams for collection or storage in HDFS.

9.3.2 Connecting Spark and ROS

- **Integration**: Instead of changing ROS and Spark interfaces, ROS nodes and Spark executors are co-located and communicate via Linux pipes, which offer a simple, efficient data communication method.

9.3.3 Performance

- **Scalability Testing**:
  - A basic image feature extraction task on 1 million images showed significant scalability, with execution time dropping from 130s to 32s as CPU cores scaled from 2,000 to 10,000.
  - In simulation tests, scaling from 1 to 8 Spark nodes reduced the simulation time from 3 hours to 25 minutes, demonstrating strong scalability.

Model Training

→ DL models (train offline)

→ GPU acceleration, in memory storage, distributed computing

Why not single server?

- huge amount of data
- slow

→ SPARK
→ PADDLE (DL framework)
→ Alluxio (in memory storage)

→ in memory (RDD)
→ reduce I/O to disk
→ fast pipeline

Training Platform Architecture

① spark → driver, executor, manager

② Paddle trainer

③ Alluxio

Alluxio → Parameter storing

→ stores params
→ new average
→ continue train

model manager → spark executor, → paddle trainer.

data in !!

spark driver → manages & coordinates

CPU vs GPU

xLS

Scalable

CPU ↓ train time

HD MAP generation :

→ very detailed map → 3D, → lane, → traffic sign

Raw data → preprocessing →

9.4 Model Training

- **Purpose**: The infrastructure supports offline model training for autonomous driving, leveraging GPU acceleration and in-memory storage through parameter servers. The large volume of raw data necessitates distributed training to ensure fast model development.

9.4.1 Why Use Spark?

- **Challenge**: Model training requires significant data preprocessing (e.g., ETL, feature extraction), which can create an I/O bottleneck when handled separately from the training process.
- **Solution**: By integrating Spark into the pipeline, intermediate data is buffered in memory (as RDDs), reducing the need for repeated I/O access to storage like HDFS. This improved throughput by doubling the system's efficiency on average.

9.4.2 Training Platform Architecture

- **Architecture**: The training platform uses Spark to manage nodes, each hosting a Paddle trainer and Spark executor. Data is partitioned across nodes, enabling parallel processing. After each training iteration, parameter updates are collected and broadcasted back to nodes via the parameter server.
- **Parameter Server**: Alluxio, a memory-centric distributed storage, is used as the parameter server to avoid I/O bottlenecks. It significantly improves I/O performance by more than 5X compared to using HDFS.

9.4.3 Heterogeneous Computing

- **GPU vs CPU**: The system demonstrates a 15X speed-up in model training when using GPUs over CPUs for Convolution Neural Networks (CNNs).
- **Scalability**: As the number of GPUs increases, training latency decreases almost linearly, confirming the platform's scalability for handling larger datasets by adding more computing resources.

9.5 HD Map Generation

- **Purpose**: The infrastructure supports the multi-stage pipeline for HD map generation, which includes raw data reading, preprocessing, pose recovery, point cloud alignment, map labeling, and the final map output. By leveraging Spark and heterogeneous computing, the system reduces I/O and accelerates the critical stages of the process.

9.5.1 HD Map Layers

- **Grid Map**: The bottom layer of an HD map consists of a grid map generated from raw LiDAR data, capturing elevation and reflection in 5 cm by 5 cm grid cells. This allows autonomous vehicles to self-localize based on real-time LiDAR scans, GPS, and IMU data.
- **Semantic Layers**: Above the grid, there are layers with additional information such as lanes, traffic signs, and reference lines. These layers provide essential context for vehicle navigation, ensuring lane adherence and safety.

9.5.2 Map Generation in the Cloud

- **Sensor Fusion**: HD map generation uses data from various sensors—LiDAR, IMU, wheel odometry, and GPS—by fusing them to accurately derive position and location.
- **Process**:
    1. **SLAM**: Simultaneous Localization and Mapping (SLAM) aligns LiDAR scans with the vehicle's trajectory.
    2. **Map Generation**: LiDAR scans are stitched together to create a continuous map.
    3. **Labeling**: Semantic and label information is added to enhance the grid map with details like lanes and traffic signs.
- **Optimization with Spark**: By linking these stages into a single Spark job and using in-memory buffering for intermediate data, the system achieved a 5X speedup compared to processing each stage separately. The most computationally expensive step—Iterative Closest Point (ICP) for point cloud alignment—benefits from the heterogeneous infrastructure, further enhancing performance.

5.2

10.1 Background and Motivations

→ e-commerce logistic

- **Context**: The chapter presents a case study on autonomous last-mile delivery vehicles in complex traffic environments, focusing on e-commerce logistics.
- **Last-Mile Delivery**: This refers to the final stage of delivering goods from local distribution centers to consumers. It is crucial for a satisfactory e-commerce experience but is challenging due to the complexities of traffic environments.
- **Motivations**:
  1. **Rising Labor Costs**: Traditional delivery services face increasing labor costs, exemplified by the example of a delivery clerk earning $20,000 annually and delivering 110 parcels per day, making each delivery costly. This issue is expected to worsen as labor availability declines.
  2. **Inefficiencies in Human Labor**: Delivery clerks waste time on tasks like contacting consumers, waiting for pick-ups, and traveling, reducing productivity.

Why AV?
- labour cost
- inefficient

- **Benefits of Autonomous Vehicles**:
  1. **24/7 Operation**: Autonomous vehicles are not affected by weather or time and can deliver goods any time of day, including nights and holidays, without extra costs.
  2. **Temporal Flexibility**: Unlike human workers with fixed hours, autonomous vehicles can operate around the clock, increasing efficiency.
  3. **Cost Reduction**: Autonomous systems eliminate the costs of labor recruitment, training, and management.
  4. **Safety and Efficiency**: Autonomous vehicles enhance safety and operational efficiency, benefiting both the delivery process and public infrastructure.

Benefits -
- 24/7
- around
- cost ↓
- safety

- **Conclusion**: Deploying autonomous vehicles for last-mile delivery addresses the challenges faced by traditional methods, offering promising solutions to improve efficiency, reduce costs, and increase safety.

10.2 Autonomous Delivery Technologies in Complex Traffic Conditions

- **Challenges in Unruly Traffic Environments**: Autonomous vehicles face more difficulty in chaotic traffic conditions due to the large number of diverse traffic participants (pedestrians, bicycles, cars) who may not follow traffic rules. These challenges are particularly common in cities in China, where the urban environment is more complex than in developed countries like North America or Europe.
- **Key Factors Contributing to Complexity**:
  1. **High Population Density**: In China, most residents live in apartment complexes, leading to high population density in urban areas. Autonomous vehicles encounter many obstacles, such as pedestrians, cyclists, and automatic gates in parking lots, which create unstructured environments.
  2. **Diverse Traffic Participants**: Urban roads host a variety of vehicles, including bicycles, electric bikes, and motorcycles, each with unique movement characteristics. In addition, some residents drive in unsafe or illegal ways, such as carrying unusually long objects on bicycles.
  3. **Traffic Congestion**: In large cities, traffic jams are frequent due to the rapid increase in motor vehicle ownership, further complicating the traffic conditions.
- **Challenges for Autonomous Delivery Vehicles**: The complex, unruly traffic conditions in these environments are distinct from those in developed countries. Autonomous driving technologies suitable for more structured environments may not work well in chaotic urban settings. These systems need to be adapted to handle such scenarios effectively, considering the budget constraints and safety requirements.
- **Unique Safety Requirements**: Autonomous delivery vehicles must obey traffic laws and ensure road safety. Unlike passenger-carrying vehicles, delivery vehicles might need to prioritize not causing any danger to other vehicles or pedestrians. In some situations, delivery vehicles may need to sacrifice their own safety to prevent harm to others, placing unique safety requirements on the system.
- **Conclusion**: Autonomous delivery vehicles must be designed to handle the complexity of unruly traffic environments, with a focus on safety, adaptability, and efficiency in densely populated and chaotic urban areas.

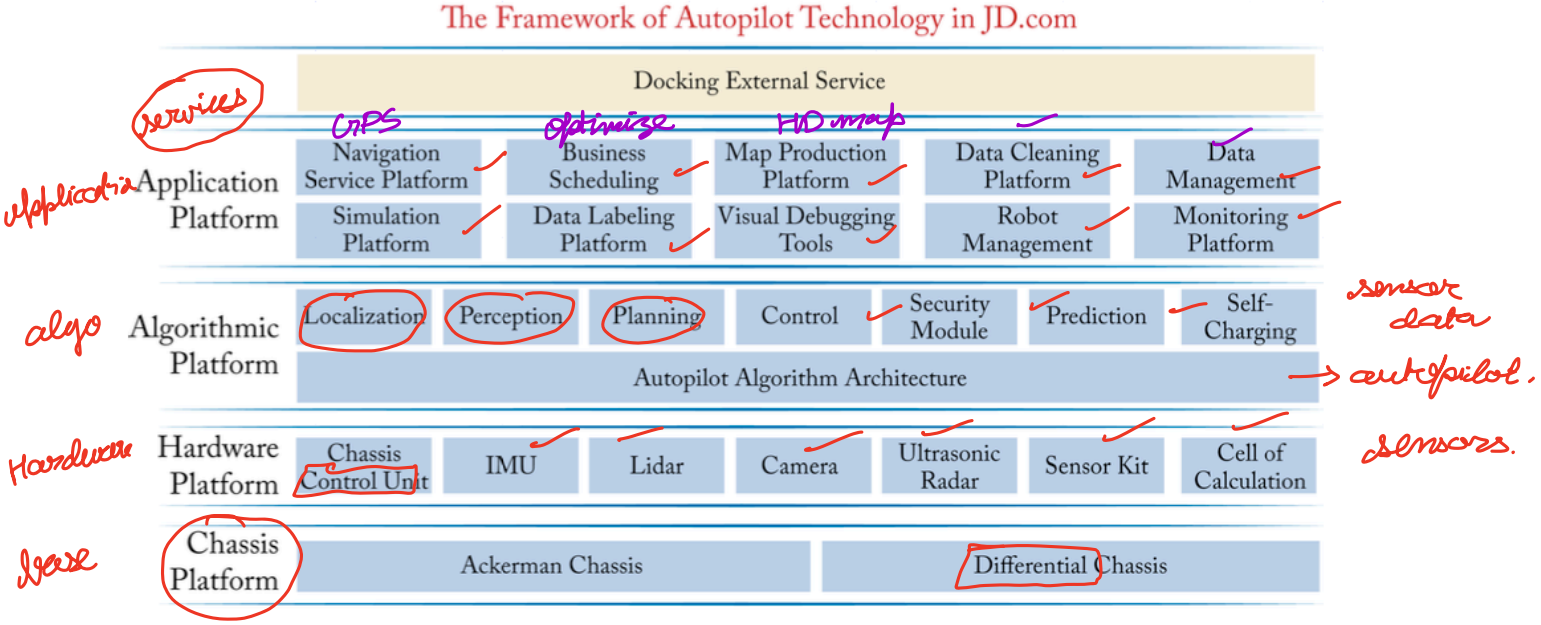


Figure 10.2: The architecture of our autonomous driving system.

10.3 JD.com: An Autonomous Driving Solution

JD.com, one of the largest e-commerce companies, is developing autonomous vehicles for last-mile delivery to reduce costs. The use of autonomous vehicles could cut delivery costs by 22.45%, and if 10% of all e-commerce orders were delivered autonomously, JD.com could save at least $110 million annually. Expanding this to 5% of the entire market could result in savings of up to $7.64 billion annually, driving further investment in autonomous vehicle R&D.

10.3.1 Autonomous Driving Architecture

JD.com's autonomous driving system consists of over 20 modules working together, divided into online and offline categories. **Online modules** function while the vehicle is on the road, covering tasks like localization, perception, prediction, decision-making, planning, and control. **Offline modules** are responsible for feature extraction, training, simulation, testing, and evaluation, such as map production and visual debugging tools. These modules support the autonomous vehicle's operation, ensuring safe, efficient, and smooth trips.

10.3.2 Localization and HD Map

Localization determines the vehicle's current position using five sources of information:

1. GPS signal (initial activation),
2. Multi-line LiDAR for matching point clouds to pre-stored HD maps,
3. Camera data,
4. Chassis-based odometry,
5. IMU data for precise calibration.

The system uses an Iterative Closest Point (ICP) method, which is calibrated by a Kalman filter. The HD map, which contains 16 layers of static and dynamic information, includes geometric, semantic, and real-time sub-maps. These maps provide details about road features such as lane width, type, intersections, traffic signals, and specific elements for delivery vehicles (e.g., gates, pillars, and safety islands).

For cost-effective map construction, JD.com uses sensors on the vehicles themselves and employs sensor fusion and SLAM technologies to overcome GPS limitations, particularly in urban areas. The company also incorporates machine learning to detect static vehicles and traffic lights, helping improve map construction efficiency. Frequent updates to the HD maps are required due to rapid changes in urban infrastructure.

10.3.3 Perception

The **perception module** is crucial for recognizing and tracking obstacles in the environment, enabling the autonomous system to navigate through complex traffic conditions. The vehicle is equipped with a range of sensors, including a 1-beam LiDAR, a 16-beam LiDAR, four mono cameras, a high-resolution HDR camera for traffic light detection, and ultrasonic receptors to prevent immediate collisions.

Object Detection

Three methods are used for object detection:

1. **Machine-learning-based method**: Processes point cloud data and is effective at classifying generic traffic-related objects, though less reliable with pedestrians.
2. **Geometry-based method**: Provides stable results for objects with typical shapes, such as pedestrians.
3. **Machine-learning-based visual method**: Focuses on color, shading, and texture, making it particularly effective for partially observed objects.

These methods complement each other to provide accurate and reliable object recognition.

Handling Unfamiliar Scenarios

When the vehicle encounters new or unusual objects, like a person riding an electric bicycle with a long stick (an illegal traffic scenario), it may initially struggle to handle the situation. In such cases, a remote operator can take over, and the vehicle will later incorporate this new object into its learning system. Data is manually marked for learning in the relevant detectors, using supervised learning techniques to improve recognition performance over time.

Additionally, a deduction approach is employed to estimate the status of surrounding traffic elements that are not directly observable, such as traffic light conditions or the status of vehicles ahead.

10.3.4 Prediction, Decision, and Planning

The **prediction, decision, and planning** module is essential for safely navigating autonomous vehicles through complex traffic environments.

Prediction

Prediction estimates the future trajectories of moving obstacles using a two-layer system:

1. **First layer**: Predicts the behavior of well-tracked vehicles using routing and lane information from the HD map, assuming vehicles follow traffic rules.
2. **Second layer**: Handles abnormal behaviors with machine learning and deduction methods.

Both layers work together to predict trajectories with associated confidence levels.

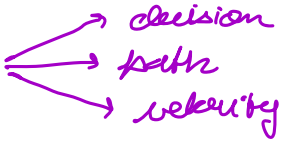
Decision and Planning

The **decision module** determines the vehicle's rough trajectory, while the **planning module** refines it to ensure safety and efficiency.

- **Decision module**: Uses a sample-and-search method with dynamic programming, considering uncertainties from perception and prediction modules. The goal is to determine a safe, robust trajectory.
- **Planning module**: Further refines the decision through path and velocity optimization.

Trajectory Planning

The planning is split into phases:



1. **Decision generation**: Determines interactions between the vehicle and surrounding objects.
2. **Path planning**: Optimizes the vehicle's path within a constrained space, avoiding obstacles.
3. **Velocity planning**: Optimizes speed while ensuring smoothness and safety.

Mathematical optimization is used to ensure smooth transitions, avoid collisions, and adhere to safety constraints. The Quadratic Programming (QP) problem is solved to minimize deviations from the desired trajectory while respecting the vehicle's physical constraints. The system solves the QP problem quickly, ensuring real-time responsiveness to sudden changes in the environment.

10.4 Safety and Security Strategies

Safety is a top priority in the design of autonomous delivery vehicles. The safety measures are implemented across three main levels: simulation, vehicle-end, and remote monitoring.

10.4.1 Simulation-Level Verification

All code is tested through extensive simulation, where raw data is used to create realistic virtual scenarios. The vehicle's performance is evaluated against defined criteria. If the code passes simulation, it is then tested in real vehicles. Any issues discovered are recorded and used to improve future code testing, ensuring rapid safety improvements.

10.4.2 Vehicle-End Monitoring

A **vehicle-end guardian module** monitors the vehicle's control system and external environment for potential emergencies. It detects failures, both internal (hardware) and external (obstacles), and employs redundant systems for fail-safes. If the redundant systems fail, the guardian module follows pre-defined recovery protocols to maintain safety.

10.4.3 Remote Monitoring

A **remote monitoring platform** allows engineers to oversee the vehicle's behavior in real-time. If necessary, engineers can take control to help the vehicle in abnormal situations. If no engineer is available, the system alerts authorities, such as the police, to handle the emergency.

These multi-layered safety strategies ensure that autonomous vehicles are continuously monitored and can respond to potential issues effectively.

10.5 Production Deployments

The deployment of autonomous vehicles progresses through four stages:

1. **Stage 1:** Autonomous driving at low speeds with manual surveillance.
2. **Stage 2:** Autonomous driving at low speeds without manual surveillance.
3. **Stage 3:** Autonomous driving at relatively high speeds with manual surveillance.
4. **Stage 4:** Autonomous driving at relatively high speeds without manual surveillance.

Progress between stages is gradual and adaptive based on technical success, with speed increases corresponding to technical improvements. Alongside the technical roadmap, the path to profitability progresses as well. Initially, low-level autonomous technology is developed for indoor applications, such as warehousing logistics, which can later be applied to autonomous delivery, ensuring a smoother transition to full deployment. The company has also worked to optimize scheduling and time efficiency across e-commerce platforms, warehouses, and distribution centers. Over 300 self-driving vehicles have been deployed for trial operations, accumulating 715,819 miles.

10.6 Lessons Learned

Key lessons learned during the deployment of autonomous vehicles include:

1. **Explainability of Algorithms:** Algorithms need to be explainable to ensure their performance is predictable and acceptable to other road users. Machine learning-based methods are favored over deep-learning end-to-end solutions.
2. **Dependence on HD Maps:** Since delivery routes are mostly fixed, HD maps are critical for recording fine details of these routes.
3. **Focus on Risk Recognition:** Aiming for a higher take-over mile index can be misleading. Recognizing risks and calling for manual take-over when necessary is a vital part of the safety system.
4. **Human-Machine Collaboration:** It is essential to differentiate between tasks suitable for humans and those for machines. Humans should be responsible for overseeing complicated, repetitive scenarios, while machines handle routine tasks. Autonomous vehicles do not eliminate the need for human involvement but enable humans to focus on more innovative tasks related to the system's maintenance.