CONTENTS

CHAPTER-1: INTRODUCTION

Driving a car requires precision and active attention. Any distraction to the driver could lead to accidents and losses. According to the data provided by Singapore Police Force, there were 94.34% of the road accidents led by human errors in year 2016 which could be avoided, for instance, failing to keep proper lookout, failing to have proper control and change lane without due care. Other road users are affected by the road accidents even though they follow the rules and regulations.

Lately, drivers divert their attention to their smartphones to answer calls, send messages or navigate to their destinations. It increases the chances of road accident when drivers are not focusing on the road condition.

The automotive industry is innovating on the smart features of car to further improve the safety and driving experience for users. The automatic braking system or collision avoidance system is one of the innovation that uses sensors such as radar to detect and mitigate collision. The innovation in preventive safety mechanism catalyses the development process of autonomous vehicle that can drive to the destination autonomously and then reduce the number of road accidents that were caused by human errors by prioritizing the safety of passengers and other road users including motorcyclists and pedestrians.

Self-driving car is the future development in automotive industry. The breakthrough in performance of parallel processing hardware and innovation in neural network design allows an artificial intelligence(AI) agent to control a vehicle autonomously. An AI agent is more capable of sustained attention and focus than humans. However, building an autonomous vehicle is a long-standing goal and complex task. Driving requires three main capabilities including recognition, prediction and planning[1].

First of all, an AI agent requires to recognize the environment with sensors. Convolutional neural networks (CNN) are the most successful neural network model that is scalable and able to extract important spatial features from spatial data. Moreover, an AI agent has to predict the future states of the environment and perform best action at the moment.

Furthermore, the planning component integrate the ability of model to understand the environment (recognition) and its dynamics (prediction) to plan the future actions to avoid unwanted situations (penalties) and drives safely to its destination (rewards). The video feed from camera and output of Light Detection and Ranging(LIDAR) are mainly used as the visual feed to AI agents.

1.1 Background

An autonomous vehicle requires to carry out multiple tasks to drive on the road with dynamic traffic condition. The tasks include following the road marking, keeping a safe distance with vehicles, reacting to emergency situation accordingly and navigating to the destination. Udacity has partnered with Didi Chuxing to organize the first self-driving car challenge in year 2017 to come up with the best way to detect obstacles using camera and LIDAR data. It could ensure self-driving cars prioritize the safety of road users and pedestrians. Besides that, Udacity has organized another challenge to predict the steering angle with deep learning based on the image inputs from a camera mounted to the windshield.

However, the quality of the driving skill and its efficiency in getting passengers to the destination are often neglected while most of the researches and designs focus on the basic features of self-driving car. The driving skill of driving agent becomes the priority to deliver the best transportation service compared to a manned vehicle. For instance, a self-driving caragent should be capable to switch to the optimal lane which maximize its speed rather than follow slow-moving vehicles in a predefined lane all the time. It is important because passengers expect to reach its destination by taking the shortest journey and shortest time along with safety.

1.2 Objective

The objective of this project is to design and implement a self-driving car-agent to maximize the its speed. A simulation environment software is required to simulate and visualize the actual traffic environment. A neural network model is essential to build a self-driving car-agent that interact with complex traffic condition. Reinforcement learning technique is used to train the model to adapt to the dynamic environment with infinite action-state sequences. The agent is required to learn the optimal policy to choose the optimal available option which maximizes the speed of vehicle.

1.3 Scope

The scopes of this project are to maximize the speed of a self-driving car on an expressway with seven lanes in a simulated environment. The simulated environment is built with pygame framework in Python environment. The autonomous vehicle is assumed to function perfectly to steer along the road. It simulates the road conditions where a self-driving car-agent on a 3 seven-lane roads with the other non-agent-controlled cars with several human driving behavior’s.

1.4 Existing Systems

Autonomous cars rely on a combination of hardware and software to navigate and operate without human intervention. Several existing systems power these vehicles, each incorporating advanced technologies such as AI, computer vision, and sensor fusion.

1. Key Systems in Autonomous Cars

(a) Perception System

- Uses sensors (LiDAR, cameras, radar, ultrasonic) to detect surroundings.
- Identifies obstacles, pedestrians, traffic signals, and road conditions.
- Example: Tesla’s Autopilot uses a vision-based system, while Waymo relies on LiDAR.

(b) Localization & Mapping

- Uses GPS, IMU (Inertial Measurement Unit), and HD Maps for positioning.
- SLAM (Simultaneous Localization and Mapping) helps in real-time environment mapping.
- Example: Waymo’s HD mapping provides centimeter-level accuracy.

(c) Planning & Decision Making

- Uses AI algorithms and deep learning to make driving decisions.
- Considers factors like road rules, traffic, and pedestrian movement.
- Example: NVIDIA's DRIVE platform uses deep neural networks for planning.

(d) Control System

- Converts decisions into actions like steering, acceleration, and braking.
- Uses drive-by-wire technology for electronic control of vehicle functions.

2. Levels of Autonomous Systems (SAE Levels 0-5)

- Level 0: No automation (manual driving).
- Level 1: Driver assistance (adaptive cruise control, lane-keeping).
- Level 2: Partial automation (Tesla Autopilot, GM Super Cruise).
- Level 3: Conditional automation (Mercedes-Benz Drive Pilot).
- Level 4: High automation (Waymo, Cruise – in geofenced areas).
- Level 5: Full automation (No human intervention – still in development).

3. Existing Autonomous Car Companies & Technologies

- Tesla Autopilot & FSD (Full Self-Driving): Vision-based AI.
- Waymo (Google): LiDAR-heavy approach with HD mapping.
- Cruise (GM): Robotaxis operating in select cities.
- Mobileye (Intel): Advanced driver-assistance systems (ADAS).
- NVIDIA DRIVE: AI-powered autonomous vehicle platform.

1.5 Proposed Systems

A proposed system for an autonomous car aims to improve upon existing technologies by increasing safety, accuracy, and efficiency. It integrates advanced AI, sensor fusion, V2X (Vehicle-to-Everything) communication, and improved decision-making algorithms to achieve full automation (SAE Level 5).

1. Key Enhancements in the Proposed System

(a) Advanced Sensor Fusion

- ➢ Combines LiDAR, RADAR, Cameras, and Ultrasonic Sensors with AI for better object detection.
- ➢ Uses thermal imaging to enhance night and fog vision.

(b) High-Precision Localization & Mapping

- ➢ Uses 5G-based GPS & Real-Time Kinematic (RTK) positioning for centimeter-level accuracy.
- ➢ Integrates SLAM (Simultaneous Localization and Mapping) for real-time adjustments.
- ➢ Dynamic HD maps with AI-driven updates to account for road changes.

(c) AI-Driven Decision-Making

- ➢ Implements deep reinforcement learning (DRL) to adapt driving behavior dynamically.
- ➢ Uses predictive AI models to anticipate pedestrian and vehicle movements.
- ➢ Applies edge computing for faster real-time decision-making without cloud dependency.

(d) V2X Communication (Vehicle-to-Everything)

- ➢ Vehicle-to-Vehicle (V2V): Communicates with nearby vehicles for coordinated movement.
- ➢ Vehicle-to-Infrastructure (V2I): Interacts with traffic lights, road signs, and smart city systems.
- ➢ Vehicle-to-Pedestrian (V2P): Detects pedestrians and cyclists for improved safety.

(e) Enhanced Cybersecurity & Privacy

- ➢ Uses blockchain for data integrity and secure communication between vehicles.
- ➢ Implements AI-driven anomaly detection to prevent hacking attempts.

(f) Sustainable Energy & Efficiency

- ➢ Optimizes battery usage in electric vehicles through AI-powered energy management.
- ➢ Uses solar-assisted power systems to increase energy efficiency.

2. Proposed Architecture of the System

- ➢ Perception Layer: Sensors collect data (LiDAR, Radar, Cameras, GPS).
- ➢ Processing Layer: AI algorithms analyze sensor data for object detection, and mapping.

- Decision-Making Layer: Reinforcement learning AI decides acceleration, braking, and lane changes.
- Communication Layer: V2X communication exchanges data with vehicles and infrastructure.
- Control Layer: Drive-by-wire system executes vehicle control commands.

3. Expected Benefits

- Higher Safety: AI-based predictive analysis reduces accidents.
- Better Efficiency: 5G, edge computing, and V2X improve traffic flow.
- Scalability: Works in complex urban environments.
- Sustainability: Optimized for electric vehicles with minimal energy waste.

CHAPTER-2: LITERATURE SURVEY

1. A Comprehensive Survey on Autonomous Vehicle Technologies

Summary: This survey explores key technologies enabling self-driving cars, including machine learning, sensor fusion, computer vision, and control systems. It discusses different levels of automation, challenges in real-world deployment, and future trends.
Reference: Thrun, S. (2010). "Toward Robotic Cars." Communications of the ACM, 53(4), 99–106.

2. Sensor Fusion and Perception in Autonomous Vehicles

Summary: This study reviews different sensors used in autonomous vehicles, such as LiDAR, radar, cameras, and IMUs. It also examines sensor fusion techniques and their role in perception, object detection, and scene understanding.
Reference: Alam, K., Sarker, M. R., & Kowadlo, G. (2014). "A Review of Sensor Fusion Techniques in Autonomous Vehicles." Journal of Robotics and Automation, 2(3), 45–57.

3. Deep Learning Applications in Autonomous Driving

Summary: This paper reviews the role of deep learning in perception, decision-making, and control. It highlights CNNs for object detection, RNNs for trajectory prediction, and reinforcement learning for behavior planning.
Reference: Bojarski, M., Testa, D., & Dworakowski, D. (2016). "End to End Learning for Self-Driving Cars." arXiv preprint arXiv:1604.07316.

4. Challenges and Ethical Considerations in Autonomous Driving

Summary: The study discusses ethical dilemmas (e.g., decision-making in crash scenarios), legal issues, cybersecurity threats, and public trust in self-driving cars.
Reference: Bonnefon, J. F., Shariff, A., & Rahwan, I. (2016). "The Social Dilemma of Autonomous Vehicles." Science, 352(6293), 1573–1576.

5. Safety and Reliability in Autonomous Vehicles

Summary: This survey examines the reliability of autonomous driving systems, covering software validation, real-world testing, and failure analysis. It also reviews accident case studies involving self-driving cars.

Reference: Koopman, P., & Wagner, M. (2017). "Autonomous Vehicle Safety: An Interdisciplinary Challenge." IEEE Intelligent Transportation Systems Magazine, 9(1), 90–96.

6. V2X Communication and Its Role in Autonomous Vehicles

Summary: This literature survey focuses on vehicle-to-everything (V2X) communication, including V2V (vehicle-to-vehicle) and V2I (vehicle-to-infrastructure). It discusses protocols, latency challenges, and security concerns in connected autonomous driving.
Reference: Abbas, R., Clancy, T., & Gopalakrishnan, K. (2018). "A Survey on V2X Communication for Autonomous Vehicles." IEEE Access, 6, 35089–35112.

7. Motion Planning and Control for Autonomous Vehicles

Summary: This paper reviews different motion planning algorithms like A*, RRT (Rapidly-exploring Random Trees), and optimization-based approaches. It also discusses vehicle dynamics, trajectory generation, and real-time control techniques.
Reference: Paden, B., Čáp, M., Yong, S. Z., Yershov, D., & Frazzoli, E. (2016). "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles." IEEE Transactions on Intelligent Vehicles, 1(1), 33–55.

8. Human-Machine Interaction in Autonomous Vehicles

Summary: This study explores how autonomous vehicles interact with human drivers, pedestrians, and passengers. It includes topics such as user trust, explainability of AI decisions, and transition of control in semi-autonomous systems.
Reference: Merat, N., Madigan, R., & Nordhoff, S. (2018). "Human Factors, User Requirements, and User Acceptance of Ride-Sharing in Automated Vehicles." Transportation Research Part A: Policy and Practice, 122, 116–127.

9. Edge Computing for Autonomous Vehicles

Summary: This paper examines how edge computing enhances real-time decision-making in autonomous vehicles. It discusses reducing latency in perception and control through decentralized processing near the vehicle.
Reference: Hou, X., Li, Y., Chen, M., Wu, D., Jin, D., & Chen, S. (2016). "Vehicular Fog Computing: A Viewpoint of Vehicles as the Infrastructures." IEEE Transactions on Vehicular Technology, 65(6), 3860–3873.

10. Cybersecurity Threats and Countermeasures in Autonomous Vehicles

Summary: This survey explores various cybersecurity challenges in autonomous vehicles, including hacking, spoofing attacks on sensors, and software vulnerabilities. It also reviews security frameworks and encryption techniques.
Reference: Petit, J., & Shladover, S. E. (2015). "Potential Cyberattacks on Automated Vehicles." IEEE Transactions on Intelligent Transportation Systems, 16(2), 546–556.

11. Energy Efficiency and Sustainability in Autonomous Driving

Summary: This paper reviews how autonomous vehicles impact fuel consumption, electric vehicle integration, and smart traffic management. It discusses how AI-driven driving can optimize energy efficiency and reduce carbon emissions.
Reference: Wadud, Z., MacKenzie, D., & Leiby, P. (2016). "Help or Hindrance? The Travel, Energy, and Carbon Impacts of Highly Automated Vehicles." Transportation Research Part A: Policy and Practice, 86, 1–18. 12. Machine Learning and AI for Autonomous Driving

12. Machine Learning and AI for Autonomous Driving

Summary: This survey explores the use of artificial intelligence in self-driving cars, focusing on supervised and reinforcement learning techniques for perception, decision-making, and control. It also reviews dataset availability and AI model performance evaluation.
Reference: Grigorescu, S., Trasnea, B., Cocias, T., & Macesanu, G. (2020). "A Survey of Deep Learning Techniques for Autonomous Driving." Journal of Field Robotics, 37(3), 362–386.

13. Autonomous Vehicles in Smart Cities

Summary: This paper reviews the integration of autonomous vehicles into smart city infrastructure, covering traffic management, urban planning, and environmental impact. It discusses how AVs interact with IoT-based smart transportation systems.
Reference: Talebpour, A., & Mahmassani, H. S. (2016). "Influence of Connected and Autonomous Vehicles on Traffic Flow Stability and Throughput." Transportation Research Part C: Emerging Technologies, 71, 143–163.

CHAPTER-3: SYSTEM REQUIREMENT SPECIFICATIONS

3.1 FUNCTIONAL REQUIREMENTS

Functional requirements for a self-driving car define the essential tasks and behaviors the system must perform to ensure safe and efficient autonomous operation. These include:

1. Perception & Sensing

- Detect and classify objects (vehicles, pedestrians, cyclists, road signs, lane markings, traffic signals).
- Use sensors (LiDAR, radar, cameras, ultrasonic sensors) to gather real-time environmental data.
- Detect road conditions (potholes, weather impact like rain or snow).

2. Localization & Mapping

- Continuously determine vehicle position using GPS, IMU, and sensor fusion.
- Maintain and update high-definition (HD) maps for navigation.
- Identify road lanes, intersections, and landmarks for better path planning.

3. Path Planning & Decision Making

- Plan a safe and optimal route from the source to the destination.
- Make real-time driving decisions (lane changes, overtaking, merging, turning, stopping at signals).
- Adapt to dynamic road conditions (accidents, detours, traffic congestion).

4. Control & Motion Execution

- Execute acceleration, braking, and steering commands smoothly.
- Maintain lane discipline and follow speed limits.
- Ensure smooth handling in emergency scenarios (sudden braking, obstacle avoidance).

5. Communication & Human Interaction

- Communicate with other vehicles (V2V) and infrastructure (V2I) for traffic updates.
- Notify passengers of critical decisions (e.g., emergency braking).
- Allow manual override and ensure seamless transition between autonomous and manual modes.

6. Safety & Compliance

- Comply with traffic rules and regulations.
- Detect and respond to emergencies (ambulances, law enforcement, roadblocks).
- Implement fail-safe mechanisms (safe stopping in case of system failure).

7. Cybersecurity & Data Management

- Ensure data encryption and secure vehicle-to-cloud communication.
- Protect against hacking and unauthorized access.
- Log and analyze driving data for continuous improvement.

3.2 NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements (NFRs) for a self-driving car define the quality attributes, constraints, and overall system performance expectations. These ensure reliability, safety, and user experience beyond just functional capabilities.

1. Performance

- Real-time Processing: The system must process sensor data and make driving decisions within milliseconds.
- Low Latency: Decision-making (braking, steering, obstacle avoidance) should happen with minimal delay.
- High Accuracy: Object detection, localization, and path planning must achieve near 100% accuracy.

2. Safety & Reliability

- Fail-Safe Mechanisms: The system should transition to a safe state in case of sensor failure, software errors, or cyber-attacks.
- Redundancy: Critical components (braking, steering, computing) must have backups.
- System Uptime: The autonomous system should operate with at least 99.99% availability.

3. Security

- Data Encryption: Secure all communications (V2V, V2I, cloud) to prevent hacking.
- Access Control: Restrict unauthorized modifications or control of vehicle functions.
- Anomaly Detection: Identify and respond to cyber threats in real-time.

4. Scalability & Maintainability

- Over-the-Air (OTA) Updates: Support remote software updates without disrupting functionality.
- Modular Architecture: Components should be upgradable or replaceable without affecting the whole system.
- Self-Diagnosis: Ability to detect and report hardware or software issues proactively.

5. User Experience (UX) & Human Interaction

- Smooth Transitions: Ensure seamless handoff between autonomous and manual driving.
- Clear Communication: Provide intuitive alerts and updates to passengers (e.g., route changes, emergency stops).
- Comfortable Ride: Optimize acceleration, braking, and turns for a smooth experience.

6. Compliance & Regulatory Standards

- Legal Compliance: Must adhere to regional traffic laws and safety regulations (e.g., ISO 26262 for automotive safety).
- Ethical Decision-Making: Ensure ethical handling of unavoidable crash situations.
- Testing & Certification: Pass rigorous validation under various weather, road, and traffic conditions.

CHAPTER-4: ALGORITHMS

4.1 REINFORCEMENT LEARNING

Reinforcement learning is one of the learning approach for neural networks. Reinforcement learning implements differently compared to supervised learning and unsupervised learning. Reinforcement learning makes an agent to generate the optimal policy in an environment and maximizes the reward [2]. An agent can learn the rewards given by taking action in respective state and subsequently learn the proper action for each state without having any predefined rules and knowledge about the environment [3]. The agent learns by trials and errors, or in another words, the actions with higher reward are reinforcement while actions with penalties are avoided in the future. It is different with supervised learning because supervised learning requires huge number of labelled dataset to train the model [4]. The training process is repeated with same set of data until the model converges. It requires efforts in labelling the data and the process is error-prone[5]. Reinforcement learning is also different with unsupervised learning because reinforcement learning aims to maximize the reward while unsupervised learning does not.

4.2 CONVOLUTION NUERAL NETWORK

The efficiency and learning ability of brain have inspired the innovation of the convolutional neural networks (CNN). The ability of animal to perceive features from complex visual map and generate responses has biological guided the construction of CNN [11]. The local correlation can be extracted spatially by allowing a local connectivity pattern between neighboring cells of adjacent layers. Figure 1 shows that each filter is applied across the entire input map and generate a feature map. The output of the feature map is connected to the adjacent layer with features extracted. Each feature map explores unique features regardless of the location across visual field.

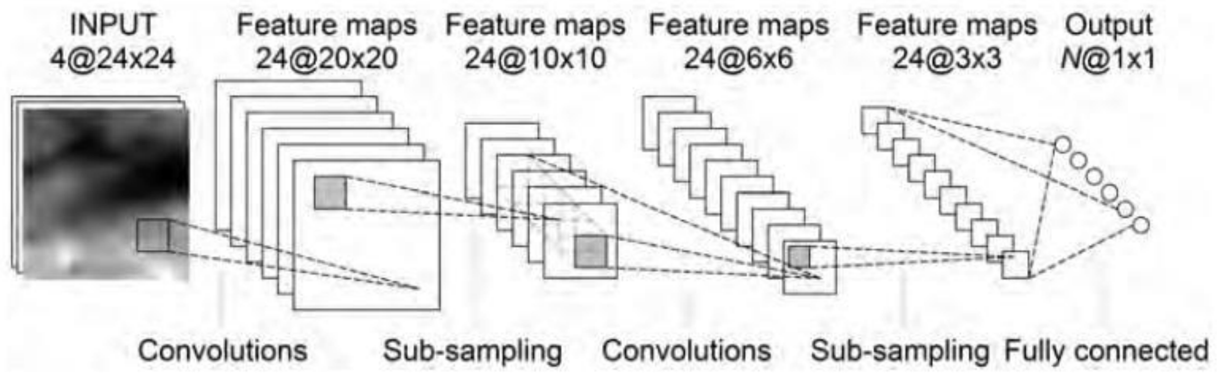


Figure 1: The illustration of convolutional layers and fully-connected layers forming convolutional neural network

4.3 DEEP Q-LEARNING NETWORKS(DQN)

Neural network is utilized as the function approximator of the Q-learning algorithm to form the Deep Q-learning network (DQN). The objective of DQN is to minimize the Mean Square Error (MSE) of the Q-values. Besides that, experience replay technique uses first-in-first-out(FIFO) method to keep experience tuples in replay memory for every time step. The replay memory stores experience tuples for several episodes to ensure that the memory holds diversified experiences for different states. The experience tuples are sampled from replay memory randomly during Q-learning updates. The implementation of experience replay can remove correlations in the observation sequence and smoothing over changes in the data distribution by randomizes over the data [13]. In practice, the most recent N experience records are sampled uniformly and randomly from replay memory. The random sampling gives equal probability to all experience tuples.

4.4 DOUBLE DEEP Q-LEARNING(DDQN)

In DQN, a batch of experience tuples are sampled in each learning iteration. The Q-learning network is updated based on the difference of the estimated future rewards. The sampling process is random and the frequent update of the network destabilizes the learning process. The network is facing difficulties to reach convergence. Thus, a separate target network, Q^ is used to improve the stability of the main neural network [14]. The target network Q^ is used to estimate the discounted future rewards given the subsequent state Q^,12 and action Q^,12 at unit time step Q^. The target network Q^ is synchronized with the main network Q^ for every C Qlearning updates on the main network.

4.5 MARKOV DECISION PROCESS

Markov Decision Process (MDP) can make a sequence of decisions based on the utilities of environment state. In which a set of possible environment states(S), a set of possible actions(A), discount factor(�), reward function (R), and state transition probability(ℙ). Environment states S are sequence of states with initial state s0. Actions A are available actions for environment states S. Reward function R is a map of immediate reward given (state, action) pair. State transition probability ℙ is the probability of transition from states S1. … Sn. However, there are infinite state sequences in the environment and calculation of transition probability for all state sequences is not feasible.

4.6 Q-LEARNING

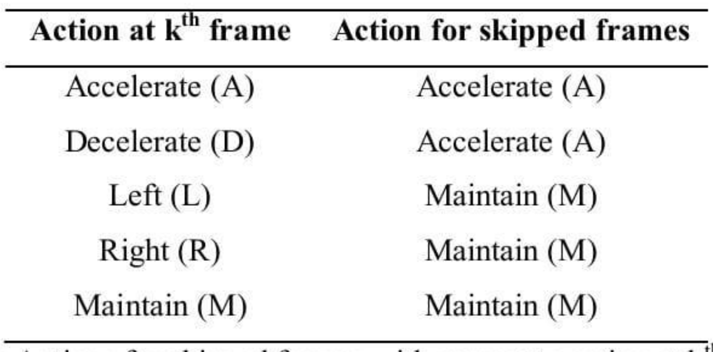
Q-learning is a reinforcement learning mechanism that compare the expected utility of available actions given a state. Q-learning can train a model to find the optimal policy in any finite MDP.

4.7 ALGORITHM

Reinforcement learning is the learning approach used in this project. Algorithm from [8] is used to interact with the simulator and use the surrounding environment states as the input of the algorithm. The S number of the most recent states St, at time step t are supplied to as input. The available action options are Accelerate (A), Decelerate (D), Left (L), Right (R) and Maintain (M). Epsilon greedy exploration is implemented to maximize the exploration and ensure agent experiences more possible states in the state domain. An action at, is randomly chosen from the action options as the agent's decision when the probability is lower than the current epsilon-greedy probability. The epsilon-greedy probability is uniformly decreases with number of processed states. Otherwise, an action with highest Q-value among the other options is chosen. The chosen action is then relayed to the simulator and the resulting state St+1,12 and reward rt, is recorded. Reward rt, is the difference of score at time step t and t− 1. The experience tuple (st,at,rt,st+1) is appended to the replay memory D with capacity N.

The learning algorithm starts at F frames so that the learning process does not over fit the limited experiences in the replay memory D. Experience replay is deployed to apply Q-learning update gradually. Learning process samples the most recent N experience rows in the replay memory. The goal of the reinforcement learning agent is to perceive the environment in the simulator and the optimal action that maximizes the future reward is chosen. The optimal Qvalue function Q*(s,a) can be modelled by a deep learning model as the nonlinear function approximator of the Q-learning algorithm. Neural networks are used to approximate the optimal Q-learning network. Thus, the parameterized neural network is used to approximate the optimal Q-learning network.

Frame-skipping technique is adopted during training. The agent reacts with the surrounding environment every kth frame to increase the response time of agent's reaction. In this project, the actions for skipped frames are determined in Table 3. Action at kth frame Action for skipped frames are determined in the following table.



Action at k^{th} frame	Action for skipped frames
Accelerate (A)	Accelerate (A)
Decelerate (D)	Accelerate (A)
Left (L)	Maintain (M)
Right (R)	Maintain (M)
Maintain (M)	Maintain (M)

Table 3: Actions for skipped frames with respect to action at k^{th} frame.

CHAPTER-5: SYSTEM DESIGN

5.1 MODEL ARCHITECTURE

The baseline model follows the network design listed. As shown in Figure 4, the baseline network takes the environment mapping of four most recent states as input, which is a set of 4×36×3 state. The state consists of Boolean values, in which true (1) if there is obstacle in the vision field and false (0) if there is not. The first hidden layer convolves 16 filters of 3×3 and stride 2. Rectified Linear Unit (ReLU) is applied to the outputs of the first hidden layer. The outputs of first hidden layer are flatten and passed to the fully-connected layer. The fully connected linear layer comes with one output for five actions.

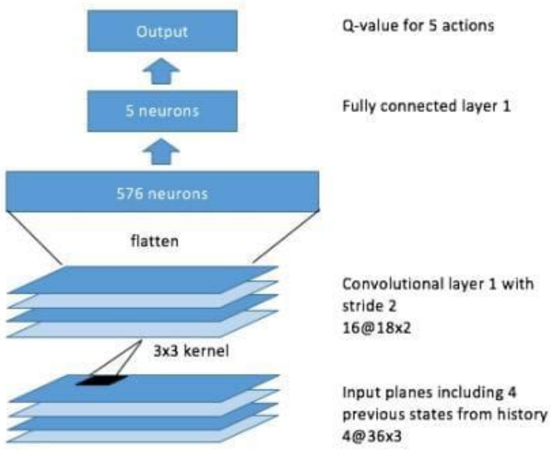

Output
Q-value for 5 actions
5 neurons
Fully connected layer 1
576 neurons
flatten
Convolutional layer 1 with stride 2 16@18x2
3x3 kernel
Input planes including 4 previous states from history 4@36x3

Figure 4: Baseline model with CNN architecture with one convolutional layer and one fully-connected layer.

The proposed model is implemented as shown in Figure 5. The network is enhanced with additional convolutional layer and fully-connected layer. The second convolutional layer compresses the output from the first convolutional layer to extract important features from the inputs. It passes the important features to fully-connected layer. The two fully-connected layers of the model brings the ability to approximate nonlinear function of action-value function. The additional sub-network is added to accept action stack as input. Action stack is a stack of action memory which stores the four most recent actions taken. The rational is to improve the stability of the agent from switching lane at high frequency.

- Perception (Sensors & Data Fusion): Detects surroundings.
- Localization & Mapping: Determines vehicle position.
- Perception & Object Detection: Identifies objects and road conditions.
- Decision-Making & Path Planning: Plans optimal routes and actions.
- Control & Actuation: Executes vehicle movements.
- Communication & V2X: Interacts with external systems.
- Cybersecurity & Safety: Ensures secure and fail-safe operations.

This architecture enables a self-driving car to navigate autonomously while ensuring safety, efficiency, and compliance with traffic regulations.

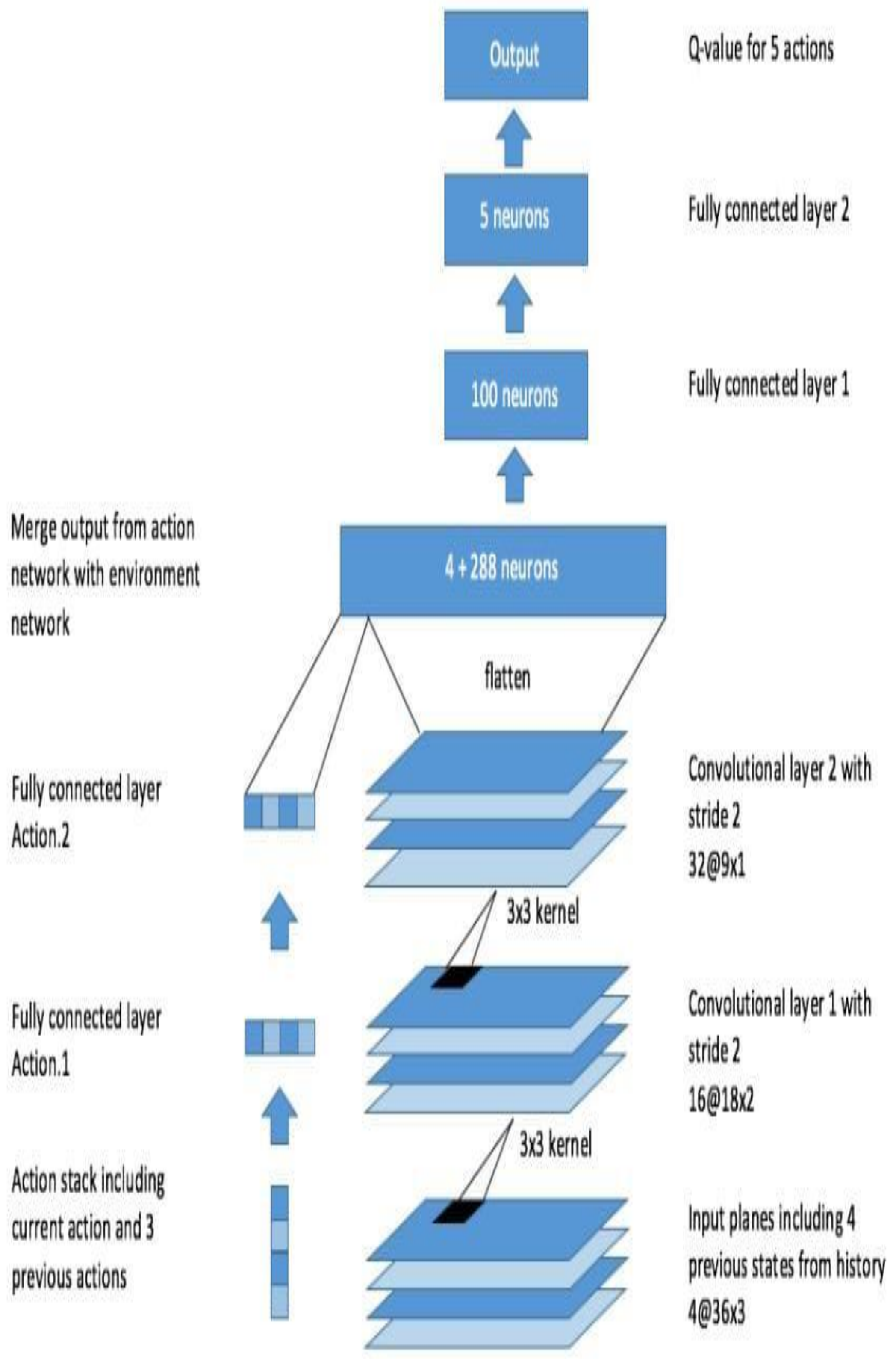

Output
Q-value for 5 actions
5 neurons
Fully connected layer 2
100 neurons
Fully connected layer 1
Merge output from action network with environment network
4 + 288 neurons
flatten
Fully connected layer Action.2
Convolutional layer 2 with stride 2 32@9x1
3x3 kernel
Fully connected layer Action.1
Convolutional layer 1 with stride 2 16@18x2
3x3 kernel
Action stack including current action and 3 previous actions
Input planes including 4 previous states from history 4@36x3

Figure 5: Proposed model with CNN architecture with 2-layer fully-connected for action stack and two convolutional layer and two fully-connected layer.

5.2 DATA FLOW DIAGRAM

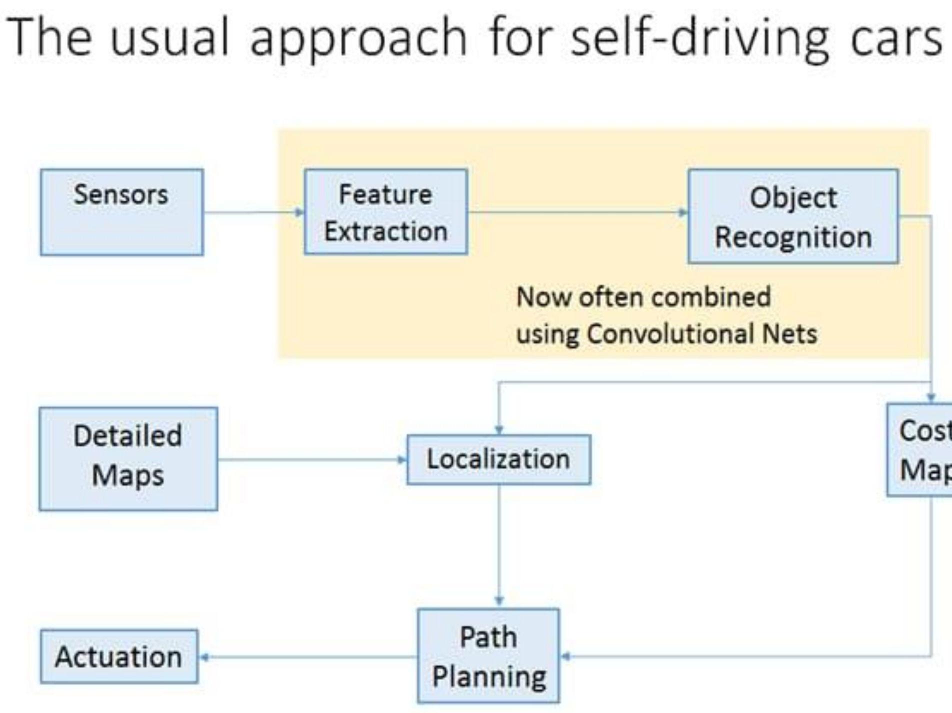


Fig: Data Flow Diagram.

The car receives data from its sensors. The ADS processes the data to understand the surroundings. The system makes a decision (e.g., accelerate, brake, turn).Sends commands to actuators (motor, brakes, steering). A DFD for a self-driving car illustrates how sensor data is collected, processed, and used for decision-making. It ensures that the car can navigate safely by understanding its environment and taking appropriate actions.

5.3 SEQUENCE DIAGRAM

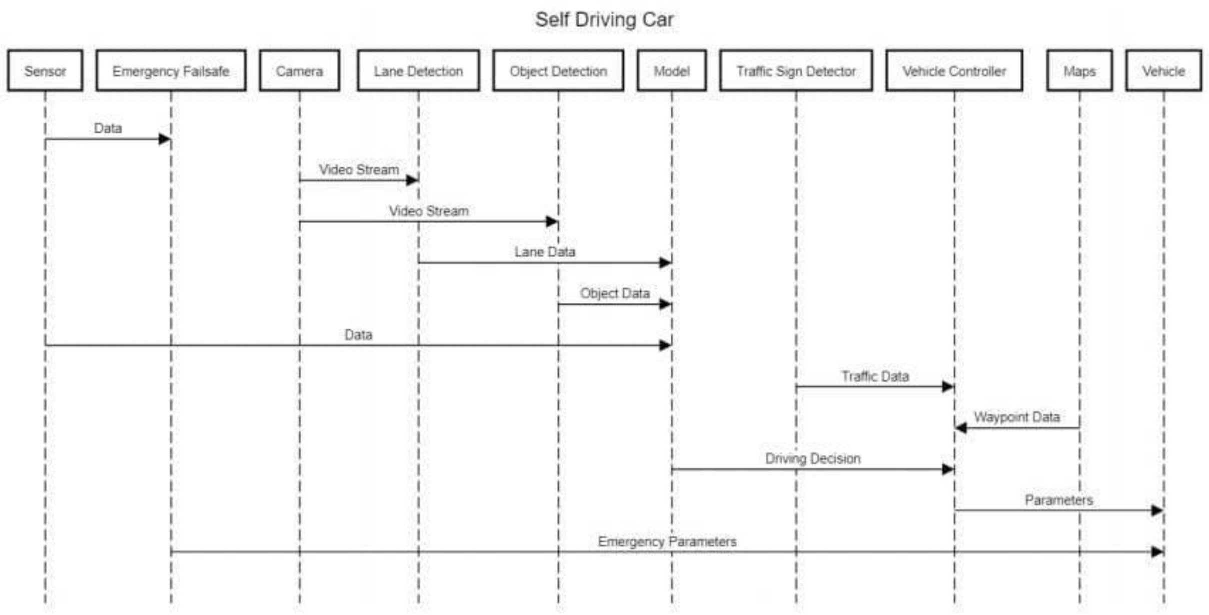


Fig: Sequence Diagram.

A sequence diagram in UML (Unified Modeling Language) represents the interaction between different components of a self-driving car in a time-ordered sequence. It shows how data flows between objects and systems to achieve autonomous driving.

Sequence Flow of Events

1. Initial Sensing & Data Collection

- Environment Sensors: Sensors continuously collect real-time data. ←
- Sensors → Perception Module: Raw sensor data is sent to be processed.

2. Perception Processing

- Perception Module → Object Detection System: Identifies pedestrians, vehicles, lanes.
- Perception Module → Localization Module: Helps determine car position relative to the environment.

3. Decision Making

- Localization Module → Planning Module:
- Provides precise location for path planning.
- Perception Module → Planning Module:
- Sends processed data about obstacles, road conditions.
- Planning Module ← Control System:
- Decides actions (accelerate, brake, turn).

4. Actuation & Execution

- Control System → Actuators: Executes
- planned actions via motor, brakes, and steering.
- Actuators ← Environment: Car moves, affecting its surroundings.

A sequence diagram for a self-driving car helps visualize how different components interact in real-time. It ensures efficient coordination between sensing, processing, decision-making, and execution, allowing the car to drive autonomously. It represents the interaction between different components of a self-driving car in a time-ordered sequence. It shows how data flows between objects and systems to achieve autonomous driving. It ensures efficient coordination between sensing, processing, decision-making, and execution, allowing the car to drive autonomously. A self-driving car illustrates how sensor data is collected, processed, and used for decision-making. It ensures that the car can navigate safely by understanding its environment and taking appropriate actions.

5.4 USE CASE DIAGRAM

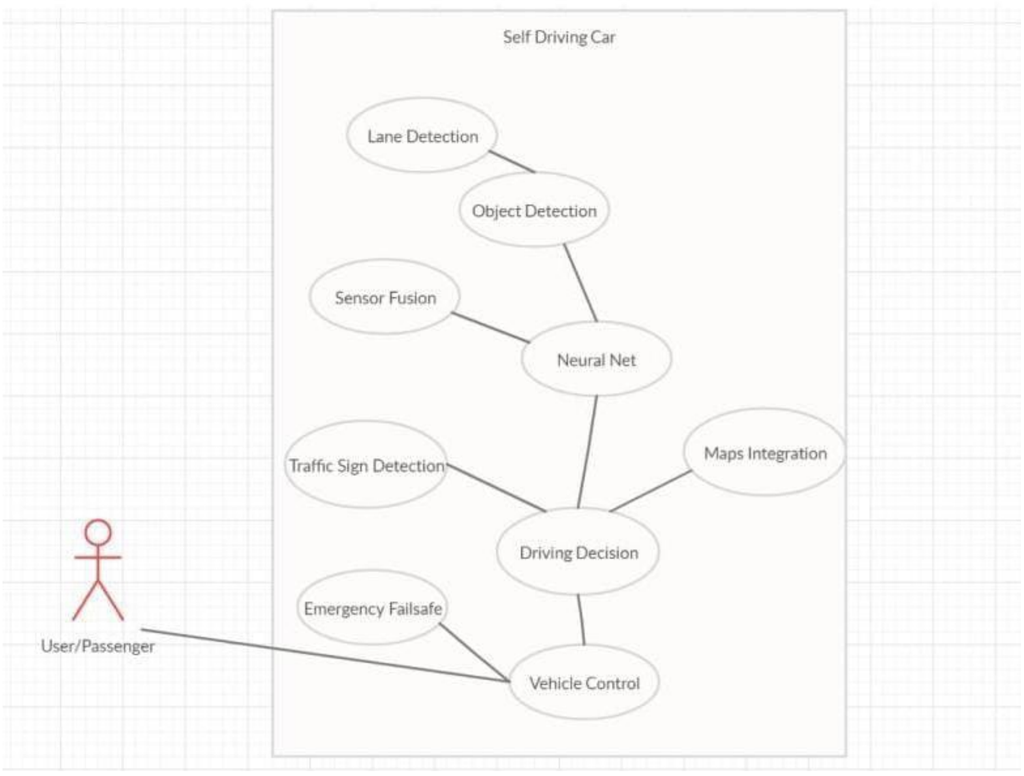


Fig: Use Case Diagram.

A Use Case Diagram for a Self-Driving Car represents the system's functionality from the perspective of users (actors) and the interactions between those users and the system. It helps visualize the high-level requirements and use cases that the self-driving car system should handle.

Key Use Cases:

1. **Start/Stop Car** :- Passenger can start or stop the car.
2. **Select Destination** :- Passenger enters a location.
3. **Autonomous Navigation**: - The car calculates the best route using GPS.
4. **Obstacle Detection & Avoidance** :- The car detects other vehicles, pedestrians, and obstacles.
5. **Follow Traffic Rules** :- The car follows speed limits, signals, and road signs.
6. **Lane Keeping & Changing** :- The car ensures it stays in the correct lane or changes lanes safely.
7. **Emerging Handling** :- The system responds to accidents or failures by alerting emergency services.
8. **Self-Parking** :- The car autonomously parks itself.
9. **Vehicle Health Monitoring** :- The system checks for issues and sends reports to the maintenance system.
10. **Weather Adaptation** :- The car adjusts driving based on rain, fog, or snow conditions.

The actors (Passenger, GPS System, Traffic Control System, etc.) are represented as stick figures .The use cases (Start/Stop Car, Navigate, Avoid Obstacles, etc.) are represented as ovals.

CHAPTER-6: SOFTWARE ENVIRONMENT

A self-driving car's software environment is a complex system that integrates multiple technologies, including artificial intelligence (AI), sensor processing, real-time decision-making, and communication systems. The software stack is typically divided into different layers to ensure safe and efficient autonomous driving.

1. Perception Layer (Sensor Data Processing)

This layer processes raw data from sensors to detect and recognize the environment.

Key Components:

- Sensor Fusion – Combines data from LiDAR, cameras, radar, and ultrasonic sensors.
- Object Detection & Classification – Uses AI and machine learning to identify pedestrians, vehicles, road signs, etc.
- Localization – Uses GPS, IMU (Inertial Measurement Unit), and SLAM (Simultaneous Localization and Mapping) to determine the car’s position.

Software & Frameworks:

- ROS (Robot Operating System)
- OpenCV (Computer Vision)
- TensorFlow/PyTorch (Deep Learning)
- NVIDIA CUDA (GPU Acceleration.

2. Planning & Decision-Making Layer

This layer generates the driving plan based on sensor inputs.

Key Components:

- Path Planning – Calculates the optimal route based on maps and real-time traffic.
- Obstacle Avoidance – Adjusts path to avoid collisions.
- Behavior Prediction – Predicts movements of pedestrians and vehicles.

Software & Frameworks:

- Apollo Planning (Baidu)
- CARLA Simulator
- Open AI Gym (Reinforcement Learning)

3. Control Layer

This layer translates planned actions into vehicle movements.

Key Components:

- Steering Control – Adjusts steering based on path.
- Throttle & Brake Control – Manages acceleration and deceleration.
- Lane Keeping & Following – Uses PID controllers and AI-based decision-making.

Software & Frameworks:

- MATLAB/Simulink (Simulation & Control Systems)
- Auto ware (Open-source Autonomous Driving Software)

4. Connectivity & Cloud Layer

This layer provides external communication and data exchange.

Key Components:

- V2X Communication (Vehicle-to-Everything) – Communicates with traffic signals, other cars, and infrastructure.
- Cloud Data Storage & Processing – Logs driving data for analysis and improvements.
- Remote Monitoring & Updates – Sends software updates and collects diagnostics.

Software & Frameworks:

- AWS IoT Greengrass / Microsoft Azure IoT
- MQTT (Message Queuing Telemetry Transport)
- 5G / DSRC (Dedicated Short-Range Communication)

5. Simulation & Testing Environment

Before deploying on real roads, self-driving software is tested in virtual environments.

Software & Tools:

- CARLA (Open-source Autonomous Driving Simulator)
- Gazebo (Robot Simulation)
- LGSVL (Self-driving Car Simulator)
- Unreal Engine / Unity (3D Simulations)

CHAPTER-7: IMPLEMENTATION

```
import os
import sys
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import time  # to time the learning process
import json  # to get the configuration of the environment
from environments.simple_road_env import Road
from brains.simple_brains import MC
from brains.simple_brains import QLearningTable
from brains.simple_brains import SarsaTable
from brains.simple_brains import ExpectedSarsa
from brains.simple_brains import SarsaLambdaTable
from brains.simple_brains import DP
from brains.simple_DQN_tensorflow import DeepQNetwork
from brains.simple_DQN_pytorch import Agent
from collections import deque
import math
from utils.logger import Logger

# seed = np.random.seed(0)
plt.rcParams['figure.figsize'] = [20, 10]
np.set_printoptions(formatter={'float': lambda x: "{0:0.2f}".format(x)})



def train_agent(using_tkinter, agent, method, gamma, learning_rate, eps_start, eps_end, eps_decay,
                window_success, threshold_success, returns_list, steps_counter_list, info_training,
                max_nb_episodes, max_nb_steps, sleep_time, folder_name=""):
    """

    :param using_tkinter: [bool] to display the environment, or not
    :param agent: [brain object]
    :param method: [string] value-based learning method - either sarsa or q-learning
    :param gamma: [float between 0 and 1] discount factor
    If gamma is closer to one, the agent will consider future rewards with greater weight,
    willing to delay the reward.
    :param learning_rate: [float between 0 and 1] - Non-constant learning rate must be used?
    :param eps_start: [float]
    :param eps_end: [float]
    :param eps_decay: [float]
    :param window_success: [int]
    :param threshold_success: [float] to solve the env, = average score over the last x scores, where x =
window_success
    :param returns_list: [list of float]
    :param steps_counter_list: [list of int]
    :param info_training: [dict]
    :param max_nb_episodes: [int] limit of training episodes
    :param max_nb_steps: [int] maximum number of timesteps per episode
    :param sleep_time: [int] sleep_time between two steps [ms]
    :param folder_name: [string] to distinguish between runs during hyper-parameter tuning
    :return: [list] returns_list - to be displayed
    """
    returns_window = deque(maxlen=window_success)  # last x scores, where x = window_success

    # probability of random choice for epsilon-greedy action selection
    greedy_epsilon = eps_start

    # record for each episode:
    # steps_counter_list = []  # number of steps in each episode - look if some get to max_nb_steps
    # returns_list = []  # return in each episode
    best_trajectories_list = []

    # track maximum return
    max_return = -math.inf  # to be set low enough (setting max_nb_steps * max_cost_per_step should do
it)
    max_window = -np.inf

    # initialize updated variable
    current_action = None
    next_observation = None

    # measure the running time
    time_start = time.time()
    nb_episodes_seen = max_nb_episodes
    #
    for episode_id in range(max_nb_episodes):  # limit the number of episodes during training
        # while episode_id < max_nb_episodes
        # episode_id = episode_id + 1

        # reset metrics
        step_counter = max_nb_steps  # length of episode
        return_of_episode = 0  # = score
        trajectory = []  # sort of replay-memory, just for debugging
        rewards = []
        actions = []
        changes_in_state = 0
        reward = 0
        next_action = None

        # reset the environment for a new episode
        current_observation, masked_actions_list = env.reset()  # initial observation = initial state

        # for sarsa - agent selects next action based on observation
        if (method == "sarsa") or (method == "sarsa_lambda"):
            current_action = agent.choose_action(current_observation, masked_actions_list, greedy_epsilon)
            if method == "sarsa_lambda":
                # for sarsa_lambda - initial all zero eligibility trace
                agent.reset_eligibility_trace()

        if method_used == "mc_control":
            # generate an episode by following epsilon-greedy policy
            episode = []
            current_observation, _ = env.reset()
            for step_id in range(max_nb_steps):  # while True
                current_action = agent.choose_action(tuple(current_observation), masked_actions_list,
greedy_epsilon)
                next_observation, reward, termination_flag, masked_actions_list = env.step(current_action)
                return_of_episode += reward

                # a tuple is hashable and can be used in defaultdict
                episode.append((tuple(current_observation), current_action, reward))
                current_observation = next_observation

                if termination_flag:
                    step_counter = step_id
                    steps_counter_list.append(step_id)
                    returns_list.append(return_of_episode)
                    break

            # update the action-value function estimate using the episode
            # print("episode = {}".format(episode))
            # agent.compare_reference_value()
            agent.learn(episode, gamma, learning_rate)

        else:  # TD
            # run episodes
            for step_id in range(max_nb_steps):
                # ToDo: how to penalize the agent that does not terminate the episode?

                # fresh env
                if using_tkinter:
                    env.render(sleep_time)

                if (method == "sarsa") or (method == "sarsa_lambda"):
                    next_observation, reward, termination_flag, masked_actions_list = env.step(current_action)
                    return_of_episode += reward
                    if not termination_flag:  # if done
                        # Online-Policy: Choose an action At+1 following the same e-greedy policy based on
current Q
                        # ToDo: here, we should read the masked_actions_list associated to the next_observation
                        masked_actions_list = env.masking_function(next_observation)
                        next_action = agent.choose_action(next_observation,
masked_actions_list=masked_actions_list,
                                                    greedy_epsilon=greedy_epsilon)

                        # agent learn from this transition
                        agent.learn(current_observation, current_action, reward, next_observation, next_action,
                                termination_flag, gamma, learning_rate)
                        current_observation = next_observation
                        current_action = next_action

                    if termination_flag:  # if done
                        agent.learn(current_observation, current_action, reward, next_observation, next_action,
                                termination_flag, gamma, learning_rate)
                        # ToDo: check it ignore next_observation and next_action
                        step_counter = step_id
                        steps_counter_list.append(step_id)
                        returns_list.append(return_of_episode)
                        break

                elif (method == "q") or (method == "expected_sarsa") or (method == "simple_dqn_pytorch"):
                    current_action = agent.choose_action(current_observation, masked_actions_list,
greedy_epsilon)
                    next_observation, reward, termination_flag, masked_actions_list = env.step(current_action)
                    return_of_episode += reward

                    if method == "q":
                        # agent learn from this transition
                        agent.learn(current_observation, current_action, reward, next_observation,
termination_flag,
                                gamma, learning_rate)

                    elif method == "simple_dqn_pytorch":
                        agent.step(current_observation, current_action, reward, next_observation,
termination_flag)

                    elif method == "expected_sarsa":
                        agent.learn(current_observation, current_action, reward, next_observation,
termination_flag,
                                greedy_epsilon, gamma, learning_rate)

                    else:  # DQN with tensorflow
                        # New: store transition in memory - subsequently to be sampled from
                        agent.store_transition(current_observation, current_action, reward, next_observation
                        # if the number of steps is larger than a threshold, start learn ()
                        if (step_id > 5) and (step_id % 5 == 0):  # for 1 to T
                            # print('learning')
                            # pick up some transitions from the memory and learn from these samples
                            agent.learn()

                    current_observation = next_observation

                    if termination_flag:  # if done
                        step_counter = step_id
                        steps_counter_list.append(step_id)
                        returns_list.append(return_of_episode)
                        # agent.compare_reference_value()

                        break

                # log
                trajectory.append(current_observation)
                trajectory.append(current_action)

                # monitor actions, states and rewards are not constant
                rewards.append(reward)
                actions.append(current_action)
                if not (next_observation[0] == current_observation[0]
                        and next_observation[1] == current_observation[1]):
                    changes_in_state = changes_in_state + 1

            # At this point, the episode is terminated
            # decay epsilon
            greedy_epsilon = max(eps_end, eps_decay * greedy_epsilon)

            # log
            trajectory.append(next_observation)  # final state
            returns_window.append(return_of_episode)  # save most recent score
            if episode_id % 100 == 0:
                time_intermediate = time.time()
                print('\n --- Episode={} ---\n eps={}\n Average Score in returns_window = {:.2f} \n
duration={:.2f}'.format(
            episode_id, greedy_epsilon, np.mean(returns_window), time_intermediate - time_start))
         # agent.print_q_table()

      if episode_id % 20 == 0:
         print('Episode {} / {}. Eps = {}. Total_steps = {}. Return = {}. Max return = {}, Top 10 =
{}'.format(
            episode_id+1, max_nb_episodes, greedy_epsilon, step_counter, return_of_episode,
max_return,
            sorted(returns_list, reverse=True)[:10]))

      if return_of_episode == max_return:
         if trajectory not in best_trajectories_list:
            best_trajectories_list.append(trajectory)
      elif return_of_episode > max_return:
         del best_trajectories_list[:]
         best_trajectories_list.append(trajectory)
         max_return = return_of_episode

      if np.mean(returns_window) > max_window:
         max_window = np.mean(returns_window)

      # test success
      if np.mean(returns_window) >= threshold_success:
         time_stop = time.time()
         print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}, duration={:.2f} [s]'.format(
            episode_id - window_success, np.mean(returns_window), time_stop - time_start))
         info_training["nb_episodes_to_solve"] = episode_id - window_success
         nb_episodes_seen = episode_id
         break

   time_stop = time.time()
   info_training["duration"] = int(time_stop - time_start)
   info_training["nb_episodes_seen"] = nb_episodes_seen
   info_training["final_epsilon"] = greedy_epsilon
   info_training["max_window"] = max_window
   info_training["reference_values"] = agent.compare_reference_value()
```

```
    # where to save the weights
    parent_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    folder = os.path.join(parent_dir, "results/simple_road/" + folder_name)
    if not os.path.exists(folder):
        os.makedirs(folder)
    agent.save_q_table(folder)

    print('End of training')
    print('Best return : %s --- with %s different trajectory(ies)' % (max_return, len(best_trajectories_list)))
    for trajectory in best_trajectories_list:
        print(trajectory)

    if using_tkinter:
        env.destroy()

    # return returns_list, steps_counter_list


def display_results(agent, method_used_to_plot, returns_to_plot, smoothing_window, threshold_success,
                    steps_counter_list_to_plot, display_flag=True, folder_name=""):
    """
    Use to SAVE + plot (optional)
    :param agent:
    :param method_used_to_plot:
    :param returns_to_plot:
    :param smoothing_window:
    :param threshold_success:
    :param steps_counter_list_to_plot:
    :param display_flag:
    :param folder_name:
    :return:
    """
    # where to save the plots
    parent_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    folder = os.path.join(parent_dir, "results/simple_road/" + folder_name)
    if not os.path.exists(folder):
        os.makedirs(folder)

    # plot step_counter for each episode
    plt.figure()
    plt.grid(True)
    plt.xlabel('Episode')
    plt.title("Episode Step_counts over Time (Smoothed over window size
{})".format(smoothing_window))
    plt.ylabel("Episode step_count (Smoothed)")
    steps_smoothed = pd.Series(steps_counter_list_to_plot).rolling(
        smoothing_window, min_periods=smoothing_window).mean()
    plt.plot(steps_counter_list_to_plot, linewidth=0.5)
    plt.plot(steps_smoothed, linewidth=2.0)
    plt.savefig(folder + "step_counter.png", dpi=800)
    if display_flag:
        plt.show()

    plt.figure()
    plt.grid(True)
    returns_smoothed = pd.Series(returns_to_plot).rolling(smoothing_window,
min_periods=smoothing_window).mean()
    plt.plot(returns_to_plot, linewidth=0.5)
    plt.plot(returns_smoothed, linewidth=2.0)
    plt.axhline(y=threshold_success, color='r', linestyle='-')
    plt.xlabel("Episode")
    plt.ylabel("Episode Return(Smoothed)")
    plt.title("Episode Return over Time (Smoothed over window size {})".format(smoothing_window))
    plt.savefig(folder + "return.png", dpi=800)
    if display_flag:
        plt.show()

    # bins = range(min(returns_to_plot), max(returns_to_plot) + 1, 1)
    plt.figure()
    plt.hist(returns_to_plot, norm_hist=True, bins=100)
    plt.ylabel('reward distribution')
    if display_flag:
        plt.show()
    agent.print_q_table()
    if method_used_to_plot not in ["simple_dqn_tensorflow", "simple_dqn_pytorch", "mc_control"]:
        agent.plot_q_table(folder, display_flag)
        agent.plot_optimal_actions_at_each_position(folder, display_flag)



def test_agent(using_tkinter_test, agent, returns_list, nb_episodes=1, max_nb_steps=20,
sleep_time=0.001,
               weight_file_name="q_table.pkl"):
    """
    load weights and show one run
    :param using_tkinter_test: [bool]
    :param agent: [brain object]
    :param returns_list: [float list] - argument passed by reference
    :param nb_episodes: [int]
    :param max_nb_steps: [int]
    :param sleep_time: [float]
    :param weight_file_name: [string]
    :return: -
    """
    grand_parent_dir_test = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    weight_file = os.path.abspath(grand_parent_dir_test + "/results/simple_road/" + weight_file_name)
    if agent.load_q_table(weight_file):

        for episode_id in range(nb_episodes):
            trajectory = []
            # reset the environment for a new episode
            current_observation, masked_actions_list = env.reset()  # initial observation = initial state
            print("{} = initial_observation".format(current_observation))
            score = 0  # initialize the score
            step_id = 0
            while step_id < max_nb_steps:
                step_id += 1
                # fresh env
                if using_tkinter_test:
                    env.render(sleep_time)
                # agent choose current_action based on observation
                greedy_epsilon = 0
                current_action = agent.choose_action(current_observation, masked_actions_list,
greedy_epsilon)

                next_observation, reward, termination_flag, masked_actions_list = env.step(current_action)

                score += reward  # update the score

                trajectory.append(current_observation)
                trajectory.append(current_action)
                trajectory.append(reward)
                trajectory.append(termination_flag)

                # update state
                current_observation = next_observation
                print("\r {}, {}, {}.".format(current_action, reward, termination_flag), end="")
                sys.stdout.flush()
                if termination_flag:  # exit loop if episode finished
                    trajectory.append(next_observation)
                    break

            returns_list.append(score)
            print("\n{}/{} - Return: {}".format(episode_id, nb_episodes, score))
            print("\nTrajectory = {}".format(trajectory))
            # Best trajectory= [[0, 3], 'no_change', [3, 3], 'no_change', [6, 3], 'no_change', [9, 3], 'slow_down',
            # [11, 2], 'no_change', [13, 2], 'speed_up', [16, 3], 'no_change', [19, 3]]

        print("---")
        print("{} = average return".format(np.mean(returns_list)))
    else:
        print("cannot load weight_file at {}".format(weight_file))


if __name__ == "__main__":
    actions_list = ["no_change", "speed_up", "speed_up_up", "slow_down", "slow_down_down"]
    state_features_list = ["position", "velocity"]  # , "obstacle_position"]
```

```
    # the environment
    flag_tkinter = False
    initial_state = [0, 3, 12]
    goal_velocity = 3
    env = Road(flag_tkinter, actions_list, state_features_list, initial_state, goal_velocity)

    # getting the configuration of the test
    env_configuration = vars(env)
    dict_configuration = dict(env_configuration)

    # avoid special types:
    not_to_consider = ["tk", "children", "canvas", "_tclCommands", "master", "_tkloaded",
"colour_action_code",
                       "colour_velocity_code", "origin_coord", "display_canvas", "origin", "_last_child_ids",
"rect",
                       "logger"]
    for elem in not_to_consider:
        if elem in dict_configuration:
            del dict_configuration[elem]
    # saving the configuration in a json
    with open('environments/simple_road_env_configuration.json', 'w') as outfile:
        json.dump(dict_configuration, outfile)

    # Different possible algorithms to update the state-action table:
    # -1- Monte-Carlo  # working
    # method_used = "mc_control"  # definitely the faster [in term of duration not nb_episodes]. below 1
min

    # -2- Temporal-Difference  # all are working - "q" performs the best
    method_used = "q"
    # method_used = "sarsa"
    # method_used = "expected_sarsa"
    # method_used = "sarsa_lambda"  # worked with trace_decay=0.3

    # -3- deep TD
    # method_used = "simple_dqn_tensorflow"
    # method_used = "simple_dqn_pytorch"  # model is correct - ToDo: hyper-parameter tuning
    # -4- Model-Based Dynamic Programming
    # Dynamic programming assumes that the agent has full knowledge of the MDP
    # method_used = "DP"

    # Instanciate an Agent
    brain_agent = None
    if method_used == "mc_control":
        brain_agent = MC(actions=actions_list, state=state_features_list, load_q_table=False)
    elif method_used == "q":
        brain_agent = QLearningTable(actions=actions_list, state=state_features_list, load_q_table=False)
    elif method_used == "sarsa":
        brain_agent = SarsaTable(actions=actions_list, state=state_features_list, load_q_table=False)
    elif method_used == "expected_sarsa":
        brain_agent = ExpectedSarsa(actions=actions_list, state=state_features_list, load_q_table=False)
    elif method_used == "sarsa_lambda":
        brain_agent = SarsaLambdaTable(actions=actions_list, state=state_features_list,
load_q_table=False,
                                   trace_decay=0.3)
    elif method_used == "simple_dqn_tensorflow":
        grand_parent_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
        results_dir = os.path.abspath(grand_parent_dir + "/results/simple_road/")
        print("results_dir = {}".format(results_dir))
        brain_agent = DeepQNetwork(actions=actions_list,
                                   state=state_features_list,
                                   learning_rate=0.01,
                                   reward_decay=0.9,
                                   # e_greedy=0.9,
                                   replace_target_iter=300,  # replace net parameters every X learning
                                   memory_size=50,
                                   summaries_dir=results_dir,
                                   # saver_dir='/tmp/tensorflow_logs/RL/saver/'
                                   saver_dir=None
                                   )
    elif method_used == "simple_dqn_pytorch":
        brain_agent = Agent(actions=actions_list, state=state_features_list)

    elif method_used == "DP":
        # make sure it does not do any training or testing (it has all its methods are implemented internally)
        brain_agent = DP(actions=actions_list, state=state_features_list, env=env, gamma=0.9)

        # ToDo: Problem at state = [3, 3]
        # q_values_for_this_state = [8.40 9.23 -inf 4.76 -2.11] makes the agent go for speed_up => Best =
17 (not 18)

        # check the interface with the environment through specific values
        final_state = [19, 3]
        final_action = "no_change"
        next_observation_dp,reward_dp,termination_flag_dp =
brain_agent.get_value_from_state(final_state, final_action)
        action = "no_change"
        obstacle_state = [12, 2]
        next_observation_dp,reward_dp,termination_flag_dp =
brain_agent.get_value_from_state(obstacle_state, action)
        print(" {}, {}, {} = results".format(next_observation_dp, reward_dp, termination_flag_dp))

        # compare value_iteration and policy_iteration
        opt_policy_pi, opt_v_table_pi = brain_agent.policy_iteration()
        np.save('opt_policy_pi.npy', opt_policy_pi)
        np.save('opt_v_table_pi.npy', opt_v_table_pi)
        opt_q_table_pi = brain_agent.q_from_v(opt_v_table_pi)
        np.save('opt_q_table_pi.npy', opt_q_table_pi)
        print("final_state_values p_i = {}".format(opt_q_table_pi[final_state[0]][final_state[1]]))
        print(opt_v_table_pi)
        print(opt_q_table_pi)

        # opt_policy_pi = np.load('opt_policy_pi.npy')
        return_of_episode_pi, trajectory_pi = brain_agent.run_policy(opt_policy_pi, [0, 3])
        print("p_i has return = {} for trajectory = {}".format(return_of_episode_pi, trajectory_pi))

        print("\n --- \n")
        opt_policy_vi, opt_v_table_vi = brain_agent.value_iteration()
        np.save('opt_policy_vi.npy', opt_policy_vi)
        np.save('opt_v_table_vi.npy', opt_v_table_vi)
        opt_q_table_vi = brain_agent.q_from_v(opt_v_table_vi)
        np.save('opt_q_table_vi.npy', opt_q_table_vi)
        print("final_state_values v_i = {}".format(opt_q_table_vi[final_state[0]][final_state[1]]))
        print(opt_v_table_vi)
        print(opt_q_table_vi)

        return_of_episode_vi, trajectory_vi = brain_agent.run_policy(opt_policy_vi, [0, 3])
        print("v_i has return = {} for trajectory = {}".format(return_of_episode_vi, trajectory_vi))

    # Training and/or Testing
    flag_training_once = True
    flag_testing = False
    flag_training_hyper_parameter_tuning = False  # Tkinter is not used when tuning hyper-parameters
    display_learning_results = False  # only used for training_once

    # for testing
    max_nb_steps_testing = 50
    nb_tests = 10
    sleep_time_between_steps_testing = 0.5  # slow to see the steps

    # for learning
    # hyper-parameters
    gamma_learning = 0.99
    learning_rate_learning = 0.02
    eps_start_learning = 1.0
    eps_end_training = 0.01
    # reach eps_end at episode_id = log10(eps_end/eps_start) / log10(eps_decay)
    # 0.99907 for 5000 at 0.01/1.0
    eps_decay_training = 0.998466
    # eps_decay_training = 0.99907  # - when 70000 episode
    # 0.99907  # for getting to 0.01 in ~5000 episodes

    # to reach eps_end at episode episode_id, eps_decay = (eps_end / eps_start) ** (1/episode_id)
    max_nb_episodes_training = 7000
    max_nb_steps_training = 25
    sleep_time_between_steps_learning = 0.0005

    # success conditions
```

```
    window_success_res = 100
    threshold_success_training = 17
    dict_info_training = {}
    # 22.97 for self.reward = 1 + self.reward / max(self.rewards_dict.values())
    # q_max = 9.23562904132267 for expected_sarsa

    if flag_training_hyper_parameter_tuning:

        # No tkinter used
        learning_rate_list = [0.003, 0.01, 0.03, 0.1, 0.3, 1]

        gamma_learning_list = [0.1, 0.3, 0.5, 0.7, 0.9, 0.95, 0.99, 1]
        nb_episodes_to_plateau_list = [300, 500, 800, 1000, 3000, 5000]
        # [0.954992586021, 0.9847666521101, 0.995405417351, 0.998466120868, 0.9995395890030,
0.9999846495505]
        eps_decay_list = [(eps_end_training / eps_start_learning) ** (1/nb) for nb in
nb_episodes_to_plateau_list]

        for i, param in enumerate(eps_decay_list):
            brain_agent.reset_q_table()  # re-initialize the model!!

            folder_name_training = str(i) + '/'
            logger_name = str(i) + '.log'
            logger = Logger(folder_name_training, logger_name, 0)

            hyper_parameters = (
                method_used,
                gamma_learning,
                learning_rate_learning,
                eps_start_learning,
                eps_end_training,
                param  # decay
            )
            logger.log(str(hyper_parameters), 1)
            # after = Register an alarm callback that is called after a given time.
            # give results as reference
            returns_list_res, steps_counter_list_res = [], []
            dict_info_training = {}

            train_agent(flag_tkinter, brain_agent, *hyper_parameters,
                        window_success_res, threshold_success_training, returns_list_res,
                        steps_counter_list_res, dict_info_training,
                        max_nb_episodes_training, max_nb_steps_training, sleep_time_between_steps_learning,
                        folder_name_training)
            logger.log(dict_info_training, 1)

            try:
                display_results(brain_agent, method_used, returns_list_res, window_success_res,
                                threshold_success_training, steps_counter_list_res,
                                display_flag=False, folder_name=folder_name_training)
            except Exception as e:
                print('Exception = {}'.format(e))

            # testing
            returns_list_testing = []  # passed as a reference
            test_agent(flag_tkinter, brain_agent, returns_list_testing, nb_tests, max_nb_steps_testing,
                       sleep_time_between_steps_learning, folder_name_training + "q_table.pkl")
            logger.log(returns_list_testing, 1)

    if flag_training_once:
        hyper_parameters = (
            method_used,
            gamma_learning,
            learning_rate_learning,
            eps_start_learning,
            eps_end_training,
            eps_decay_training
        )
        print("hyper_parameters = {}".format(hyper_parameters))
        returns_list_res, steps_counter_list_res = [], []
        if flag_tkinter:
            # after(self, time [ms] before execution of func(*args), func=None, *args):
            # !! callback function. No return value can be read
            env.after(100, train_agent, flag_tkinter, brain_agent,
                *hyper_parameters,
                window_success_res, threshold_success_training, returns_list_res,
                steps_counter_list_res, dict_info_training,
                max_nb_episodes_training, max_nb_steps_training, sleep_time_between_steps_learning)
            env.mainloop()
            print("returns_list_res = {}, window_success_res = {}, steps_counter_list_res = {}".format(
              returns_list_res, window_success_res, steps_counter_list_res))
        else:
            train_agent(flag_tkinter, brain_agent, *hyper_parameters,
                 window_success_res, threshold_success_training, returns_list_res,
                 steps_counter_list_res, dict_info_training,
                 max_nb_episodes_training,                              max_nb_steps_training,
sleep_time_between_steps_learning)
        try:
            display_results(brain_agent, method_used, returns_list_res, window_success_res,
                    threshold_success_training, steps_counter_list_res,
                    display_flag=display_learning_results)
        except Exception as e:
            print('Exception = {}'.format(e))
        print("hyper_parameters = {}".format(hyper_parameters))

        # print(brain_agent.reference_list)
    if flag_testing:
        returns_list_testing = []
        if flag_tkinter:
            env.after(100, test_agent, flag_tkinter, brain_agent, returns_list_testing, nb_tests,
max_nb_steps_testing,
                sleep_time_between_steps_testing)
            env.mainloop()
        else:
            test_agent(flag_tkinter, brain_agent, returns_list_testing, nb_tests, max_nb_steps_testing,
                 sleep_time_between_steps_testing)
```

CHAPTER-8: EXPERIMENTS

8.1 EXPERIMENTAL SETUP

The experiment is carried out in simulation program as shown in Figure 6. The simulator simulates seven-lane traffic condition with at most N number vehicles at time step t. Cars with different driving behavior are deployed randomly to interact with agent-controlled car. It helps agent to generalize strategy and policies to maximize speed of self-driving car. Each model is trained with 2000 episodes and each episode consists of at most 4000 frames. During training, the agent makes decision every 4th frame and during evaluation. It allows agent to react to environment with the human response rate. The aim of the experiment is to find the hyperparameters which give the best performance to the self-driving car-agent.

The visual input of the model is environment state of current lane and the adjacent lanes on the left and on the right. The visual input is sufficient for agent to make decision from an egocentric perspective. The visual input detect vehicle up to 20 meters ahead and 10 meters behind.

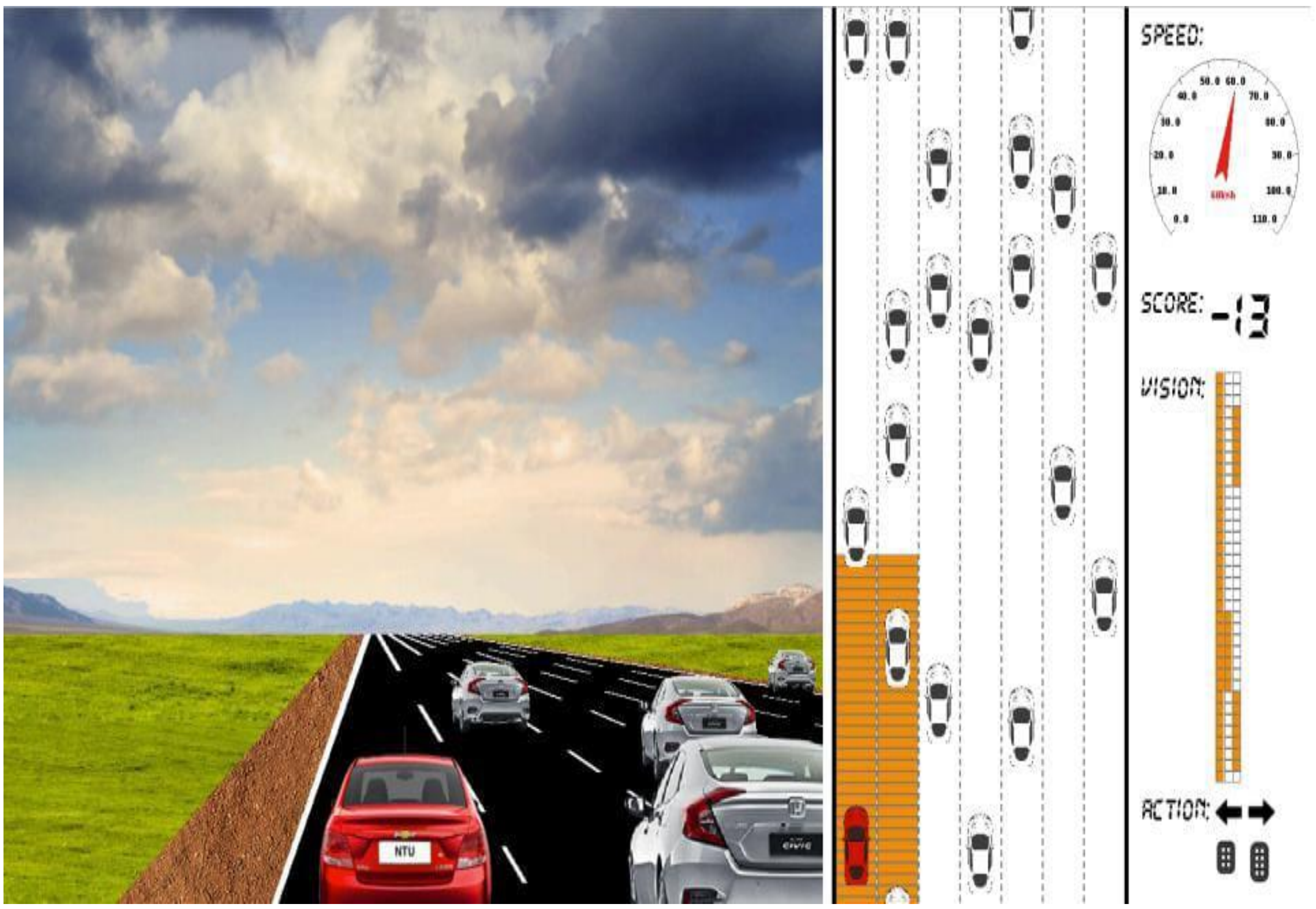
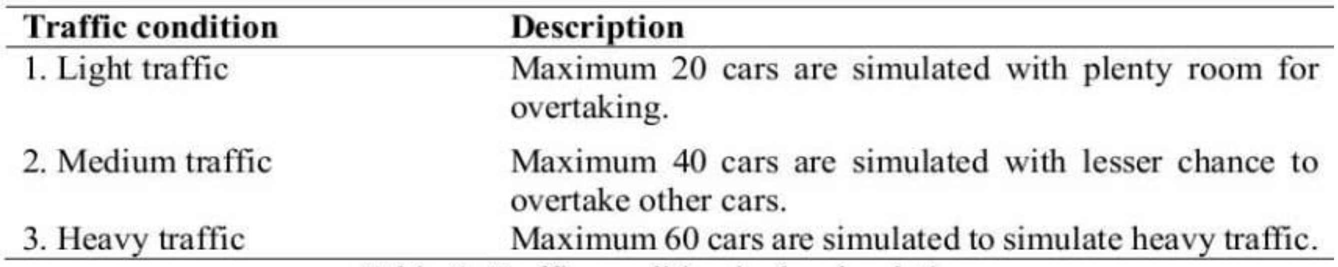


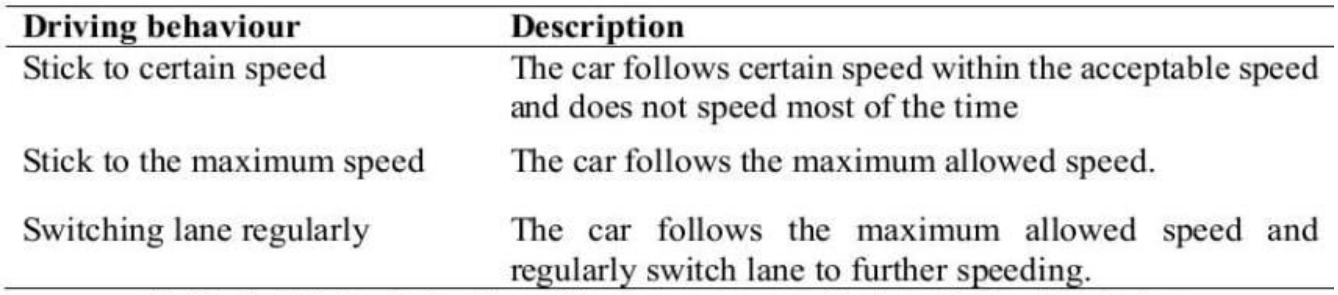
Fig: Screenshot of simulator

The left part of the simulator shows the third-person view of the agent-controlled car(red). The middle part of the simulator shows the bird’s eye view of the road with agent-controlled car (red car). The right side of the simulator shows the speed of the agent-controlled car, current score of the episode, vision input to model, and action taken by the agent.



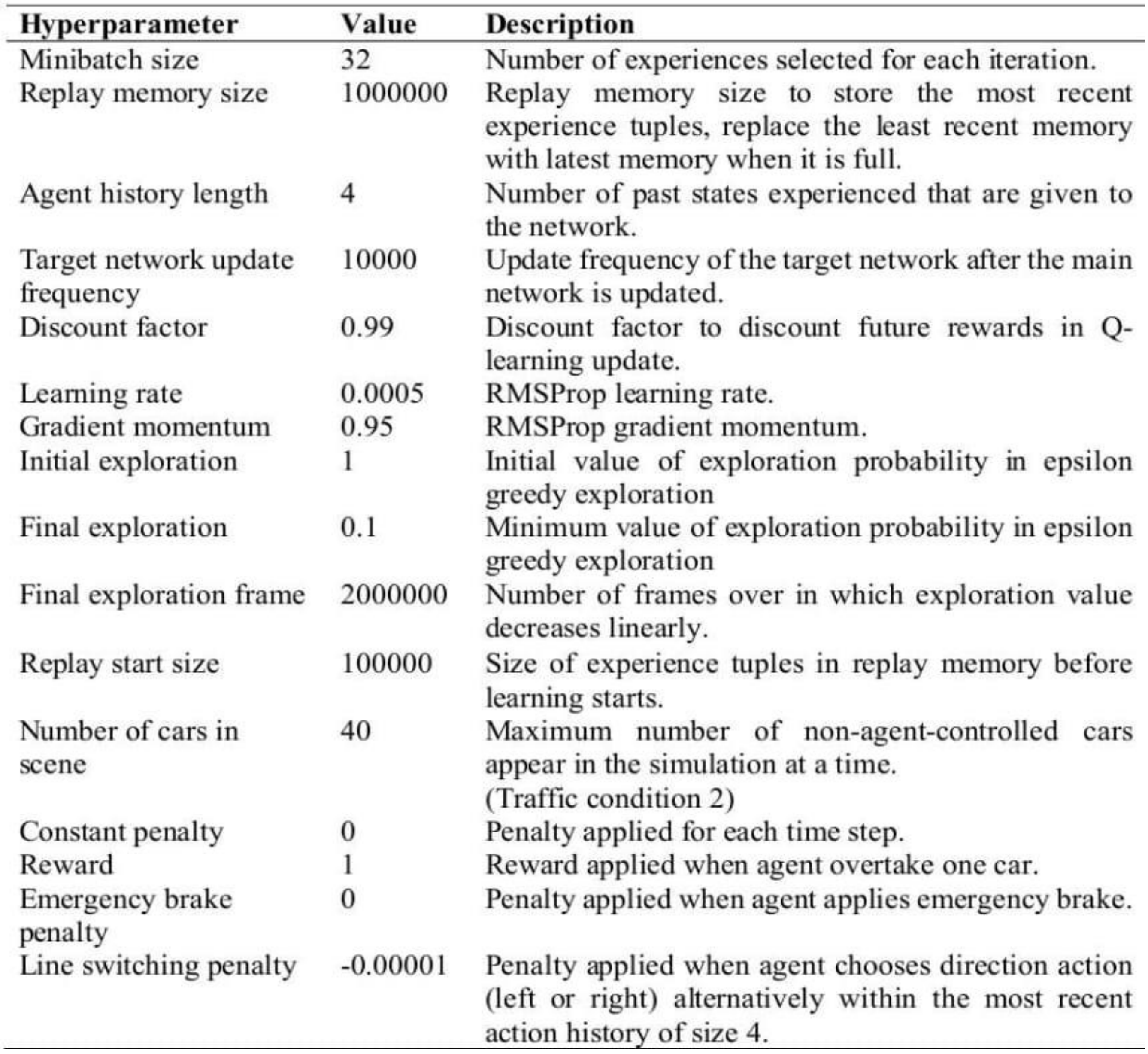
Traffic condition	Description
1. Light traffic	Maximum 20 cars are simulated with plenty room for overtaking.
2. Medium traffic	Maximum 40 cars are simulated with lesser chance to overtake other cars.
3. Heavy traffic	Maximum 60 cars are simulated to simulate heavy traffic.

Table 4: Traffic condition in the simulation.



Driving behaviour	Description
Stick to certain speed	The car follows certain speed within the acceptable speed and does not speed most of the time
Stick to the maximum speed	The car follows the maximum allowed speed.
Switching lane regularly	The car follows the maximum allowed speed and regularly switch lane to further speeding.

Table 5: Driving behaviour of non-agent-controlled car in the simulation.



Hyperparameter	Value	Description
Minibatch size	32	Number of experiences selected for each iteration.
Replay memory size	1000000	Replay memory size to store the most recent experience tuples, replace the least recent memory with latest memory when it is full.
Agent history length	4	Number of past states experienced that are given to the network.
Target network update frequency	10000	Update frequency of the target network after the main network is updated.
Discount factor	0.99	Discount factor to discount future rewards in Q-learning update.
Learning rate	0.0005	RMSProp learning rate.
Gradient momentum	0.95	RMSProp gradient momentum.
Initial exploration	1	Initial value of exploration probability in epsilon greedy exploration
Final exploration	0.1	Minimum value of exploration probability in epsilon greedy exploration
Final exploration frame	2000000	Number of frames over in which exploration value decreases linearly.
Replay start size	100000	Size of experience tuples in replay memory before learning starts.
Number of cars in scene	40	Maximum number of non-agent-controlled cars appear in the simulation at a time. (Traffic condition 2)
Constant penalty	0	Penalty applied for each time step.
Reward	1	Reward applied when agent overtake one car.
Emergency brake penalty	0	Penalty applied when agent applies emergency brake.
Line switching penalty	-0.00001	Penalty applied when agent chooses direction action (left or right) alternatively within the most recent action history of size 4.

Fig: The list of hyperparameters and their values used in the experiments

CHAPTER-9: RESULTS

9.1 OPTIMAL ACTIONS AT EACH POSITIONS

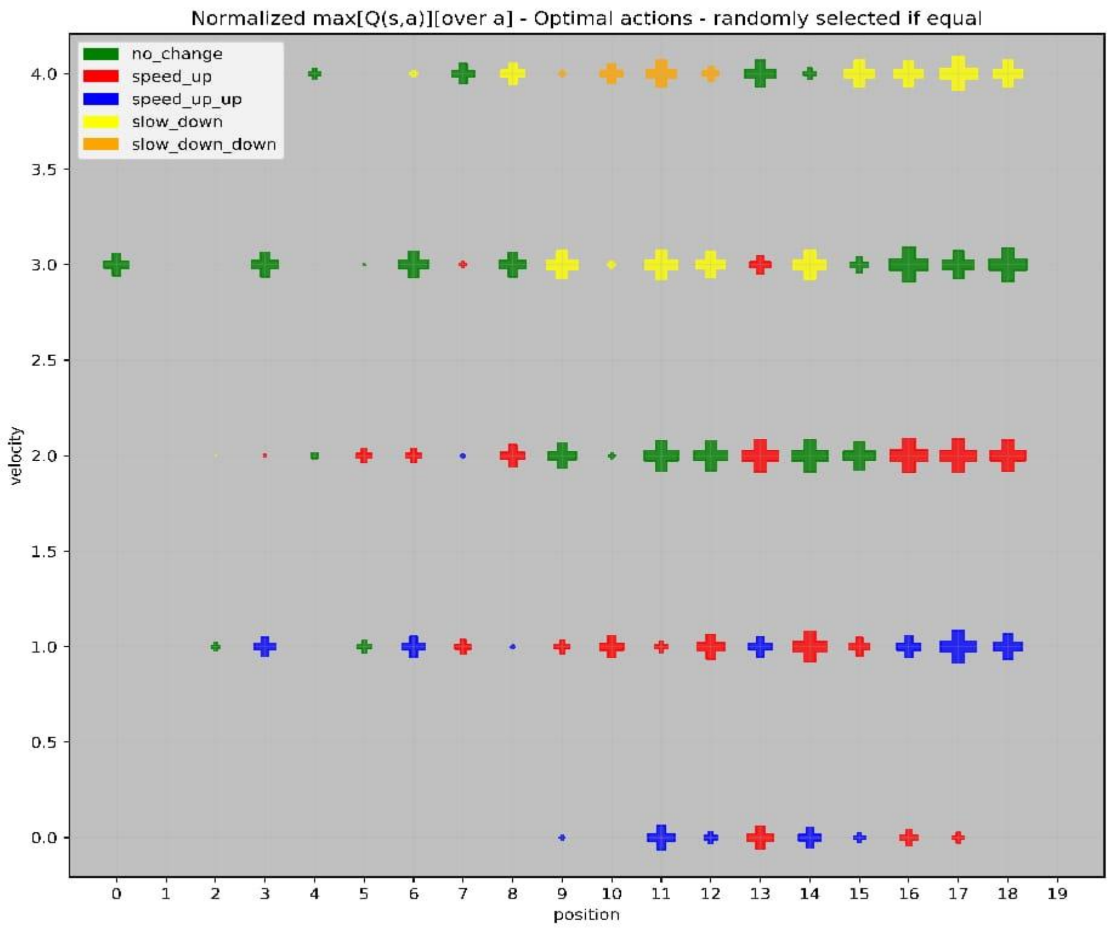


Fig: Optimal Actions

The given plot represents a reinforcement learning (RL) policy for a self-driving car, where actions are selected based on normalized Q-values.This approach helps a self-driving car navigate efficiently by learning an optimal policy through reinforcement learning.

9.2 DISTINGUISHING POSITIVE AND NEGATIVE VALUES WITH MARKER TYPE

This visualization represents a reinforcement learning (RL) policy for a self-driving car, where the agent selects actions based on normalized Q-values at different positions and velocities. The x-axis represents the car's position, while the y-axis denotes its velocity. Each colored marker corresponds to an action: maintaining speed (green), slightly accelerating (red), significantly accelerating (blue), slightly slowing down (yellow), or significantly slowing down (orange). The marker size reflects the magnitude of the Q-value, indicating how strongly the model prefers a given action. Larger markers represent higher confidence in the action choice, while smaller markers suggest uncertainty. The pattern shows that at low velocities, the model prefers acceleration, while at high velocities, it tends to slow down. In mid-range velocities, the car often maintains its speed or makes minor adjustments. As the car progresses along the track, the decisions become more pronounced, likely adapting to upcoming road conditions. This approach helps the self-driving car learn an optimal driving strategy, balancing speed and control based on its environment.

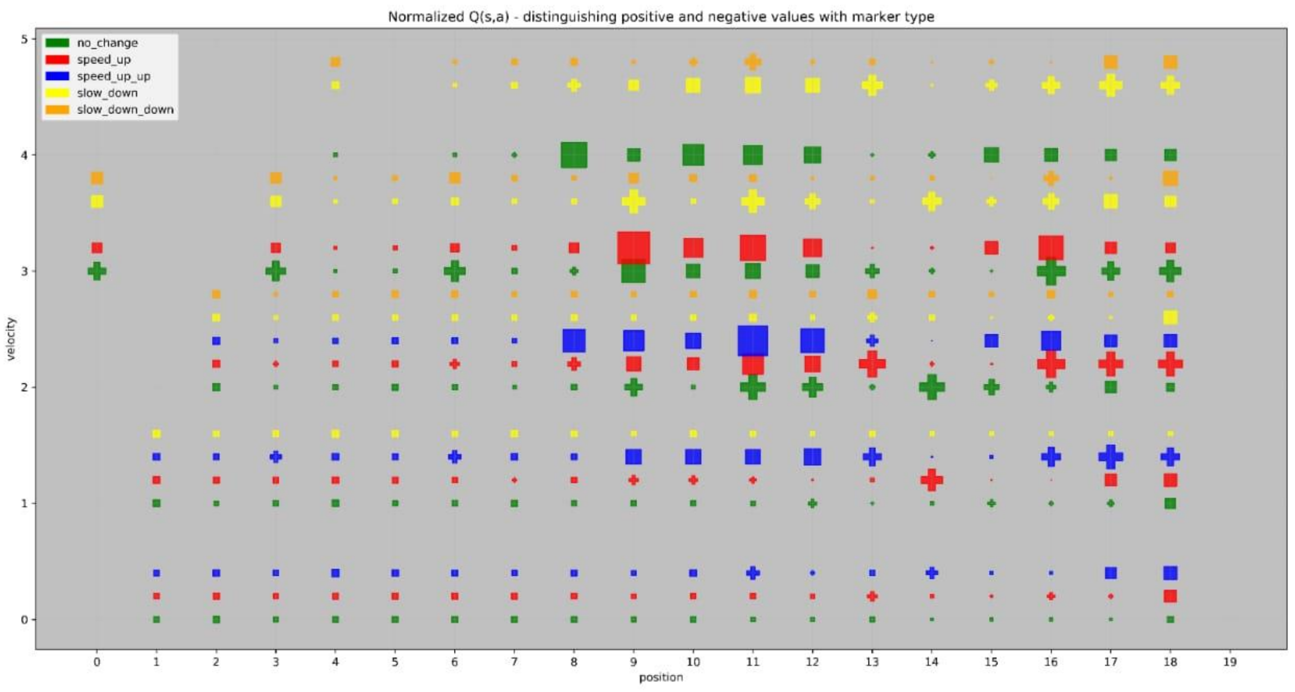


Fig : Distinguishing positive and negative values

9.3 COMPARES POLICY ITERATION AND VALUE ITERATION

This visualization compares Policy Iteration (PI) and Value Iteration (VI), two reinforcement learning (RL) methods used to find optimal driving strategies for a self-driving car. The left two heatmaps display the value functions for states (position, velocity) under PI and VI, where brighter colors (yellow-green) indicate higher values and darker colors (blue-purple) represent lower values, likely associated with undesirable states. The right two heatmaps show the optimal policy (best action to take) for each state under PI and VI, where different colors correspond to different driving actions: speeding up, slowing down, or maintaining speed. The patterns in the policy heatmaps suggest that slower velocities require acceleration (yellow, orange), while higher velocities involve more deceleration (blue, purple), ensuring the car maintains control. The similarity between PI and VI indicates that both methods converge to comparable policies, but VI often requires fewer iterations. This analysis helps design autonomous driving policies that balance speed and safety efficiently across different track conditions. . The left two heatmaps display the value functions for states (position, velocity) under PI and VI, where brighter colors (yellow-green) indicate higher values and darker colors (blue-purple) represent lower values, likely associated with undesirable states.

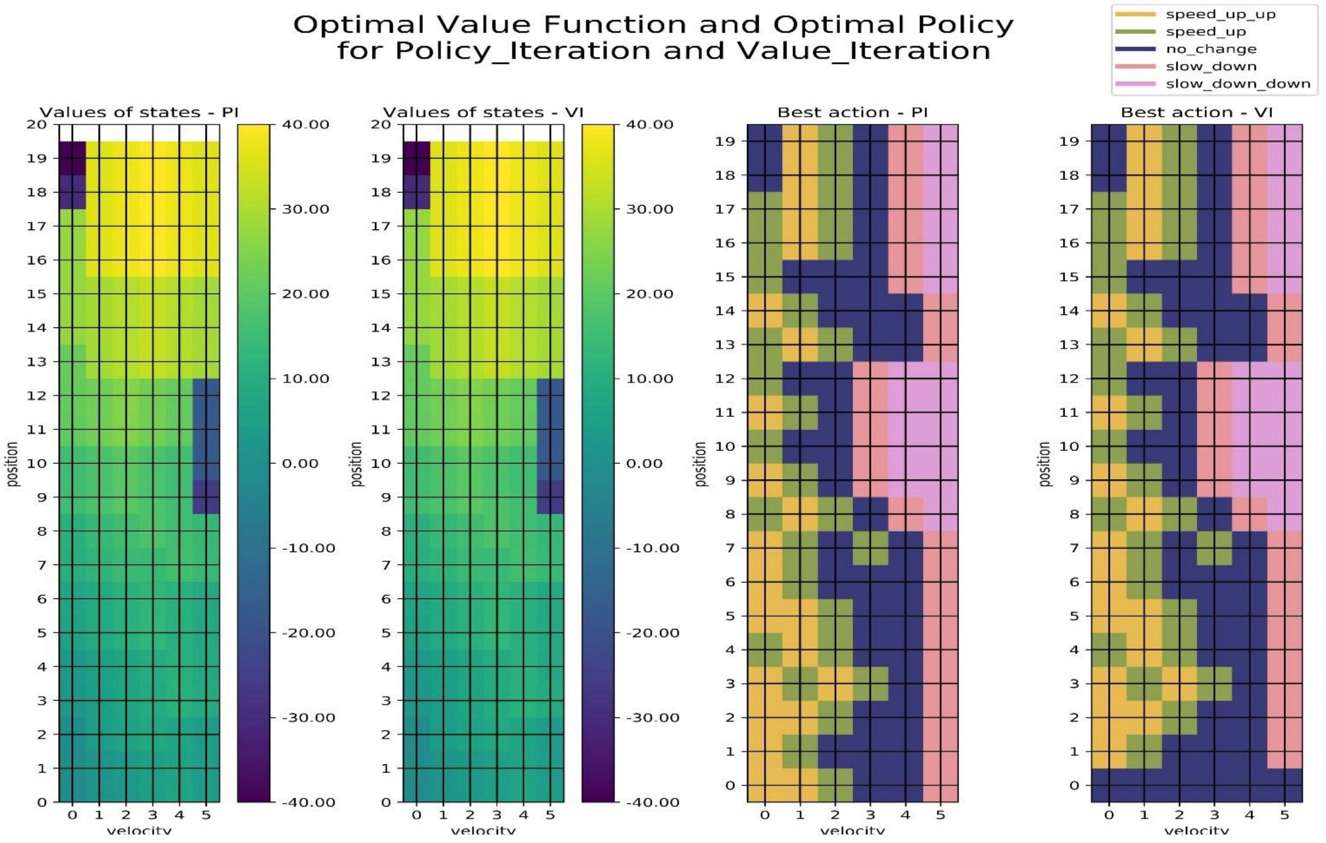


Fig: Optimal value function and optimal policy for policy iteration and value iteration

9.4 TRAINING LOSS

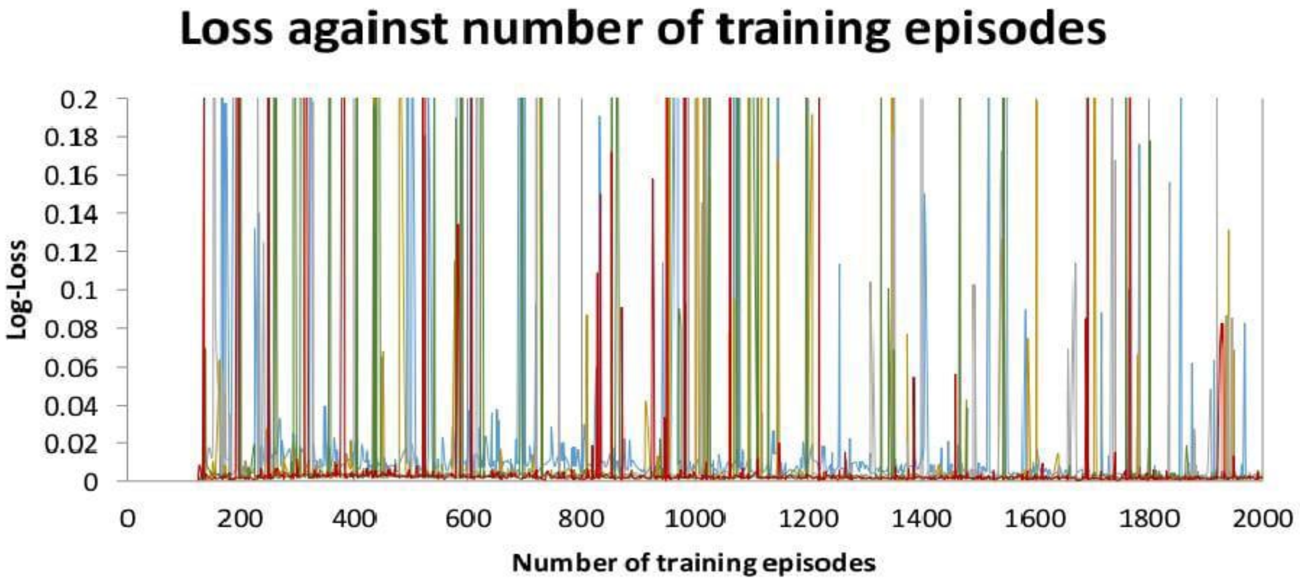


Fig: Log-scaled loss against number of training episodes.

Losses over training is another indicator to show how close a model behaves to the environment. Figure shows the training losses decreases in overall. The spikes in the training losses was resulted by the regular update of the target network to the main network. The models were designed with DDQN methodology and a separated target network was implemented during training to improve the stability of the network. The target network was synchronized with the main network every 10000 trainings.

CHAPTER-10: CONCLUSION

This paper demonstrated the implementation of deep learning model trained with reinforcement learning to maximize speed of self-driving car in multi-lane expressway. This implementation proposed the use of state-of-the-art deep learning technique to approach complex tasks with clear objective. The generated model from Experiment 1 is equipped with high performance in achieving its objective to maximize the speed of autonomous car. The model can be applied with artificial intelligent agents to control an autonomous car in the dynamic environment. The generated models were trained in the simulation environment in which the sensors collect complete and correct data as sensory input.

The models are not tolerated to noises from input data. However, the data from sensory input has higher probability to be interfered, modified, corrupted or lost due to the limitations of sensors in real world environment. Besides that, the proposed model placed the speed maximization as the main objective for the reinforcement learning. However, the user experience of the autonomous car is another important aspect to be prioritized as one of the objectives in reinforcement learning.

The future work can focus on improving user experience as one of the objectives in the implementation of self-driving car-agent. A penalty can be added to the reinforcement learning algorithm to penalize the learning process when user experience degrades during the training phase. The suggested metrics such as average number of emergency brake and average number of lane switching alternatively can be used to portray the user experience of the self-driving car.

Besides that, the future work can also focus on accepting corrupted data as the model input. It is crucial when the external factors such as weather can affect the precision of sensors and noises in collected input. It can improve the stability and accuracy of the model to interact with the dynamic environment on the road. Denoising autoencoders can be implemented to remove noise and extract important features from sensory input with unsupervised learning.

The models from the experiments show that they have average speed from the range from 81.66km/h to 87.96km/h under traffic condition 2. These models have higher average speed compared to baseline model under three traffic conditions. The proposed model used in the experiments is more complex to capture features from the environment states in the simulation.. It shows that the model learnt the most optimal policy to maximize its reward, and subsequently produce the highest average speed. However, the average number of emergency brake applied is higher when the average speed of the autonomous car is higher. It shows the direct proportionality between these two properties. The higher speed of the car led to higher occurrences of emergency brake, which will affect the user experience of autonomous car.

CHAPTER-11: REFERENCES

[1] Sallab, A.E., Abdou, M., Perot, E., and Yogamani, S.: ‘Deep reinforcement learning framework for autonomous driving’, Electronic Imaging, 2017, 2017, (19), pp. 70-76.

[2] Littman, M.L.: ‘Markov games as a framework for multi-agent reinforcement learning’: ‘Machine Learning Proceedings 1994’ (Elsevier, 1994), pp. 157-163.

[3] Mahadevan, S., and Connell, J.: ‘Automatic programming of behavior-based robots using reinforcement learning’, Artificial intelligence, 1992, 55, (2-3), pp. 311-365.

[4] Srivastava, N., Mansimov, E., and Salakhudinov, R.: ‘Unsupervised learning of video representations using lstms’, in Editor (Ed.)^(Eds.): ‘Book Unsupervised learning of video representations using lstms’ (2015, edn.), pp. 843-852.

[5] Donmez, P., Carbonell, J.G., and Schneider, J.: ‘Efficiently learning the accuracy of labeling sources for selective sampling’, in Editor (Ed.)^(Eds.): ‘Book Efficiently learning the accuracy of labeling sources for selective sampling’ (ACM, 2009, edn.), pp. 259-268.

[6] Ng, A.Y.: ‘Shaping and policy search in reinforcement learning’, University of California, Berkeley, 2003.

[7] Hermanns, H.: ‘Markov Chains’: ‘Interactive Markov Chains’ (Springer, 2002), pp. 35- 55.

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M.: ‘Playing atari with deep reinforcement learning’, arXiv preprint arXiv:1312.5602, 2013.

[9] Sutton, R.S.: ‘Learning to predict by the methods of temporal differences’, Machine learning, 1988, 3, (1), pp. 9-44.

[10] Boyan, J.A., and Moore, A.W.: ‘Generalization in reinforcement learning: Safely approximating the value function’, in Editor (Ed.)^(Eds.): ‘Book Generalization in reinforcement learning: Safely approximating the value function’ (1995, edn.), pp. 369- 376.

[11] LeCun, Y., Kavukcuoglu, K., and Farabet, C.: ‘Convolutional networks and applications in vision’, in Editor (Ed.)^(Eds.): ‘Book Convolutional networks and applications in vision’ (IEEE, 2010, edn.), pp. 253-256.

[12] Bakas, S., Zeng, K., Sotiras, A., Rathore, S., Akbari, H., Gaonkar, B., Rozycki, M., Pati, S., and Davazikos, C.: ‘Segmentation of gliomas in multimodal magnetic resonance 28 imaging volumes based on a hybrid generative-discriminative framework’.

[13] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., and Ostrovski, G.: ‘Human-level control through deep reinforcement learning’, Nature, 2015, 518, (7540), pp. 529-533.

[14] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N.: ‘Dueling network architectures for deep reinforcement learning’, arXiv preprint arXiv:1511.06581, 2015.

[15] Bellemare, M.G., Veness, J., and Bowling, M.: ‘Investigating Contingency Awareness Using Atari 2600 Games’, in Editor (Ed.)^(Eds.): ‘Book Investigating Contingency Awareness Using Atari 2600 Games’ (2012, edn.), pp.

[16] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., and Zhang, J.: ‘End to end learning for self-driving cars’, arXiv preprint arXiv:1604.07316, 2016.

[17] https://github.com/songyanho/Reinforcement-Learning-for-Self-Driving-Cars.

[18] Chen, C., Seff, A., Kornhauser, A., and Xiao, J.: ‘Deepdriving: Learning affordance for direct perception in autonomous driving’, in Editor (Ed.)^(Eds.): ‘Book Deepdriving: Learning affordance for direct perception in autonomous driving’ (2015, edn.), pp. 2722-2730.

[19] Yu, A., Palefsky-Smith, R., and Bedi, R.: ‘Deep Reinforcement Learning for Simulated Autonomous Vehicle Control’, Course Project Reports: Winter, 2016, pp. 1-7.