College of Engineering, Design and Physical Sciences

Department of Electronic and Computer Engineering

PhD Systems Engineering

Dissertation Title:

Enhancing Safety in Autonomous Driving through Reinforcement Learning: A Comparative Study of Machine Learning Approaches

Author:

Farshad Mirzarazi

Supervisor: Prof. Alireza Mousavi

Submitted for examination

Submission Date: 30/01/2024

ABSTRACT

With the emergence of autonomous vehicles, the automotive industry promises to revolutionize human mobility, offering many advantages such as increased driving comfort, reduced congestion, and improved road safety. Despite significant advancements in sensor technology and perception algorithms, ensuring the safety of autonomous driving remains a critical challenge. This dissertation aims to explore how reinforcement learning techniques can be leveraged to further enhance the safety of autonomous driving systems.

The study presents an in-depth review of the vast landscape of deep neural network and Reinforcement Learning (RL) methods, emphasizing their limitations in applicability and functional safety. Various modifications to state-of-the-art DQN RL algorithms are proposed, assessing their impact on training stability and agent performance.

An essential contribution of this dissertation is the integration of profound safety body of knowledge, in alignment with automotive safety standards, with advanced machine learning expertise. This work extensively investigates the practical implementation of deep neural network classifiers, identifies safety risks inherent in every development phase, and puts forth both theoretical and practical solutions to address and mitigate these risks.

A safety layer for RL agents, comprising eight key features, is introduced to enhance autonomous driving safety. This includes methods to quantify and optimize exploration behaviour in continuous state spaces. The safety layer integrates human expert guidance, prevents unsafe actions, imposes safety constraints, dynamically shapes rewards, introduces redundancy, and ensures a fail-safe strategy for Operational Design Domain (ODD) violations. An additional method enhances RL agent adaptability by emulating human drivers.

In the final chapter, the study utilizes the OPEN AI Gym environment for highway driving experiments. Reinforcement learning-based agents are equipped with the safety layer features to make real-time decisions in dynamic and varied driving scenarios, and unexpected events. Quantitative comparisons of experimental results are drawn to assess RL agent performance, safety KPIs, and other relevant metrics.

The findings of this dissertation contribute to the ongoing discourse on autonomous vehicles, offering valuable insights into the capabilities and limitations of RL-based autonomous driving systems. This work shall also increase awareness about the criticality of safety in AI-based solutions and guide how such sophisticated solutions comply with normative standards.

ACKNOWLEDGMENT

I would like to express my sincere appreciation to the following individuals for their invaluable support and guidance throughout my Ph.D. journey:

My supervisor, Professor Ali Mousavi, for his trust, constructive feedback, and exceptional mentorship at every stage of my Ph.D. journey, from day one to its culmination.

The late Professor John Stonham and Dr. Hongying Meng for their encouragement and valuable insights provided during numerous panel reviews.

Dr. Sebelan Danishvar for his collaborative efforts in our research activities.

My employer, Robert Bosch-Drive Assistant Systems, for the exceptional opportunities and practical knowledge I acquired in the field of ADAS ECUs, which played a pivotal role in shaping the foundation of my Ph.D. dissertation.

My special thanks also extend to the unnamed scientists and university professors who generously share their knowledge, allowing everyone to benefit and learn from their expertise.

DEDICATION

I dedicate this dissertation to my beloved family. I am deeply grateful for your understanding and patience during the long hours when I was captivated by my research and studies.

Table of Contents

List of Abbreviations

A2C = Advantage Actor-Critic

A3C = Asynchronous Advantage Actor-Critic

AC = Actor-Critic

AD = Autonomous Driving

AdaDelta = Adaptive Delta

Adagrad = Adaptive Gradient Algorithm

ADAM = Adaptive Moment Estimation

ADAS = Advanced Driver Assistance Systems

ADAS = Advanced Driver Assistance System

AI = Artificial Intelligence

ANN = Artificial Neural Networks

AV = Autonomous Vehicle

CNN = Convolutional Neural Networks

ConvNet = Convolutional Neural Network

CPU = Central Processing Unit

DDPG = Deep Deterministic Policy Gradient

DL = Deep Learning

DNN = Deep Neural Network

DOI = Digital Object Identifier

DP = Dynamic Programming

DQN = Deep Q-Network

DT = Decision Tree

ECU = Electronic Control Unit

EM = Exploration Metric

FC Networks = Fully Connected Networks

FN = False Negative

FP = False Positive

FTTI = Fault Tolerance Time Intervals

HAD = Highly Automated Driving

HAZOP = Hazard and Operability

HIL = Hardware-in-the-Loop

ISO = International Organization for Standardization

KL = Kullback-Leibler Divergence

KPI = Key Performance Indicator

Lat = Lateral

LC = Lane Centre

LD = Lateral Deviation

LIDAR = Light Detection and Ranging

LKAS = Lane Keeping Assist system

Long = Longitudinal

LSTM = Long Short-Term Memory networks

MAE = Mean Absolute Error

MARL = Multi-Agent Reinforcement Learning

MC = Monte Carlo

MDP = Markov Decision Process

MIMO = Multi-Input and Multi-Output

ML = Machine Learning

MLP = Multilayer Perceptrons

MNIST = Modified National Institute of Standards and Technology database

MSBE = Mean Squared Bellman Error

MSE = Mean Squared Error

ODD = Operation Design Domain

OEM = Original Equipment Manufacturers

PAS = Publicly Available Specification

PCA = Principal Component Analysis

PER = Prioritized Experience Replay

POMDP = Partially Observable Markov Decision Process

PPO = Proximal Policy Optimization

REINFORCE = REward Increment Factor Offset Reinforcement

ReLU = Rectified Linear Unit

RL = Reinforcement Learning

RMSprop = Root Mean Squared Propagation

RN = Random Number

RNN = Recurrent Neural Networks.

SAE = Society of Automotive Engineers

SARSA = State-Action-Reward-State-Action

SB3 = Stable-baseline3

SGD = Stochastic Gradient Descent

SiL = Software in the Loop

SL = Safety Layer

SISO = Single-Input and Single-Output

SLP = Single-Layer Perceptron.

SOTIF = Safety Of The Intended Functionality

SP = Safety Penalty

SVM = Support Vector Machine

Tanh = Hyperbolic Tangent

TB = Tensorboard

TD = Temporal Difference

TN = True Negative

TP = True Positive

TRPO = Trust Region Policy Optimization

TS = Thompson Sampling

UCB = The Upper Confidence Bound

List of Figures

List of Tables

List of Algorithms

1 Chapter 1 - INTRODUCTION

Motivation

In recent years, the automotive industry has undergone a significant transformation, transitioning from conventional driver assistance systems to fully autonomous driving (AD). This transformation is primarily attributed to substantial advancements in computer and sensor technologies, as well as the valuable contributions of academics to the field of machine learning. Combining a background in electrical engineering (BSc, MSc) and over a decade of experience in automotive industry this thesis embarks on a challenge to improve the performance of autonomous vehicles on roads. The focus of this thesis is to contribute to the rapidly evolving area of autonomous driving technology.

Identification of gaps

Substantial progress has been achieved in the control systems of autonomous driving; however, there is a noticeable gap in the suitability of autonomous driving systems for practical applications, particularly in terms of safety and reliability.

Despite the fact that cutting-edge technologies have been used to realize these solutions, there have been documented instances of accidents resulting in catastrophic consequences, even when operating under favourable weather conditions, with valid sensor data, and within routine driving scenarios. These incidents are characterized by the system's failure to accurately perceive critical factors such as the vehicle's surroundings, obstacles, and right of way. Furthermore, there have been instances where the system has misinterpreted the driver's readiness to resume control.

Deep Learning (DL) solutions are considered state-of-the-art in autonomous driving (AD) systems. However, it is essential to acknowledge the inherent safety risks associated with this technology preference. These risks encompass issues such as a limited capacity for adaptability in unforeseen driving scenarios, a heavy reliance on labelled data for training, and unsafe behaviour when encountering unseen or unexpected driving situations. In addition, the complexity of deep learning architecture, along with substantial design variations, presents significant challenges in ensuring systematic verification, validation, and compliance with regulatory standards.[1]

Research Questions

To enhance the safety of autonomous driving control systems, addressing these gaps is important. Consequently, the following research questions are formulated to specifically target and address the gaps:

[1] Chapter 4 of this dissertation thoroughly explores these risks and offers potential solutions.

1. To what extent are state-of-the-art systems autonomous driving systems suitable for real-world applications, considering factors such as safety, reliability, and regulatory standards?
2. What steps are required to enhance the safety and maturity level of these systems, and how can these improvements be practically implemented?
3. Which Machine Learning Approach is most suitable for safety of Autonomous Driving Systems?
4. Can alternative ML method (e.g. Reinforcement Learning (RL) address the challenges of Outperform Deep Learning in Autonomous Driving?

The aim and objectives of the thesis

To answer these research questions, the primary aim of this study is defined as:

Explore and propose a novel and safe Artificial intelligent-driven solution to solve the safety challenges of autonomous vehicle control driving on public roads.

To accomplish the stated goal, the following objectives have been set:

1. Explore to acquire the necessary subject area knowledge.

2. Develop suitable methods for evaluating and identifying gaps in the current state-of-the-art (comparative analysis).

3. Create an experimental and analytical framework, including a simulation environment, for implementing the proposed methodologies and solutions.

4. Conduct tests and evaluations of the proposed solution in comparison to state-of-the-art and practical implementations.

Hypothesis

The hypothesis of this doctoral dissertation asserts that Reinforcement Learning (RL), when employed as an autonomous driving control solution, can provide significant advantages compared to other machine learning (ML) methods. Reinforcement learning techniques have demonstrated their effectiveness across a wide range of applications, from outperforming even the most skilled human players in games like chess to managing complex industrial systems.

A prominent attribute of RL methods is their remarkable data efficiency, allowing them to learn from limited real-world data and progressively enrich their knowledge base through interaction with their environment. Consequently, RL methods exhibit reduced dependence on expensive labelled data for all possible dynamic driving situations.

In accordance with the paramount principle of prioritizing safety, often referred to as "safety first," this dissertation postulates that the integration of a safety layer into RL-based decision-making agents has the potential to substantially enhance the safety margin of autonomous driving systems, particularly concerning factors like distance and velocity control.

Methodology:

To accomplish each of the objectives of the thesis, following methods are pursued:

Objective 1: Acquiring Subject Area Knowledge

A comprehensive literature review was conducted to deepen the understanding of the subject area. The review covers:

1. Fundamentals of Advanced Driver Assistance System (ADAS) systems and autonomous driving (Chapter 6)
2. Automotive safety standards such as ISO 26262, SOTIF 21448, and ISO PAS 8800 (Chapter 6) consequently mapping the standard with existing body of knowledge and transpose this relationship with the gap in the state-of-the-art and devise a solution to go beyond the state-of-the-art.
3. Fundamentals of machine learning techniques, including SVM, Neural Networks, and Reinforcement Learning (see Chapters 2, 3, 4)

Objective 2: Gaps in state-of-the-art solutions (comparative analysis)

Employ a comparative analysis approach with particular focus on identifying safety risks associated with state-of-the-art Deep Neural Network classifiers. This analysis considered mandatory safety standards, proposing measures to bridge gaps and mitigate risks (see Chapter 6).

Objective 3: Creating Experimental Framework

To fulfil this objective, the necessary experimental frameworks were developed using Python, OPEN AI Gym Environments, and relevant PyTorch machine learning libraries. These frameworks are used to numerically evaluate proposed Reinforcement Learning algorithms based on criteria including training time, stability, agent performance in terms of total rewards, and resource consumption (runtime, memory, etc.) (see Chapters 3 and 5).

Objective 4: Conducting Tests and Evaluations

Tests and evaluations were conducted using the simulation environment, practical implementations, and state-of-the-art systems. Both quantitative and qualitative data were collected and analysed to assess the performance, safety, and reliability of the proposed solutions (see Chapter 7). The principle of software in the loop (SiL) are deployed to validate and verify the contribution in the subject area.

Thesis structure

Chapter 1 – Introduction

Chapter 2 – Data-Driven Systems

The chapter will serve as an essential entry point into the world of machine learning. Within this chapter, the fundamental concepts of data-driven systems and machine learning algorithms that form the basis for the research will be explored. The chapter provides a brief analysis of classical control systems' limitations and emphasizes the notable advantages offered by contemporary data-driven solutions. Additionally, the distinctions between classification and regression and supervised vs unsupervised learning are reviewed.

Furthermore, the chapter delves into the intricate realm of objective functions, investigating the advantages and disadvantages of widely used loss functions such as Mean Squared Error (MSE), Cross-Entropy Loss, Kullback-Leibler Divergence, and Huber Loss. The significance of these functions as fundamental design variations for effective machine learning models is elucidated.

The chapter culminates with a concise overview of machine learning algorithms beyond the scope of reinforcement learning and neural networks, providing insights into support vector machines, decision trees, and other relevant methodologies.

Chapter 3 Neural Networks

This chapter presents a concise yet informative literature review of neural network fundamentals, covering essential aspects such as types of neural networks, training methodologies, and activation functions. It is important to note that this review does not aim to provide an exhaustive, in-depth review but rather a review from a practical point of view.

The primary objective of this review is to acquire a comprehensive understanding of neural networks, serving as a foundational basis for identifying their inherent shortcomings and safety risks. Within this context, Chapter 6 focuses on the identification of safety gaps in neural network applications, thereby contributing to the broader body of knowledge.

Additionally, it is essential to gain foundational knowledge about neural networks for two specific contributions in this dissertation. In Chapter 5, novel variations of the DQN (Deep Q-Network) algorithm will be developed, and in Chapter 7, new reinforcement learning agents will be introduced to address complex highway driving scenarios. The effective development and implementation of these innovations relies on a solid understanding of the foundational principles of neural networks.

Chapter 4 Fundamentals of Reinforcement Learning

This chapter provides a comprehensive literature review of reinforcement learning (RL) methods, which stands as a fundamental component within the context of this doctoral research. A profound understanding of this field of study is an essential step toward accomplishing the objectives of the Ph.D., as objectives, as various RL algorithms will be utilized throughout this dissertation.

In the subsequent subchapters, the wide landscape of RL algorithms will be explored, encompassing their underlying principles, mathematical foundations, and applicability in intelligent systems such as autonomous driving. In this context, various RL algorithms will be discussed, including Q-learning, SARSA, policy gradients, and deep reinforcement learning. In addition, key scientific papers and prominent textbooks relevant to this field will be examined. The strengths and limitations of these methods will be highlighted, with a focus on aspects including sample efficiency, ease of implementation, training stability, and more.

Chapter 5 Optimizing DQN Reinforcement Learning: A Comprehensive Study on the Impact of Variations on performance and training stability

This chapter presents novel adaptations of DQN Reinforcement learning, providing experimental assessments of these adaptations and benchmarking of the impact of these adaptations. The study encompasses variations in the architecture of the neural network integrated into DQN, hyperparameter variations, as well as the examination of different Replay Buffer Sizes. The research extensively investigates and discusses the impact of these variations on the performance of RL agents operating under comparable conditions.

Chapter 6 - Safety Framework for ADAS Systems – Deep Neural Network Classifiers

In this chapter, an intensive exploration unfolds into the safety landscape linked with the integration of state-of-the-art deep neural network (DNN) classifiers within Advanced Driver Assistance System (ADAS) systems. The chapter makes a distinctive contribution by meticulously identifying safety risks inherent in the integration of DNNs within ADAS systems. It presents a novel comparative analysis between DNN classifiers and the mandatory requirements of automotive safety standards, aiming to shed light on unaddressed gaps and provide actionable measures.

After presenting a short overview of ADAS systems,5 levels of autonomous driving, and automotive safety standards such as ISO26262 and SOTIF 21448, a review of influential papers in this field is conducted.

Subsequently, a safety framework for DNN classifiers is introduced. It involves the identification of safety risks across the design, development, and implementation lifecycle, as well as the proposal of instrumental solutions for risk mitigation and compliance with the safety standards.

By presenting this comprehensive safety framework and proposing practical solutions, it contributes to a better estimation of unresolved remaining risks in the deployment of DNN solutions in ADAS systems.

Chapter 7 – Enhancing Safety in RL Agents: The 'Safety Override Layer' for Autonomous Driving

This pivotal chapter introduces a significant development in autonomous driving through the incorporation of a dedicated "Safety Override Layer" with eight key features. Seamlessly integrated into a Reinforcement Learning (RL) agent tailored for enhanced safety, this innovative safety framework strategically maximizes RL training efficacy. It safeguards the agent's valuable prior knowledge, proactively prevents unsafe actions, and significantly increases the safety margin.

Beyond these features, the safety layer actively influences the decision-making process of the RL agent, imposing a safety constraint on agent policy optimization, dynamically shaping rewards based on the safe margins of automated driving, introducing redundant agents, and upholding a fail-safe strategy.

Chapter 8 – Implementing, validating, and Verifying Safe Highway Driving RL agent with a safety layer

The experimentation framework focuses on integrating the Safety Layer (SL) into state-of-the-art Reinforcement Learning (RL) agents—specifically, DQN, PPO, A2C, and DDPG—within the customized Highway-Env environment, with a particular emphasis on highway driving scenarios.

Two safety Key Performance Indicators (KPIs), aligned with the safety margin scheme introduced in Chapter 7, Section 7.3, are established as crucial metrics for evaluating safety performance in different highway driving scenarios and analysing crash incidents caused by RL agents.

The analysis explores the impact on safety metrics when activating specific Safety Layer features, such as SL-Redundant RL agents, SL-Safety Dependent Policy Optimization, and SL-Prevent Unsafe Actions, providing empirical insights into the advantages of each feature in enhancing the safety aspects of autonomous driving.

Chapter 9 – Conclusion, Contributions, and Future Work

This final chapter provides a conclusive summary of the dissertation, highlighting key contributions and findings. It also outlines potential avenues for future research and development in the field.

2 Chapter 2 - Data Driven Systems

2.1 Introduction

Data-driven systems generally refer to intelligent computer-based systems and models that rely on data to make informed decisions. Data-driven approaches range from simple data analysis, visualization, and statistical methods to complex systems that utilize machine learning techniques for classification or regression tasks.

As computational capabilities advanced, data-driven systems began to extend their applicability in real world applications. Some clear examples are autonomous vehicle driving, advanced high-precision automated manufacturing, robotics, and software-oriented adaptive solutions (e.g. satellite navigation systems, text, and image processing). This growth was boosted further when access to large volumes of data, often known as "big data," was made possible by new data acquisition and sensor technologies.

The captured data, large in size and complexity, must be refined to convert to information and knowledge to be useful for data-driven systems. Data mining is the process of applying computer-based technologies to mine information and knowledge out of raw data [1]. The process begins with data pre-processing, where raw data is collected, cleaned, and pre-processed to ensure its quality and relevance (section 2.3). Once prepared, data is extracted, often involving data mining techniques to uncover patterns and correlations.

Machine learning techniques, such as supervised and unsupervised learning (as discussed in Section 2.4), are then employed to develop models that can learn from the data and make predictions (regression) or classifications (as discussed in Section 2.5) based on patterns they have discovered. In the post-processing phase, these models are continuously refined and improved as more data becomes available.

As systems grow in complexity, there is a shift in the paradigm from classical control methods to data-driven approaches for revealing the governing equations of these systems [2].

2.2 Shortcomings of Classic control theory

With the advancement of technology and imposition of further complexity and demand on artefacts/man-made systems, it seems classical model oriented monitor and control solutions are struggling to find accurate, integrated, and reliable control and optimum solutions.

One of the primary challenges of classical control models is that real-world systems are generally nonlinear and multi-dimensional [2]. Classical control systems are best suited for single-input and single-output (SISO) system design and, without neglecting the interactions between variables, may struggle to effectively manage multi-input and multi-output (MIMO) systems, which are more prevalent in real-world applications.

The nonlinearity of systems is also resolved in classical control approaches through ideal approximation and the derivation of simple differential equations [2], indicating another key challenge of classical control methods: their lack of adaptability to unknown dynamics. As a result, tuning of the controller parameters may not be robust to all dynamics, uncertainties and disturbances.

Another challenge of Classical control models, which might be even more difficult to tackle, is that some of the real-world systems are of unknown dynamics, there is a basic lack of known physical law to model the system behaviour [2]. Fields such as neuroscience, epidemiology, and ecology are examples of such systems [2].

2.3 Data Preparation

Data preparation is a crucial phase in the data-driven system's pipeline, as it directly impacts the quality and reliability of the results obtained through data analysis and machine learning. In data processing for data-driven systems, various challenges may arise, including issues like noise, outliers/anomalies, missing values, duplicate data, incorrect data recording, expired data, sensor errors, and more. This section briefly presents solutions and techniques to effectively address these challenges.

noisy data

The presence of noise refers to the distortions of true values caused by random disturbances. To address this issue in signal processing, a range of filtering techniques is employed to remove or reduce the effect of these distortions. In signal processing electronic (hard) filters and mathematical (soft) filters can be utilized for this purpose. The latter category, composed of mathematical algorithms, is specifically designed to manipulate the harmonic components of the signal. Examples of mathematical soft filters include the moving average filter and the Fourier filter [3].

extreme values – outliers -

Extreme values are those values that significantly deviate from the average value of data. To reduce the effect of extreme values, two potential approaches can be taken. One approach involves their removal from the dataset. Alternatively, algorithmic filters or adjustments to model parameters can be employed to reduce the model's sensitivity to outliers [3].

missing values

The issue of missing values frequently arises in data mining, attributed to various underlying factors. In response to this circumstance, several strategies can be employed, including the removal of data objects with missing values, estimation of missing values, their replacement with other available values (e.g., mean, median, potentially weighted), or, when feasible, disregarding them during the analysis [3].

duplicate data

In scenarios involving duplicate data, one potential approach is the consideration of duplicate removal [3]. The existence of duplicate data might be even harmful for the training of data-driven systems, particularly when a maximum number of training iterations (epochs) is planned to train the model.

high-dimensional data

Data size reduction is another form of data pre-processing, often necessary to make it compatible with the signal processing unit. For example, the input layer of a neural network may have limitations on the maximum image size it can handle.

In the context of data pre-processing for data-driven systems, the choice of suitable data pre-processing procedures depends on the unique demands of the application. These procedures may include various techniques, such as data aggregation, data sampling, dimensionality reduction, feature creation and selection, as well as data discretization and binarization, each chosen in accordance with the unique requirements of the task at hand [3].

neural networks for two specific contributions in this disse

2.4 Supervised Learning versus Unsupervised Learning

Supervised learning and unsupervised learning along with reinforcement learning are the key machine learning techniques used in data mining. “Supervised and unsupervised learning methods aim to create algorithms for classification, clustering, or regression” [3].

Supervised Learning

The primary goal of supervised learning is to predict the value (output) of the function for any new object (input) once the model has been trained and learned from the labelled data [3]. Supervised data mining algorithms operate on datasets that have been annotated with labels. In such datasets, each input is associated with a known output or target, which is provided by a domain expert. This data, which is observed and labelled, represents the past experience of the model [2].

The machine learning model is subsequently trained on this labelled data, and during this training process, the model's parameters are updated by minimizing the prediction error on labelled data. The resulting model is then applied for regression or classification tasks when presented with new, and unseen data.

Supervised learning is the most common technique for training for neutral networks and decision trees.

According to [4] the most important supervised algorithms are:

- K-nears neighbours
- Linear regression
- Neural networks
- Support vector machines
- Logistic regression
- Decision trees and random forests

Unsupervised Learning

Unlike supervised learning, in unsupervised learning algorithms, labelled data is not available and the model is adapted to its observations to find patterns in the data in order to determine how to cluster and classify new data [2]. Clustering algorithms, e.g. k-means clustering, are counted as classical examples of unsupervised learning [3].

Solving classification and clustering tasks in high-dimensional spaces is inherently complex [2]. Dimensionality reduction techniques aid unsupervised learning models in uncovering hidden patterns and grouping within the data.

The most important unsupervised algorithms according to [4] are:

- Clustering: k-means, hierarchical cluster analysis
- Association rule learning: Eclat, apriori
- Visualization and dimensionality reduction: kernel PCA, t-distributed

2.5 Regression vs Classification

Classification and regression are two fundamental techniques in machine learning and statistics, each with distinct purposes and characteristics. Classification is used to categorize data points into predefined discrete classes of objects. For example, it can be employed to determine whether an email is spam or not, detected object on the road is a motorcycle or a car. The key aspect of classification is that it deals with discrete and categorical outcomes.

On the other hand, regression is used to predict continuous numerical values or outcomes. Regression models find their roots in the pioneering work of the Sir Francis Galton (1822-1911) on "regression towards the mean" [3]. Regression models establish the relationship between input features and a continuous target variable, enabling the model to make quantitative predictions. For example, regression models are used to forecast prices or predict the longitudinal distance of a car to an obstacle.

According to [3], several commonly used classification algorithms in real-world applications include:

- Decision trees
- Bayesian classifiers/Naive Bayes classifiers
- Neural networks
- Genetic algorithms
- k-nearest neighbour classifier
- Support vector machines

Similarly, several commonly used regression algorithms include:

- Linear Regression
- Polynomial Regression
- Support Vector Regression
- Decision Trees for Regression
- Random Forest Regression

The theory of regression and classification models is incredibly rich and has a solid mathematical foundation, offering deep insights into data analysis and pattern recognition. Moreover, key aspects such as feature selection, model evaluation, and ensemble methods are very interesting related topics, but they are beyond the scope of this dissertation. Interested readers are encouraged to explore these topics further in relevant literature and resources.

2.6 Objective Functions

"An objective function is either a loss or gain function (in specific domains, variously called a reward function, a profit function, a utility function, a fitness function, etc.), in which case it is to be maximized." [5]

Depending on the mathematical optimization task, an objective function is called a loss function or cost function, a utility function or in fitness function [6].

For example, in neural network supervised learning, when the objective function quantifies the prediction error (e.g., mean squared error), it is referred to as a loss function, and the goal is to either globally or locally minimize it. In gradient-based reinforcement learning (as discussed in section 3.10), the objective function embodies the agent's expected reward, and the objective is to maximize it.

Objective functions stand as a fundamental concept in optimization theory. The optimization process culminates when the model's parameters are updated in a manner that leads to the local maximization or minimization of the objective function.

The significance of choosing appropriate objective functions depends on the target tasks and availability of computation resources.

Here are some examples of objective functions of different types and domains:

1. Classification:
    - Classification accuracy: The proportion of correct predictions.
2. Regression and NN supervised learning:
    - Mean squared error (MSE): The average squared difference between the predicted values and the actual values.
    - Mean absolute error (MAE): The average absolute difference between the predicted values and the actual values.
3. Reinforcement learning:
    - Cumulative reward: The total reward the RL agent receives in one episode. (See section 4.3.1)

The choice of algorithm to minimize or maximize the objective function depends on the nature of the objective function itself. The most commonly used algorithms are:

- Gradient descent, (section 3.3.2) or gradient ascent
- Backpropagation (section 3.3)
- and genetic algorithms

2.7 Convergence Criteria and Alternative Stopping Criteria

Convergence criteria are essential in iterative numerical methods, ensuring that the iterative training process reaches a stable and accurate results in terms of model parameters. These criteria determine when to stop the training based on a predefined tolerance level, usually linked to the fulfilment of an objective function.

Certain training algorithms are guaranteed to converge by satisfying the objective function and their convergence can be mathematically, often referred to as convergence theorem, proven. See for example the perceptron convergence theorem in [7]. Nevertheless, this is not always the scenario, and in such cases, monitoring the model's convergence behaviour in terms of its rate or stability or examining the changes in model parameters may suggest alternative, more suitable stopping criteria [7].

For example, the authors in [7] propose an alternative convergence criterion for a backpropagation algorithm:
“The back-propagation algorithm is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small.”

However, they also advise that utilizing such a stopping criterion may lead to premature termination of the training process, potentially before the model parameters have been fully optimized.

To summarize, here are some commonly used convergence or stopping criteria:

- Objective function criterion met (achievement of local or global optima points).
- A predefined maximum number of iterations or epochs is reached
- The improvement in the objective function becomes negligible.
- The change in model parameters falls below a specified threshold.
- A time limit for training is exceeded.
- The Reinforcement Learning agent accumulated a minimum level of rewards.
- The Reinforcement Learning agent completed a specified number of episodes.

2.8 A Short Overview of Important Machine Learning Algorithms

2.8.1 Decision Trees

Decision trees serve as hierarchical models in supervised learning, applicable to both classification and regression tasks. They are composed of nodes, categorized into decision nodes and leaf nodes, interconnected by branches. Decision nodes act as decision points, initiating data splits based on specific attributes and their corresponding decision rules. In contrast, leaf nodes act as endpoints, delivering the final decision or prediction. The branches connecting these nodes symbolize the potential outcomes leading to subsequent nodes or ultimate predictions [1].

Decision trees are straightforward and efficient in computation compared to other machine learning methods. The challenges in decision tree modelling include finding optimal rules and avoiding overly complex sizes to prevent overfitting. Given their inherent simplicity, decision trees may face performance issues, particularly with complex regression problems [8].

2.8.2 Random Forest

The Random Forest algorithm, introduced by L. Breiman in 2001, is an ensemble learning method commonly used for both classification and regression tasks [9]. It operates by constructing a large number of individual decision trees during training and outputs the mode of the classes for classification tasks or the mean prediction for regression tasks [1].

A key aspect of the Random Forest methodology involves the inclusion of bootstrapping and aggregation tasks. In the bootstrapping phase, the algorithm randomly selects subsets from the training data points and features, enabling each tree to be trained on a diverse set of examples. In the subsequent aggregation step, the algorithm combines predictions from multiple trees by averaging results, aiming to improve overall predictive accuracy and strengthen generalization capabilities [1].

The randomness in both the selection of data points and the features considered at each split enhances the model's robustness and generalization [1]. However, a primary limitation of the algorithm is that, similar to the trees they are made of, it tends to have a tendency to overfit [8].

The Random Forest algorithm finds applications in the classification of traffic signs [10], as well as in medical, finance, and other industrial domains.

Interested readers can refer to the original paper [9] and related resources to gain insights into the Random Forest algorithm.

2.8.3 Support Vector Machine (SVM)

Developed by Vladimir Vapnik and his colleagues in the 1960s and later popularized in the 1990s, Support Vector Machine (SVM) algorithm has become a fundamental algorithm widely employed in pattern classification problems and non-linear regression [3]. SVM algorithm exhibits strong generalization performance in pattern recognition without requiring explicit domain knowledge [3].

The key concept behind SVM is to find the optimal hyperplane that maximally and non-linearly separates different classes in the feature space. This hyperplane is determined by support vectors, which are the data points that are closest to the decision boundary.

The ‘kernel trick’ is a technique in SVM for addressing non-linear separation problems by mapping points into a higher-dimensional space. This transformation enables linear classification in the new space, equivalent to non-linear classification in the original space.

An illustrative example of a higher-dimensional space is a circular space defined by its x1 and x2 components where two classes of data are classified. One class of data is located inside the circle of equation $x_1^2 + x_2^2 = 1$ and the other class of data which is located outside the circle. [3]

SVM models have demonstrated efficacy in various applications, including their use in autonomous driving, such as the classification of road severity and the detection of road bumpiness [11].

SVMs are one of the most popular machine learning techniques known for high efficiency and good performance. They exhibit slightly better results compared to deep learning neural networks in domains with limited input training data for supervised learning [7]. However, its application in the context of autonomous driving is beyond the scope of this dissertation. Interested readers can refer to related resources to gain insights into the algorithm.

2.8.4 Neural Network

Please refer to chapter 3.

2.8.5 Reinforcement Learning

Please refer to chapter 4.

3 Chapter 3 - Neural Networks

3.1 Introduction to Neural Networks

Artificial Neural Networks (ANNs), initially proposed in the 1940s and 1950s by pioneers like Warren McCulloch and Walter Pitts, experienced a decline due to various factors, notably hardware limitations and insufficient data availability. However, in the 2000s, there was a renewed interest in neural networks, leading to the rise of "deep learning," which has since emerged as a highly dynamic and influential field driving advancements in modern machine learning [1].

Neural networks are categorized as supervised learning algorithms and can be used for both classifications and regression tasks. They have found extensive practical application in various domains, including industrial and automotive sectors. In healthcare, they are utilized for medical image analysis, enabling the early detection of diseases. In the financial sector, they enhance fraud detection systems by identifying irregular transaction patterns.

In the automotive sector, neural networks have been widely employed in Advanced Driver Assistance Systems (ADAS) to enhance driving comfort and safety. They significantly contribute to features such as adaptive cruise control, lane-keeping assistance, and traffic sign recognition [2].

With access to high-performance hardware, it has become feasible to implement deep learning networks which are state-of-the-art machine learning algorithms for regression and classification [3] with large-scale hidden layers. This unlocks their potential to handle large training datasets and complex tasks.

3.2 Types of Neural Networks

Warren McCulloch and Walter Pitts (1943) introduced the simplest of the neural network models, the Single-Layer Perceptron (SLP), in their seminal paper titled "A Logical Calculus of Ideas Immanent in Nervous Activity".

The SLP is acknowledged by machine learning scientists as a source of inspiration for the development of all types of artificial neural networks [4]. See Figure 3.1.

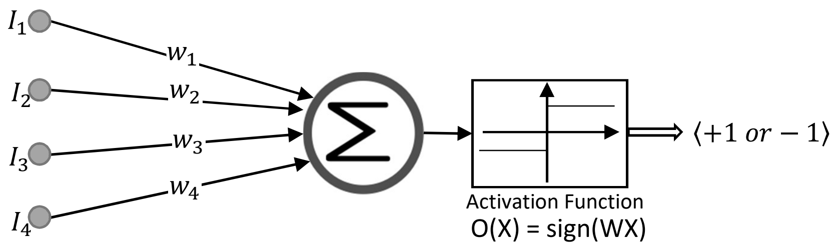


Fig. 3.1: A Single-layer Perceptron Network

A Single Layer Perceptron is a simple binary classification algorithm that can separate two classes of data (like dog or cat) using a linear decision boundary. It consists of input features, weights, and an activation function that produces an output, typically 1 or -1, based on whether the input falls above or below a certain threshold.

The Single Layer Perceptron (SLP) is considered overly simplistic for extracting complex features and is associated with several limitations. One major constraint is its inability to address problems that demand non-linear decision boundaries, as demonstrated by its incapability to solve the exclusive OR (XOR) classification problem [4].

Artificial Neural Networks (ANN) come in various types of architecture, each designed for specific tasks and applications. Some of the most common types of ANNs include:

- Multilayer Perceptron's (MLP) or Fully connected (FC) Networks
  - Commonly used for tasks like image classification and regression.
- Convolutional Neural Networks (CNN)
  - Primarily employed in computer vision tasks such as image recognition and object detection.
- Recurrent Neural Networks (RNN)
  - RNN networks are a type of neural networks that make use of sequential data streams. Sequential data is commonly found in applications like speech recognition, where sentences and phrases exhibit distinct temporal structures to generate meaningful outputs [3].
- Long Short-Term Memory networks (LSTM)
  - LSTMs are a subset of RNNs and used for sequential data processing tasks such as speech recognition [3] and time series prediction [4] (finance forecasting, anomaly prediction, …)
- Transformers
  - Transformers are a type of neural network architecture that utilizes self-attention mechanisms, eliminating the need for recurrent or convolutional structures, originally designed for natural language processing. [5] but successfully utilized in image and video processing [6].
- Deep Learning
  - while not a specific network type, deep learning encompasses a wide range of neural network architectures with multiple layers [4]. Fully connected (FC) and convolutional neural networks (CNN) are among few others the most common types of hidden layers used in deep neural networks.

Preliminary analyses revealed that CNNs and MLPs are most relevant to this dissertation. They will be discussed in more detail in the following sections of this review.

3.2.1 Multi-Layer Perceptron (MLP) Neural Networks

MLPs (Multi-Layer Perceptron): feed-forward neural networks, also known as the multilayer perceptron (MLP) are the most basic form of neural networks. MLPs are used for a wide range of tasks, including regression and classification, and have been enhanced with innovations in training techniques like Batch Normalization [7] and ReLU activations [8] but they don't have built-in mechanisms for handling grid-like data images.

MLP networks consist of input layers, one or more hidden layers with densely connected neurons, and an output layer. Figure 3.2 shows a Multi-Layer Perceptron Network featuring 2 hidden layers.

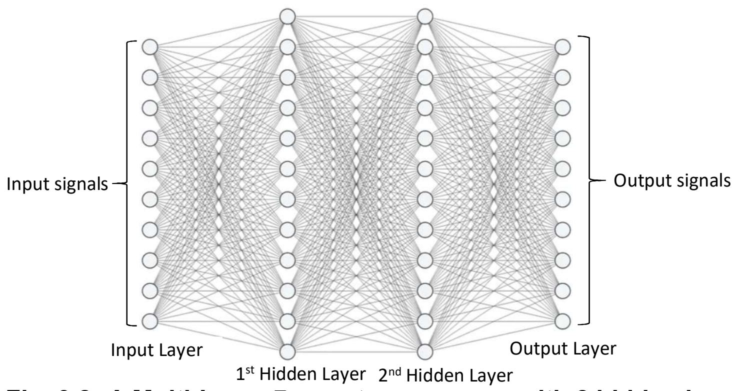


Fig. 3.2: A Multi-Layer Perceptron Network with 2 hidden layer

MLP networks are distinguished by a notable characteristic—the high degree of connectivity between their layers [9]. A Multilayer Perceptron model consists of multiple layers intricately interconnected to form a feed-forward neural network. In this architecture, each neuron within a given layer establishes direct connections with neurons in a distinct layer, allowing for the transmission of information in a forward direction through the network [4]. Due to this characteristic, MLPs are inherently a kind of fully connected (FC) neural network.

In MLP networks, connections between neurons include weighting factors, denoted as w_{ij}, for strength, and each neuron has a bias term denoted as b_i, for non-linear adaptability. These factors contribute to the model's ability to learn complex patterns during training. See section 3.4.

3.2.2 Convolutional Neural Networks (CNN)

Convolutional Neural Networks, abbreviated as ConvNets or CNNs, are comprised of a series of filter layers that are applied to an input image or feature map to extract features. ConvNets are well-suited for image recognition tasks, such as object classification for autonomous driving. They work by extracting features from images using a series of convolution and pooling layers [10].

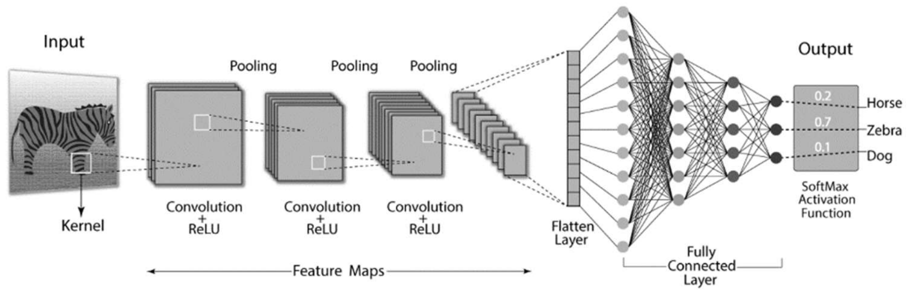


Fig. 3.3: Architecture diagram of a Convolutional Neural Network

Source: https://developersbreach.com/convolution-neural-network-deep-learning/

Figure 3.3 depicts the architecture diagram of a convolutional neural network. The network comprises convolutional, pooling, activation functions, and flatten layers. It also illustrates that a fully connected layer might be employed for classification by flatten layer.

“ConvNets have neurons arranged in three dimensions: width, height, and depth. The neurons in a given layer are only connected to a small region of the prior layer” [4] as illustrated with dotted lines in Figure 3.3.

ConvNets have an advantage over Fully Connected Networks by modelling effectively without the high neuron count typical in FC networks [11].

<u>Convolution layer:</u>

Convolutional filtering involves sliding a filter (kernel) over an input image data to extract features. For example, consider a 4x4 input matrix:

Input Grayscale Image = [[120, 125, 130, 135], [110, 115, 120, 125], [100, 105, 110, 115], [90, 95, 100, 105]]

Filter with Weights = [[0.5, 0.5], [0.5, 0.5]]

The 2x2 filter is applied to the grayscale image to produce a feature map by computing the weighted sum of the corresponding elements.

(120 * 0.5) + (125 * 0.5) + (110 * 0.5) + (115 * 0.5) = 77.5 and so on.

Resulting Feature Map = [[77.5, 82.5], [67.5, 72.5]]

<u>Pooling layer:</u>

In the pooling layer, following convolution layer, local pixel pooling is performed to extract information. The primary objective is to reduce feature map dimensionality, often referred to as downsampling. The downsampling factor, known as stride, determines the extent of downsampling.

For instance, applying a stride of 2 to vectors x1 and x2 below: [11]

x1 = [1, 10, 8, 2, 3, 6, 7, 0, 5, 4, 9, 2]

x2 = [1, 100, 8, 20, 3, 60, 7, 0, 5, 40, 9, 20]

with stride s = 2, the downsampling operation selects every second element from the input will both produce: [1, 8, 3, 7, 5, 9].

This illustrates while downsampling aids in dimensionality reduction, it may discard essential information, especially when representing distinct input states. [11]

Another widely used pooling operation is called Max-Pooling, which selects the maximum value in a window.
For example, applying Max-Pooling to the matrix: x1 = [[5, 7], [3, 8]]
will result in [5, 8], where each element represents the maximum value in its respective window.

3.3 Training of Neural Networks - Backpropagation

The training of neural networks, particularly Multilayer Perceptron (MLPs), is categorized as supervised learning and involves an iterative optimization process with the objective of minimizing an error function [12].

At the core of this process is the backpropagation algorithm, a fundamental technique introduced by Rumelhart et al. in 1986[2]. It is employed to compute the gradient of the network's prediction error function, which is then used to update the network parameters, specifically the weighting factors and biases denoted by $[\vec{w}, \vec{b}]$ in an iterative manner.

Backpropagation starts with initialization of the network parameters by drawing random numbers from a normal (Gaussian) distribution [12]. It encompasses two phases: the forward pass and the backward pass.

In the forward pass of the backpropagation algorithm, input data from the training dataset is propagated through the neural network, passing through each layer's neurons. The weighted sum of inputs, combined with activation functions, yields the network output. The error value, representing the difference between the actual value and the network output value, is used to update the error function (objective function) and its derivative.

In the backward pass, the backpropagation algorithm employs an optimization method such as gradient descent to adjust network parameters based on the derivative of the error function.

The algorithm iterates through successive forward and backward passes until the network training converges, signified by the minimization of the error function to a predefined threshold.

Backpropagation in Convolution Neural Networks [11]

Backpropagation algorithm for Convolution Neural Networks is slightly different. To train a convolutional layer, it is essential to compute the gradients with respect to both its parameters and inputs. Pooling layers lack trainable parameters. Their gradients are computed with respect to the preceding layer.

The training process of a neural network is computationally expensive [3] and faces challenges such as gradient vanishing, network overfitting, and other issues, as discussed in Section 3.6.

[2] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by backpropagating errors.” Cognitive Modeling 5.3 (1988)

3.3.1 Forward Pass

Computation of network output using the current network parameters $[\vec{w}, \vec{b}]$ is formulated as:

$$\hat{y}_k(t) = \sum_k w_k . X_k + b_k \tag{3.3.1}$$

Where X is the input data from the dataset, w represents the weighting factors, and b represents the bias vector.

The Cost function (error function) is defined as:

$$E(t) = \frac{1}{k} \sum_{k=1} (y_k(t) - \hat{y}_k(t)\,)^2 \tag{3.3.2}$$

Which is a mean squared error objective function. The training aims to minimize this function. The error function represents the network performance as it measures the distance of the neural network output from the expected value.

The error function is zero when the network makes a perfect prediction on every training sample [1].

3.3.2 Backward pass: Gradient-Based optimization

In the backpropagation algorithm, the objective is to minimize the error by iteratively adjusting the weights during the backward pass. This involves updating the weights in the opposite direction of the gradient (negative gradient) to reach the minimum of the error function. If the gradient is positive, then the weights must be decremented and vice versa.

For this purpose, gradient descent, a widely employed optimization algorithm, iteratively adjusts model parameters in the steepest descent direction of the objective function [4].

<u>Gradient descent:</u>

The partial derivative of the error function with respect to the weights of the neural network $\frac{\partial E}{\partial W}$, also called the gradient, represents the sensitivity of the error to changes in the weights. It indicates how much the error would change if we make small adjustments to a specific weight.

So the Gradient Descent update rule for the networks weights is defined as:

$$New\ Weight = Old\ Weight - Learning\ Rate \times \frac{\partial E}{\partial W}$$

$$\Rightarrow W_{i+1} = W_i - \eta \times \frac{\partial E}{\partial W} \tag{3.3.3}$$

Where $W_i$is the ith weighting factor parameter of the network η is the learning rate, and $\frac{\partial E}{\partial W}$ is the partial derivative of the cost function E with respect to W_i.

The learning rate (η) significantly influences the training speed of neural networks. Reducing the learning rate leads to more careful training, while higher rates may overlook optimal weight settings. Although a lower learning rate tends to yield better results, it can substantially increase training time. The recommendation is to lower the learning rate during training, considering its impact alongside batch size and data sampling [3].

3.3.3 Optimization Algorithms

Optimization, a critical aspect of neural network training, refers to fine-tuning network parameters to minimize the loss function within the backpropagation algorithm.

Gradient-based methods stand out as the most commonly used techniques for training neural networks, while non-gradient-based optimization methods like genetic algorithms and particle swarm optimization are typically employed for optimizing simpler functions [11].

Among the commonly employed optimization algorithms, gradient descent, as briefly introduced in 3.3.2, plays a prominent role. Beyond gradient descent, various optimizers offer distinct advantages. Notable examples include the Adagrad, RMSprop, and Adam optimizers. The Adam optimizer, widely embraced in deep learning, presents a compelling fusion of the strengths exhibited by both Adagrad and RMSprop optimizers, making it a prominent choice in the optimization landscape.

Adam stands for Adaptive Moment Estimation. The algorithm computes the adaptive learning rates for each parameter and uses the first and second moments of the gradients to adapt the learning rate. This helps in providing a different learning rate for each parameter and hence more precise parameter updates [13].

3.4 Activation Functions

Activation functions are an essential component of the neural network architecture as they shape the output of neurons [14] and enable the neural network to map non-linear input-output relations, provided that the network has enough neurons and layers [11].

This non-linearity allows neural networks to approximate sophisticated nonlinear systems, model complex relationships in data, and enables them to learn and adapt to various patterns, and perform complex decision making.

The choice of activation function impacts the network's learning capacity, and selecting the appropriate function is crucial for successful training and performance. Common activation functions include the sigmoid, hyperbolic tangent (Tanh), Softmax, rectified linear unit (ReLU), and Leaky rectified linear unit (Leaky ReLU).

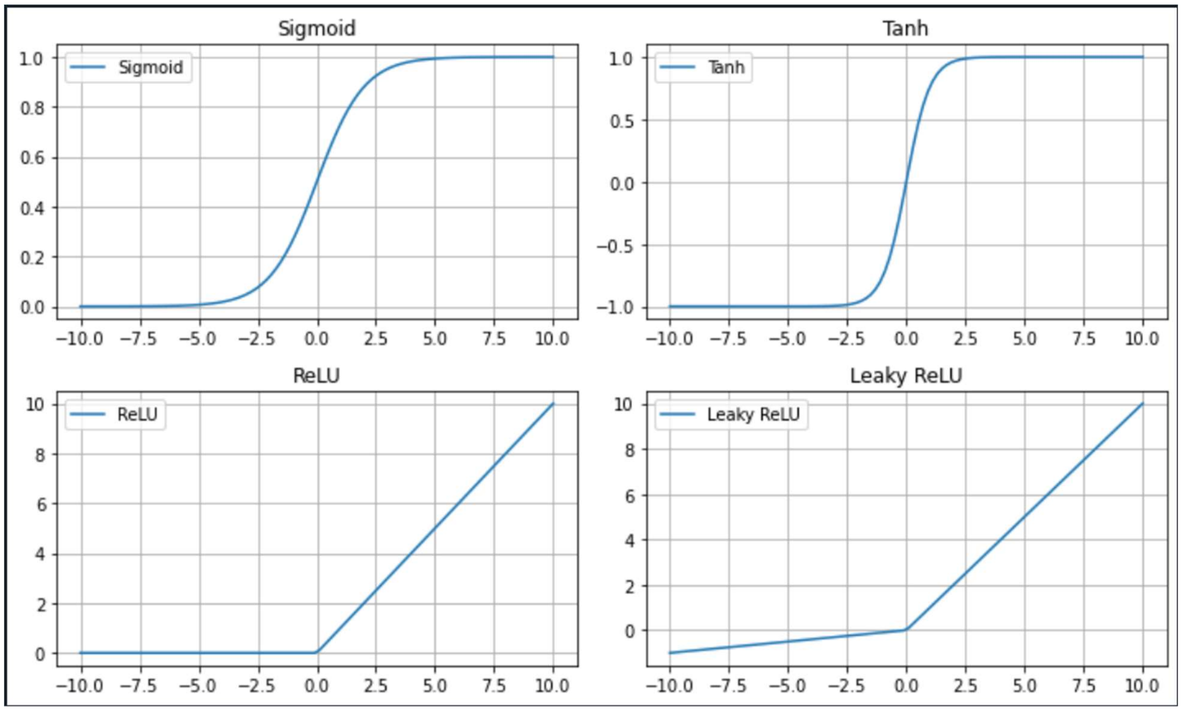


Fig. 3.4: Commonly used Activation functions. (a) Sigmoid, (b) Tanh, (c) ReLU, and (d) Leaky ReLU

Sigmoid function

The sigmoid function, often denoted as the logistic function, is a mathematical curve that takes any number as input and squashes it to an output value between 0 and 1. The slope of the sigmoid function is determined by its time constant τ.

Mathematically, the sigmoid function is represented as:

Sigmoid (x) = $\frac{1}{1+e^{-\tau}}$ (3.4.1)

The sigmoid function smoothly transforms inputs to binary outputs and hence is a suitable choice for binary classification tasks. However, being a squashing function makes it prone to the vanishing gradient problem (see section 3.5), where gradient values become small, potentially causing the network to stop learning [11].

The second problem of the sigmoid function is that the output of the sigmoid is not zero-centred (refer to Figure 3.4 (a)) and therefore, the outputs are consistently positive or negative.

This can lead to a situation where the gradients during backpropagation are also consistently in one direction, either all positive or all negative. This can cause optimization algorithms, like stochastic gradient descent (SGD), to update the weights in a way that results in slow or inefficient convergence.

While sigmoid functions are commonly used, they are not optimal for the reasons mentioned. ReLU and Tanh are typically preferred choices due to their superior performance in various scenarios [14].

Hyperbolic tangent (Tanh) function

The hyperbolic tangent function, abbreviated as Tanh, is a rescaled version of the sigmoid function but unlike sigmoid its output is zero-centred between -1 and +1. As a result, it does not have the optimization problem as for sigmoid function [11].

The Tanh activation function is mathematically expressed as:

$$\text{Tanh}(x) = \frac{2}{1+e^{-2x}} - 1 \qquad (3.4.2)$$

The hyperbolic tangent function closely resembles the identity function near the origin (refer to Figure 3.4 (b)), indicating that for small input values (close to zero), the output is approximately equal to the input. This feature is advantageous in neural networks as it helps retain information of small input values during calculations and hence increases the speed of the gradient descend algorithm [11]. Because of this characteristic the Tanh activation function is preferred over the sigmoid function but it similarly suffers from the gradient vanishing problem [11].

Rectified Linear Unit (ReLU) function

The sigmoid and hyperbolic tangent activation functions are suitable for shallow neural networks with a limited number of layers, encountering challenges in deep neural networks primarily due to the gradient vanishing problem [11]

Nair et al. (2010) introduced the Rectified Linear Unit (ReLU) activation function as an alternative to address the gradient vanishing problem [8]. Unlike sigmoid and hyperbolic tangent, ReLU overcomes this limitation and has become a popular choice in deep neural networks. It does not include any complex and expensive operation like exponential operation and is computationally efficient [11].

The ReLU activation function is mathematically expressed as:

$$\text{ReLU}(x) = \max(0, x) \qquad (3.4.3)$$

ReLU is a non-linear, non-saturating activation function that outputs zero for negative inputs and has a linear slope of 1 for inputs greater than or equal to zero (see Figure 3.4(c)). ReLU has gained a lot of popularity in recent years and apart from simplicity of implementation it is claimed that “networks with ReLUs consistently learn several times faster than equivalents with saturating neurons” [15]. Due to its superior training performance, ReLU is commonly used in hidden layers of neural networks, while either softmax or linear activations are applied to the output layer [13].

Dying ReLU is a significant issue for ReLU activation function and can cause some neurons to die and become useless in a multi-layer neural network. This occurs when a ReLU neuron is stuck in the negative side and its output is always zero [11].

<u>Leaky ReLU function</u>

Leaky ReLU is a variant of the ReLU activation function designed (Maas et al. 2013) to address the "dying ReLU" problem. Unlike traditional ReLU, Leaky ReLU allows a small, non-zero gradient for negative inputs, preventing neurons from becoming completely inactive during training. Refer to Figure 3.4(d).

The Leaky ReLU activation function is mathematically expressed as:

$$LeakyReLU(x) = \begin{cases} ax, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{3.4.4}$$

Where, a is the hyperparameter determining the slope of the function for negative inputs.

This simple modification enhances the robustness of the activation function, making it a popular choice in deep neural networks. In practical application, ReLU and Leaky ReLU often yield comparable results [11].

3.5 Challenges of Neural Networks

Neural networks encounter several challenges despite their capabilities. Some of the well-known challenges include the struggle to find global optima during optimization, managing model complexity, and network overfitting—where models excel on training data but underperform with new inputs. Additionally, the vanishing gradient problem presents a significant challenge to effective training in deep networks.

It is important to note that these challenges represent only a subset, and various other hurdles exist in the complex landscape of neural network development.

<u>Local Optima Instead of Global Optima in Backpropagation</u>

In the training of neural networks, the optimization of a non-linear model often involves utilizing the back-propagation algorithm. The challenge lies in achieving convergence toward a global optimum, introducing the complication of multiple local minima. This complexity can be misleading, making a solution seem effective while failing to reach the global minimum [4]. As a result, the model is trained to a sub-optimal solution only.

<u>Complexity of NN models</u>

Model complexity and a high number of hyperparameters pose serious challenge in artificial neural networks. For instance, in a scenario where data from the MNIST database[3] is processed through a neural network with two hidden layers of 30 neurons each, and a final softmax layer with 10 neurons, the total number of parameters reaches almost 25,000 [1]. This highlights the trade-off between the complexity and simplicity of neural network models.

Vanishing Gradient

In a gradient based learning procedure of a neural network with backpropagation method partial derivatives of the error function is fed back to the network to update the weighting vector. The network learns the new values of weights by analysing how much changes they caused. It is often observed that due to non-optimized initialization of the weights or big jumps in input values or too high learning rate or wrong selection of the activation function the gradients of the error with respect to weights become extremely small and in the continuation of the learning have no influence on updating the weights. This leads to the case where weights are not updated anymore even though learning is not completed and network has not converged yet [4].

Overfitting (overtraining) problem

Overfitting in neural networks occurs, when the model performs well on training data but fails to generalize to new, unseen data. The presence of random noise and outliers poses a risk, potentially causing the model to memorize specific patterns during training, resulting in suboptimal generalization to new data [14].

Overfit models exhibit significant errors when extrapolating. Employing cross-validation techniques is a strategy to mitigate the detrimental impacts of overfitting [3].

[3] The MNIST database is a collection of 28x28 pixel grayscale images of handwritten digits (0 to 9)

4 Chapter 4 – Reinforcement Learning Algorithms

4.1 Introduction to Reinforcement Learning

Within this chapter, an in-depth exploration is conducted into the mathematical and theoretical foundations of reinforcement learning. It must be noted that, in the context of this dissertation, the reinforcement learning algorithms and their foundational formulas are adapted to align with the specific requirements of the study.

Reinforcement Learning (RL) is a remarkable foundation within the field of artificial intelligence, distinguished by its unique and interesting attributes. Unlike other paradigms of machine learning, such as supervised and unsupervised learning, RL replicates the core principles of learning through the direct interaction of an agent with its environment.

Reinforcement learning techniques have demonstrated their effectiveness in a wide range of applications, from outperforming even the most skilled human players in games like chess to managing complex industrial systems.

RL enjoys a solid and well-established mathematical background, providing a rigorous framework for modelling decision-making processes in both stochastic and deterministic scenarios.

Labelled supervised learning as it is being practiced in Deep learning solutions, is known for its rigidity, relying on predefined training data and labels. These solutions can be less adaptable to dynamic and unpredictable real-world scenarios. In contrast, reinforcement learning performs well in these contexts due to its inherent adaptability. RL agent's interaction with the environment enables it to handle dynamic use cases and applications with changing conditions, uncertainties, and unforeseen challenges. This adaptability empowers RL to find optimal solutions in real-time.

When RL techniques combined with other effective machine learning techniques, such as deep reinforcement learning, they can handle high-dimensional and complex data, making it a valuable choice for dynamic, decision-making tasks in fields ranging from autonomous vehicles to robotics and industrial control systems.

4.2 Markov Decision Process (MDP)

Markov Decision Process (MDP) is a mathematical framework for modelling and analysing sequential decision-making problems in stochastic environments. MDPs are widely used in the field of reinforcement learning because they offer a rigorous basis for understanding an agent's interaction and decision-making in uncertain and dynamic environments. Figure 4.1 illustrates an example of an MDP process.

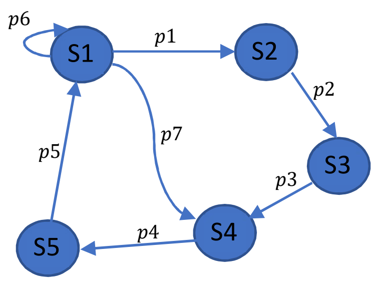


Fig. 4.1: An example of a MDP process

An MDP is defined by following MDP components: [4]

- State space (S):
  All possible states $\forall s \in S$ that the system can take is called the state space. This includes the inclusion of an initial state (s0), as well as one or more terminal states.

- Action space (A):
  All possible actions $\forall a \in A$ that the system can choose in any state: s is called the action space. Actions represent the decisions or moves the agent can make to transit from one state to another.

- State Transition Probabilities (P): transition-function/model/dynamics
  The MDP defines the probability of transitioning from one state to another when an action is taken i.e. $p(s_{t+1}|s_t, a_t)$. These transition probabilities characterize the dynamics of the environment.
  State transition matrix P defines transition probabilities from all states s to all successor states s'.
  $P_{ss'} = p[s_{t+1} = s'|s_t = s]$ (4.1.1)
  $$P = \begin{bmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nn} \end{bmatrix} \quad (4.1.2)$$

- A Reward (Cost) function (R):
  For every state-action combination, there exists a reward signal that offers prompt feedback to the agent. The reward reflects how attractive the action taken by the agent is within a specific state. The objective of the agent is to maximize the total rewards it receives over a sequence of actions.

- Policy (π):
  A policy defines the behaviour of an agent. More specifically, a policy determines how the agent should make action selections within each state.

  In stochastic environment the policy is defined by a probability distribution of selecting an action when the environment is in state: s.
  $A = \pi(a|s) = \pi\big(P(a|s)\big) = P(A_t = a|S_t = s]$ (4.1.3)
  In deterministic environments the policy is a set of pre-defined rules that directly maps each state to a specific action: $A = \pi(s)$ (4.1.4)

Markov property:

MDPs adhere to the Markov property. The Markov property states that in a Markov process, the future state of a system depends only on the current state and is independent of the sequence of events that preceded it [3]. This inherent independence simplifies the modelling of dynamic systems.

A simplified Example of Markov Decision Process in Autonomous Driving:

To illustrate the Markov Decision Process (MDP) in the context of autonomous driving, consider the following simplified example: the environment consists of a road and an autonomous vehicle, and the state is determined based on the position of the ego vehicle, distance to obstacles, and velocity of the vehicle as:

S [0]: the longitudinal distance between the ego vehicle and obstacles on the road

S [1]: the lateral deviation from the lane centre

S [2]: the longitudinal velocity of the ego vehicle

All state variables are derived from sensor inputs.

The autonomous vehicle, acting as the agent, follows its policy and takes actions such as accelerating, braking, or steering.

The objective is to maximize cumulative rewards over time, where positive rewards are earned for the vehicle driving safely without accidents. Negative rewards are incurred if the safety margin decreases, emphasizing the importance of cautious behaviour.

4.3 Fundamentals of Reinforcement Learning

In Reinforcement learning the interaction of the agent with the environment can be summarized as follows: See Figure 4.2.

Observation: The agent observes the current state of the environment, which provides information about the operating condition and situation of the controlled system.

Action: The agent follows its current policy and takes an action from a set of available actions.

Environment response: After the agent takes the action, the environment responds by transitioning to a new state and providing a numerical reward signal.

Learning: The agent uses the observed state, the selected action, and the received reward to update its policy for making better decisions in future.

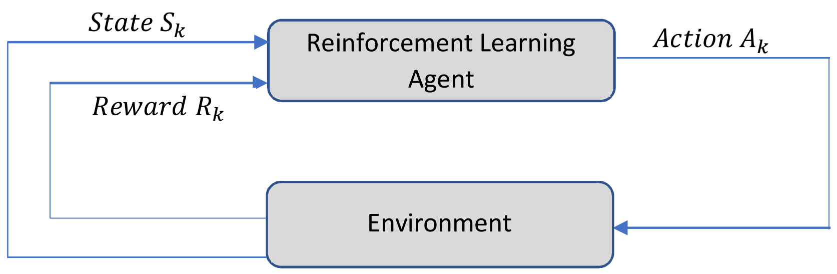


Fig. 4.2: Interaction of an RL agent with environment

4.3.1 RL Terminology and Definitions [1], [2], [5], [6]

Agent: The entity or system that interacts with an environment and actively selects actions and learns from the environment feedback signal to achieve predefined goals.

Environment: The external system with which the agent interacts. It includes the state, actions, and rewards that the agent perceives and can be either stochastic or deterministic.

State (S): see 4.2

Action (A): see 4.2

Reward (R): see 4.2

Policy (π): see 4.2.

Trajectory or Episode(τ): A sequence of states, actions, and rewards that an agent experiences during a particular interaction with the environment.

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots) \qquad (4.1.5)$$

It begins with the initial state-action-reward tuple $(s_0, a_0, r_0)$and continues to capture subsequent transitions (s_1, a_1, r_1) and so on.

RL tasks can be categorized as either episodic, where the task terminates in a defined terminal state, or continuing, where the task has no natural endpoint and continues indefinitely.

Horizon - Episode Length (H): The number of time steps or interactions between the agent and the environment within a single episode. Episodes in RL can have varying lengths depending on the specific task. The horizon H can be finite or infinite.

Discount Factor (γ): A hyper parameter, called gamma; $0 \le \gamma \le 1$, used to discount future rewards in RL to give less importance to distant rewards in the decision-making process.

Return (G_t): The cumulative sum of rewards obtained by the agent, throughout an episode. It is a measure of the total reward received by the agent starting at time step t ($0 \le t \le H-1$) up to the horizon H where the episode ends.

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots + \gamma^{H-1} r_{H-1} = \sum_{i=t}^{H-1} \gamma^{i-t}.r_i \qquad (4.1.6)$$

State Value Function (Vπ(s)): The state-value function estimates the expected cumulative reward (return) that the agent can achieve starting from state: s, and then following policy π.

$$v_\pi(s) = E_\pi[G_t | S_t = s] = E_\pi\,[\sum_{k=0}^{\infty} \gamma^k R_{k+t+1}\,|S_t = s]\ \ for\ all\ s \in S \qquad (4.1.7)$$

The action-value function Q (s, a):

The action-value function, denoted as $Q_\pi(s,a)$, is the expected return starting from state: s, taking action a, and then following policy π.

$$Q_\pi(s,a) = E_\pi[G_t | S_t = s, A_t = a] = E_\pi\,[\sum_{k=0}^{\infty} \gamma^k R_{k+t+1}\,|S_t = s, A_t = a]\ \ for\ all\ s \in S \qquad (4.1.8)$$

Greedy Policy: A greedy policy selects actions which maximizes the expected return based on current value estimates. It tends to exploit the agent's current knowledge.

$$A_t = \underset{a \epsilon A(s)}{\text{argmax}}\, Q_\pi(s,a) \qquad (4.1.9)$$

Policy Evaluation: The process of assessing the quality of a policy by estimating the expected return it can achieve.

Policy Improvement: The process of refining a policy to make it better in terms of maximizing expected return.

Exploration vs. Exploitation: The trade-off between exploring new random actions irrespective of expected reward and exploiting known best actions to achieve the most possible long-term rewards. See 4.4

Temporal Difference (TD) Error: The difference between the estimated value of a state or state-action pair and the value obtained from the actual reward signal is called the temporal difference error.

Convergence: In the context of RL algorithms, convergence refers to the process of the agent's value function (V(s)) or policy (π(s)) becomes progressively more stable over time, ultimately reaching a point of equilibrium, signifying that the agent has learned an optimal or sub-optimal strategy. This plateau indicates that further training or interaction does not yield substantial improvements.

4.3.2 The Bellman Equations

Dynamic programming (DP):

Dynamic programming is a mathematical optimisation approach and an algorithmic paradigm for solving problems by breaking them down into smaller sub-problems and solving them recursively [7].

Richard Bellman (1957), a pioneering mathematician published his influential book "Dynamic Programming ". In this work, he made significant contributions to the field of dynamic programming, particularly in optimizing decision-making in Markov Decision Processes (MDPs).
He derived the Bellman expectation equation and the Bellman optimality equation, which are widely used in Reinforcement learning.
At their core, Bellman equations express the value of a state or state-action pair in terms of the expected cumulative reward that can be obtained from state or state-action pair in a recursive manner [1].

4.3.2.1 Bellman Expectation Equation:

The Bellman expectation equation states that the value function of a state can be decomposed into immediate reward plus discounted value of successor state. [1]

Bellman expectation equation for state-value function $V(s)$:
The state-value function can again be decomposed into immediate reward plus discounted value of successor state.

$$v_\pi(s) = E(R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] = r_{t+1} + \gamma \sum_{s' \in S} p(s'|s,a)\ v_\pi(s') \tag{4.1.10}$$

Where, $v_\pi(s)$ represents the value of a state s. r_{t+1} is the immediate reward obtained in state: s. γ is the discount factor, and P(s' | s, a) is the transition probability from state: s to next state: s' when action a is taken.

Bellman expectation equation for action-value function $Q(s,a)$:

$$Q_\pi(s,a) = E(R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

$$= \gamma \sum_{s' \in S} p(s'|s,a) \sum_{a' \in A} \pi(a'|s') q_\pi(s',a') \qquad (4.1.11)$$

$\Sigma\, \pi(a' \mid s')$ denotes the sum over all possible actions a' in state: s' according to the policy.

4.3.2.2 Bellman optimality equation

The Bellman optimality equation is an extension of the Bellman expectation equation and is used to find the optimal value function and policy by following an optimal policy.

Optimal Value Functions:

The optimal state-value function, denoted as $V_*(s)$, is defined as the maximum state-value function over all policies:

$$V_*(s) = \max_\pi v_\pi(s). \qquad (4.1.12)$$

Similarly, the optimal action-value function, denoted as $Q_*(s,a)$, is defined as the maximum action-value function over all policies:

$$Q_*(s,a) = \max_\pi q_\pi(s,a) \; = \max_\pi E[R_t \,|\, s_t = s, a_t = a, \pi] \qquad (4.1.13)$$

A Markov Decision Process (MDP) is "solved" when the optimal value function is known.

The Bellman optimality equation expresses that the value of a state under an optimal policy must equal the expected return for the best action from that state [1]

Bellman optimality equation for state-value function $V_*(s)$:

$$\begin{aligned} V_*(s) &= \max_a E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)\big(r + \gamma v_*(s')\big) \qquad (4.1.14) \end{aligned}$$

The Bellman optimality equation for action-value function $Q_*(s,a)$:

$$\begin{aligned} Q_*(s,a) &= \; E\left[R_{t+1} + \gamma \max_a Q_*(S_{t+1},a') \middle| S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s',r|s,a)(r + \gamma \max_{a'} Q_*(s',a')) \qquad (4.1.15) \end{aligned}$$

$Q_*(s,a)$ represents the optimal action-value function for state-action pair (s, a).

$\max_{a'} Q_*(s',a')$ denotes the maximum value over all possible actions a' in the next state: s'.

These equations provide a recursive and iterative approach to solving reinforcement learning problems. By starting with initial value estimates and iteratively updating them using the Bellman equations, an agent can learn an optimal policy that maximizes its expected cumulative reward over time. Value iteration and policy iteration are common algorithms that leverage the Bellman equations to solve MDPs and derive optimal policies.

4.3.3 Challenges of Reinforcement learning

While reinforcement learning offers promising solutions, it is essential to recognize the associated challenges. Several notable challenges include:

<u>Exploration vs exploitation Dilemma:</u>
Balancing the exploration of new actions and the exploitation of known, high-reward actions is a fundamental challenge in reinforcement learning. [1] (see 4.3.4)

<u>Delayed reward:</u>
One of the challenge in reinforcement learning is the delayed nature of rewards, where the outcomes of an agent's actions are not immediately evident. This delay can hinder the learning process, as the agent may find it challenging to associate its decisions with their respective consequences in a timely manner. To mitigate this challenge, various workarounds have been proposed, as discussed in reference [8].

<u>Temporal credit assignment in reinforcement learning</u>
Temporal Credit Assignment refers to the challenge of Reinforcement Learning agents in associating actions with their long-term consequences. This challenge becomes more of a serious challenge specially if the reward signal is noisy or delayed and can negatively influence the training process [30].

<u>Partial Observability:</u>
In partially observable Markov decision process (POMDP) environments, the agent does not have full observability of the environment, making it challenging to accurately estimating the state of the environment becomes difficult. Consequently, the agent can only select suboptimal actions. A POMDP is an MDP environment without the Markov property. Playing Cards with RL agent is an example of a POMDP because the RL agent cannot see the opponent’s cards [2].

<u>Addressing RL Challenges for Enhanced Safety in Autonomous Driving</u>

In the context of autonomous driving, these challenges may significantly impede both the training and performance of an RL agent.

If the agent has not sufficiently explored the environment, it may miss out on discovering potentially safer or more efficient driving strategies. Conversely, if the agent over explores, it may struggle to develop a reliable decision-making policy for dynamic driving conditions.

The delayed reward and credit assignment problem present challenges when an RL autonomous driving agent makes a series of correct and accurate decisions in navigating through traffic, but it receives its rewards only at the end of the journey.

Researchers have explored these challenges and proposed solutions to address them. (see for example [30]). However, given the significant impact of these challenges on self-driving car agents, the primary emphasis should be on minimizing their occurrence.

Reward shaping emerges as a suitable approach, leveraging the vehicle's sensors to accurately perceive driving conditions and shaping the reward accordingly.

This approach suggests a promising strategy for efficiently addressing challenges such as delayed rewards, credit assignment problems, and partial observability, offering a valuable concept for further investigation.

4.3.4 Exploration and Exploitation Dilemma

Reinforcement learning (RL) agents must keep the balance between two competing goals: exploration and exploitation. In the literature of RL, exploration refers to the process of trying new random actions in order to learn about the environment and discover new rewards. Exploitation on the other hand is the process of selecting the best actions known to the agent in order to maximize immediate reward.

The exploration-exploitation dilemma is a challenge because neither exploration nor exploitation can be pursued exclusively without failing at the task [1]. If the agent explores too much, it will not accumulate enough reward to learn. If the agent exploits too much, it will miss out on opportunities to discover actions that may even lead to higher rewards.

The exploration-exploitation trade-off is a dilemma because the agent must choose between exploring the environment to discover new rewards and exploiting its current knowledge to maximize immediate reward.

The exploitation is represented by the argmax function. It basically means that the agent takes the only action which maximizes the total reward. $A(t) = \underset{a \in A}{argmax}\ Q(t)$

Example: $A(t) = \underset{a \in A}{argmax}\left(\begin{pmatrix} 0.2 \\ 0.8 \\ 0.1 \end{pmatrix}\right) = a2$

This dilemma has been a subject of intense study by mathematicians and researchers for many decades, but as of now, there is no definitive solution [1]. However, a variety of methods, such as epsilon-greedy exploration, Optimistic Initial Value, Upper Confidence Bound (UCB) exploration, Softmax exploration, and Thompson sampling have been proposed to enhance the balance between exploration and exploitation for RL agents.

4.3.4.1 Action selection: Epsilon-greedy algorithm

The epsilon-greedy algorithm is one of the most widely used strategies for balancing exploration and exploitation in reinforcement learning. The agent selects the best-known action with probability 1-ε (exploitation) and explores a random action with probability ε (exploration).

Epsilon is a hyperparameter that represents the exploration rate. For example, with ε = 0.9, the agent would 90% of the time explore randomly and 10% of its time choose the best-known action.

The value of epsilon (ε) dynamically adjusted over time through a decaying mechanism. The purpose of this decay is to gradually reduce the exploration rate as the agent gains more experience and its estimates of action values become more accurate.

Linear or exponential decay methods are popular strategies for decaying epsilon.

Implementation of the algorithm:

At each time step, the algorithm generates a random number (RN) between 0 and 1. If the generated number is less than or equal to ε, the agent chooses a random action from its action space(exploration). Otherwise, it selects the best known action with highest Q value (exploitation) [5].

$$\pi(a|s) = \begin{cases} random\ action; if\ RN \leq\ \epsilon\ (exploration) \\ \underset{a \in A}{argmax} Q(s,a); if\ RN >\ \epsilon\ (exploitation) \end{cases} \tag{4.1.16}$$

Epsilon decaying:

Linear decaying: $\epsilon(t) = \epsilon_{max} - k.t$ or Exponential decaying: $\epsilon(t) = \epsilon_{max}.e^{-k.t}$ (4.1.17)

Where ε(t) is the exploration rate at time step t, ϵ_{max} is the initial exploration rate, k is a decay rate parameter, and t is the current time step.

4.3.4.2 Action selection: Softmax exploration

Softmax action selection, rooted in the Gibbs or Boltzmann distribution, serves as a technique in Reinforcement Learning enabling agents to make informed choices within uncertain environments. It functions by assigning weights to actions based on their estimated values, leading to a preference for actions perceived as more advantageous.

Let $Q_t(a)$ be the action-value estimate of action $A_t = a$ in a specific state at time step t. Then the probability of selecting action a using softmax is defined as:

$$P\{A_t = a\} = \frac{e^{\frac{Q_t(a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q_t(b)}{\tau}}} \tag{4.1.18}$$

In this approach, the RL agent computes the probability of choosing each action through a softmax function. This function normalizes the exponentials of action values by a temperature parameter τ, which, in turn, regulates the magnitude of the weights and, consequently, influences the agent's exploratory behaviour. A higher temperature weakens the impact of the action value estimate, promoting exploration, while a lower temperature amplifies this impact, favouring exploitation.

Notably, the action probabilities sum to 1, ensuring the accuracy of the selection process.

4.3.4.3 Action selection: Upper confidence bound (UCB)

The Upper Confidence Bound (UCB) algorithm was first developed by Auer, Cesa-Bianchi and Fischer (2002) The algorithm selects the action with the highest upper confidence bound and is stated as follows:

Confidence bound (uncertainty term) = $c\sqrt{\frac{\ln(N_{total})}{N(a)}}$ (4.1.19)

$$A(t) = \frac{max}{a}\left[Q(t) + c\sqrt{\frac{\ln(N_{total})}{N(a)}}\right] \tag{4.1.20}$$

Where c > 0 is a parameter that controls the level of exploration, and N(a) represents the number of times an action has been selected.

The UCB algorithm encourages exploration by adding a term that depends on the number of times an action has been selected and the total number of actions taken. As the action is selected more often, the uncertainty term decreases, leading to more exploitation of actions with higher estimated rewards.

4.3.4.4 Action selection: Thompson Sampling (TS) algorithm

Thompson Sampling (TS) is another notable technique for addressing exploration-exploitation trade-offs and action selection in reinforcement learning. However, this algorithm is not covered in this dissertation. Interested readers can refer to related resources and literature to gain insights into the Thompson Sampling algorithm.

4.4 Taxonomy of RL Algorithms and Methods

In the context of solving reinforcement learning problems and finding the policy, value function V(s), or action-value function Q (s, a) all RL methods are often classified into 3 major categories: [2], [6].

- Model-based or model-free
  In Model-Based RL, agents build an internal environment model for action selection, while Model-Free RL learns directly from interactions without explicit environment modelling.
- Value-based or policy-based
  Value-Based RL estimates the value function for decision-making, while Policy-Based RL directly seeks the optimal policy without necessarily estimating value functions.
- On-policy or off-policy
  On-Policy RL uses one current policy for action selection and updates it based on experiences. Off-Policy RL separates the target policy for updates from the behaviour policy used for action selection.

This dissertation categorizes RL methods based on their training and update mechanisms, and to visually represent these categories, it introduces Figure 4.3's taxonomy diagram. This visual representation provides a structured overview of the diverse approaches employed in the study.

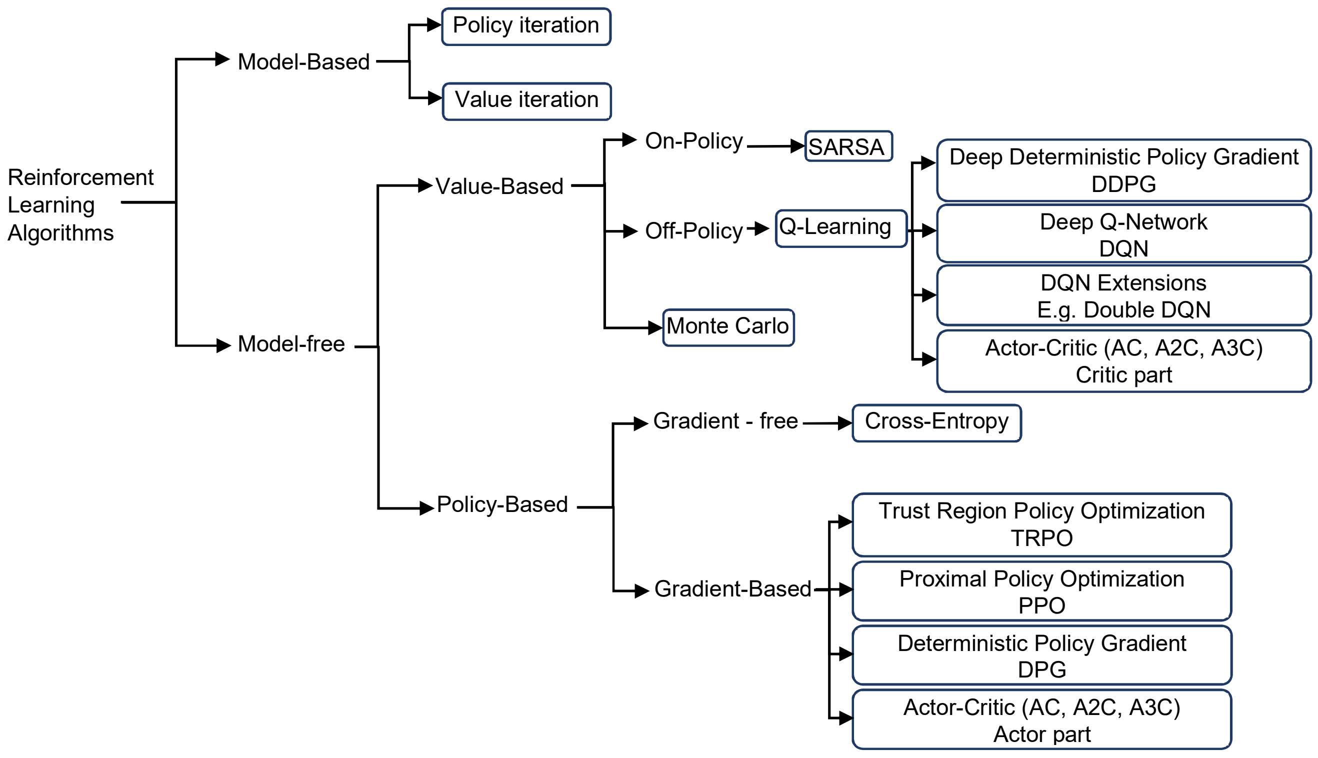

Reinforcement Learning Algorithms
Model-Based
Policy iteration
Value iteration
Model-free
Value-Based
On-Policy
SARSA
Off-Policy
Q-Learning
Monte Carlo
Deep Deterministic Policy Gradient DDPG
Deep Q-Network DQN
DQN Extensions E.g. Double DQN
Actor-Critic (AC, A2C, A3C) Critic part
Policy-Based
Gradient - free
Cross-Entropy
Gradient-Based
Trust Region Policy Optimization TRPO
Proximal Policy Optimization PPO
Deterministic Policy Gradient DPG
Actor-Critic (AC, A2C, A3C) Actor part

Fig. 4.3: A Taxonomy of RL Learning algorithms

Model-Based Methods: Policy Iteration, Value Iteration
Characteristics: Rely on an available model of the environment, including transition probabilities and rewards. See 4.5.1.

Model-Free Methods:
Characteristics: Do not rely on an explicit model of the environment.

Model-Free Monte Carlo Methods:
Characteristics: Estimate values through sampling episodes, model-free. See 4.5.2.

Model-Free Temporal Difference Methods:
Examples: SARSA, Q-Learning
Characteristics: Estimate values through bootstrapping from current estimates, model-free. See 4.5.3.

Policy Approximation Methods:
Example: Deep Q-Network (DQN)
Characteristics: Approximate the optimal policy using function approximation, typically neural networks. See 4.6.

Policy Gradient Methods:
Examples: REINFORCE, TRPO, PPO, etc.
Characteristics: Optimize the policy directly using gradient-based methods. See 4.7.

4.5 Reinforcement Learning Tabular Methods

4.5.1 Model-based: Policy and Value based Iteration methods

Policy Iteration:

Policy Iteration, known for its effectiveness, is an iterative strategy used to compute the optimal policy for a Markov Decision Process (MDP). This method combines policy evaluation and improvement in each iteration.

The Model-based Policy Iteration Algorithm follows these steps:[4]

Training Algorithm of the Model-based Policy Iteration

- The algorithm starts by choosing an arbitrary policy.
  $\pi \leftarrow randomly\ chosen\ a\ policy\ \pi \in \Pi$
- In an iteration loop:
- Under policy π, in state: s, select an action: a using e.g. Epsilon-greedy method Formula: (4.1.16)
- Evaluate the current policy by calculating the state value function V(s) based on Bellman's expectation equation see (4.1.10).
  $$v_\pi(s) = r_{t+1} + \gamma \sum_{s' \in S} p(s'|s,a)\ v_\pi(s')$$
- update the policy (policy improvement) estimates for each state with the largest state-action value based on Bellman's expectation equation see (4.1.12).
  $$\pi_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\big(r + \gamma V(s')\big)$$
- The iteration loop is terminated when the agent's policy converges to its optimal value.

```
if ( $\pi_*(s) == \pi(s)$ ) then
        $break; policy\ converged!$
else update the policy : $\pi \leftarrow \pi_*$
```

Value Iteration:

Value Iteration, known for its implementation simplicity, is an iterative algorithm used to find the optimal value function for a Markov Decision Process (MDP).

The goal of Value Iteration is to determine the best possible cumulative reward an agent can achieve from a given state, considering the environment's dynamics and the agent's actions. Similarly, the Model-based Value Iteration Algorithm follows these steps:

- The algorithm starts with initialization of the state value functions.
- In an iteration loop it updates the value estimates for each state with the largest state-action value.
- The iteration loop is terminated when state values converge to the optimal values.

[4] Adapted from [5]. The algorithm adaptation presented in this dissertation offers a more straightforward implementation.

Table 4.1 compares the value-iteration and policy-iteration methods in reinforcement learning.

	Value Iteration	**Policy Iteration**
Initialization	random value function	random policy
Algorithm complexity	Simple	Complex
Computation cost	More expensive	Cheap
No. Of iteration to converge	takes significantly more iteration to converge	takes fewer iteration to converge
Convergence guarantee	yes	yes

Table.4.1 value-iteration vs policy-iteration methods

4.5.2 Model-free: Monte Carlo Evaluation Method

In reinforcement learning, specifically Monte Carlo (MC) methods, such as first-visit MC and every-visit MC, serve as model-free techniques for estimating state values, $V_\pi(s)$, or state-action functions, $Q_\pi(s,a)$, under a given policy π. The estimation process involves computing the average returns G_t for each episode and updating the estimates at the end of each episode using the Monte Carlo RL Update rule [1].

Monte Carlo RL Update rule: [1]

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{4.1.21}$$

or similarly, for state-action function: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)]$

where $G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$ and α is the time-step constant parameter.

This iterative process refines our understanding of state values and contributes to policy improvement over multiple episodes.

First-visit vs every-visit MC:

Each occurrence of state: s in an episode is called a visit to s. The first-visit MC method computes vπ(s) as the average of returns following the first visits to s, while the every-visit MC method averages returns following all visits to s.

The Model-free Monte Carlo Reinforcement Learning Algorithm follows these steps:[5]

Training Algorithm of the Model-free First-visit Monte Carlo

- State-action function and returns are initialized to zero.
  $Q(S_t, A_t) \leftarrow 0, G_t \leftarrow 0;\ \forall s, a$
- Initialize the count of first-visits to each state-action pair to zero.

  $N(s, a) \leftarrow 0$
- In an iteration loop and for each time-step in the episode:
- Under Policy π, in state: s, select an action: a using e.g. Epsilon-greedy method Formula: (4.1.x)
- If this is a first visit:
  increment the first-visit count, append Returns, and update state-action function.

  $if\,(S_t, A_t)\ is\ the\ first\ visit\ then$

  $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$

  $G_t \leftarrow G_t + R_t$

  Variant 1: $Q(S_t, A_t) \leftarrow \frac{G_t}{N(S_t, A_t)}$ Simple averaging of Returns

  Or

  Variant 2: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{G_t}{N(S_t, A_t)}[G_t - Q(S_t, A_t)]$ with importance sampling
- The iteration loop is terminated when the state-action function converges to its optimal value. (e.g.

In Monte Carlo (MC) reinforcement learning, the episode must finish for calculating the return G_t, which serves as the estimate for action values. Hence, the update of value function in MC learning occurs exclusively at the end of the episode [5].

[5] Adapted from [5]. The algorithm adaptation presented in this dissertation offers a more accessible and straightforward implementation.

4.5.3 Temporal Difference Learning

Bootstrapping Technique in Reinforcement Learning:

Bootstrapping in Reinforcement Learning involves updating state value estimates $V(S_t)$ based on the values of successor states $V(S_{t+1})$, utilizing one or more estimated values in the update step for the same kind of value. This general idea is referred to as bootstrapping [1].

One notable disadvantage of bootstrapping in reinforcement learning is the risk of introducing bias into value estimates due to the iterative updating process and dependency on the estimate of the next state [5].

Temporal Difference (TD) learning:

Previous sections have delved into Dynamic Programming (DP) techniques using the Bellman equations, and Monte Carlo (MC) Methods. Another method, often seen as a combination of DP and MC approaches, is temporal difference learning [1]. The temporal difference learning utilizes the bootstrapping technique to estimate the next value function [5] as stated below:

Temporal Difference RL update rule:

New estimate ← Old estimate + step size [Target – Old estimate]

$$V(S_t) \leftarrow V(S_t) + \alpha \left[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{Traget} - \underbrace{V(S_t)}_{Old\ estimate} \right] \qquad (4.1.22)$$

In most Temporal-Difference (TD) update rules, the TD target is an estimate for the true value of V(s), and also called the TD target.

Temporal difference error estimation is then defined as: TD Target – Old Estimate

$$\sigma_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \qquad (4.1.23)$$

In temporal difference reinforcement learning unlike Monte Carlo method, the updating of Value function and Q function occurs during the same episode.

4.5.4 Temporal Difference Q-Learning and SARSA

Temporal Difference learning serves as the foundation from which Q-learning and SARSA (State-Action-Reward-State-Action) algorithms are derived in reinforcement learning. Both Q-learning and SARSA are instances of TD learning, sharing the common framework of updating value functions based on the difference between the current estimate and a bootstrapped estimate of the future value [5].

Q-learning emphasizes off-policy learning, allowing the agent to learn a policy different from the one it employs for action selections in training, while SARSA embodies on-policy learning, adjusting its policy during learning. Refer to the taxonomy in Figure 4.3 for further clarification.

Update Rule for Q-Learning and SARSA: [5]

Temporal difference:

New estimate ← Old estimate + step size [Target – Old estimate]

<u>Q-learning (Off-policy TD learning):</u>

The Target value for Q-learning is the reward plus the discounted maximum Q-value of the next state, expressed as: $r_t + \gamma \max_a Q(s_{t+1}, a)$

The update rule for Q-learning is:
$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$ (4.1.24)

<u>SARSA (On-policy TD learning):</u>

The Target value for SARSA is the reward plus the discounted estimated Q-value of the next state-action pair, expressed as: $r_t + \gamma Q(s_{t+1}, a_{t+1})$

The update rule for SARSA is:

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$ (4.1.25)

Here, α is the step size (learning rate), $\gamma$is the discount factor, r_t is the reward received at time-step t, s_t and a_t are the state and action at time t.

The training algorithms of TD learning, Q-learning, and SARSA are similar to those covered in the previous section for policy iteration and Monte Carlo learning, with the exception of the update rule. Readers seeking a detailed explanation of these training algorithms can refer to relevant sources.

4.6 Deep Q-Network (DQN) RL Algorithm

Dynamic programming and tabular based reinforcement learning methods such as Mont Carlo, policy iteration, Temporal differences, SARSA, and Q-learning (Watkins, 1989) all suffer from the curse of dimensionality [1].

Tabular reinforcement learning methods are a class of reinforcement learning algorithms that use a table to store the value function for each state-action pair. They are simple and efficient for problems with small state and action spaces, but it becomes intractable as the size of the state space grows.

Real-world complex tasks have more dynamic environments with a very high-dimensional state and action spaces. As the size of the state and action space grows the training of the RL agent becomes computationally expensive resulting in low training speed and very likely leading to only a suboptimal policy. Additionally, tabular methods are sensitive to the order of training and therefore these methods are not suitable to scale to real-world problems.

The paper "Human-level Control Through Deep Reinforcement Learning" by Mnih et al. (2015) from Google DeepMind presents a significant advancement in reinforcement learning using a novel technique called Deep Q-Networks (DQN).

Deep Q-Networks (DQN) RL was a valuable contribution to the field of reinforcement learning. They demonstrated the remarkable performance of DQN on a variety of Atari games and achieved human-level performance. See Figure 4.4.

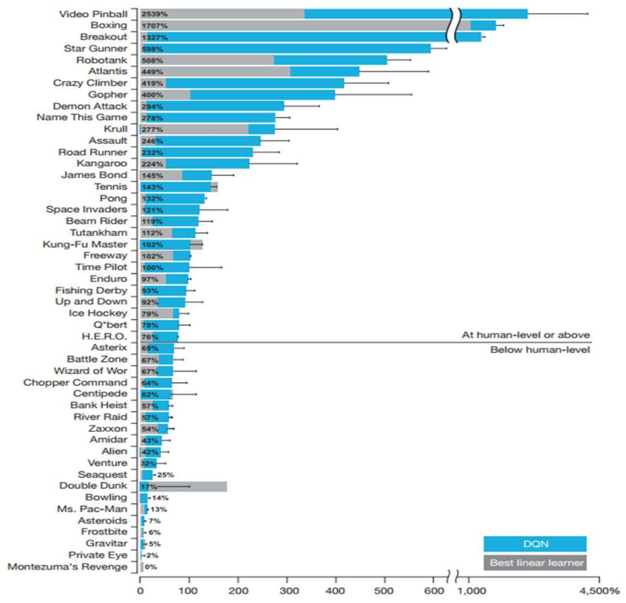


Fig. 4.4: Comparison of the DQN agent with the best reinforcement learning methods in the literature [9]
100% represents a professional human games tester

DQN uses a neural network as a nonlinear function approximation to estimate the state-action value function (Q-function) to handle high-dimensional state spaces more efficiently [9].

4.6.1 Key concepts

DQN RL algorithm is a model-free, value-based, off-policy reinforcement learning method that uses a deep convolutional neural network to approximate the optimal action-value function.

- Q Update Rule for Policy network using the Bellman equation

  $$Q(s, a, \theta) = R(a) + \gamma \underset{a \in A}{argmax} \hat{Q}(s', a', \hat{\theta}) \qquad (4.6.1)$$

  where Q is the policy Q Network (or alternatively called online network), R is the current reward, γ is the discount factor, s' is the next state, a' is the next best action in order to maximize the expected accumulated reward.

  $\hat{Q}$ represents the Target Q Network which is functionally approximated by a neural network with weighting factor $\hat{\theta}$.

Stability issue:

In their seminal work, Mnih et al. [9] address the issue of instability in Reinforcement Learning (RL). They highlighted two key causes contributing to this problem.

Firstly, RL often involves dealing with correlated observations in the sequence of data. These correlations can lead to difficulties in learning an optimal policy as they introduce noise and bias into the learning process. Secondly, small updates to the Q-values, which are essential for determining the action-value function, can have a substantial impact on the policy and consequently alter the data distribution.

The authors [9] suggest solutions to address the instability problem in deep RL.

- Experience replay Buffer: All iterations of the agent's experiences in the form of state, action, reward, and next state tuples:

  $$e_t = (s_t, a_t, r_t, s_{t+1}) \qquad (4.6.2)$$

  in dealing with the environment within each episode are stored in a Replay buffer. Later a randomized sample(mini-batch) of this replay buffer is used to de-correlate sequential experiences for the training of the online Q network.

- Online network: Online network is the main neural network in DQN that is intended to approximates the optimal action-value function. It is used to interact with the environment and select actions. Its weights get updated through backpropagation.
- Target network: Target network is a copy of the online Q network and is used to compute the output of the network when we compute the loss value. The target network is updated at a lower rate of (c) from the online Q network, which helps to prevent the Q-values of the main Q network from fluctuating too much around the optimal and hence stabilizes the Q-values.

- Reward clipping: Reward clipping is used to prevent the Q-values from becoming too large or too small and limits the scale of the error derivative. This helps to prevent the agent from overvaluing or undervaluing certain actions.

Scalability aspect:

One of the key presumption of the DQN is that the neural network architecture is general enough to be trained for a variety of applications and this is a huge scalability advantage of DQN solutions.

The authors [9] used a single neural network architecture with the given hyper-parameters as the deep Q-network, to learn successful policies for various Atari games.

4.6.2 DQN Neural Network

As mentioned before, the optimal action-value function Q^* in reinforcement learning is approximated using a neural network model called online network Q $(s, \mathrm{a}\ \theta)$.

In DQN RL algorithm a copy of the online network, called target network $\hat{Q}(s, \mathrm{a}\ \theta)$, with the identical NN architecture, weighting factors and biases is used to stabilize the learning process. The target network is used to generate the target values for the Q-learning update.

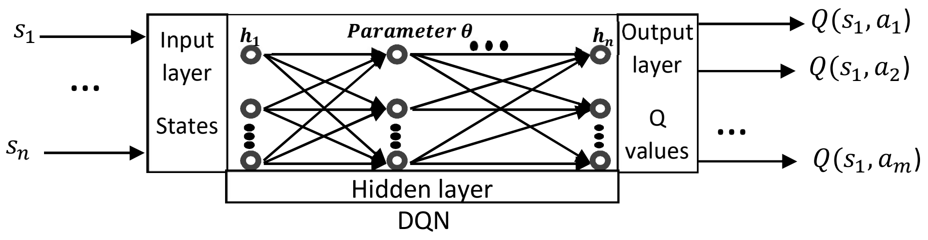


Fig. 4.5: The schematic of a neural network in Deep Q-network reinforcement learning

The inputs to a DQN neural network are the state or observation of the environment. This could be a vector of numerical values representing different aspects of the environment, such as the position of objects, velocities, or any other relevant information. The output of the DQN neural network is the vector of the estimated action-values for each possible action in the given state.

As a result, the input layer of the DQN has as many neurons as we have possible states and the output layer of DQN has as many neurons as we have possible actions.

Architecture of the DQN hidden layer:

The architecture of DQNs consists of multiple hidden layers which can be convolutional or fully connected layers. Using fully connected layers might be advantageous compared to convolutional networks because their dense connectivity allows the DQN network to learn complex relationships between features in the input data. For instance, in radar object detection, fully connected layers can capture intricate dependencies among characteristics such as the location, size, and velocity of the detected object.

The number of hidden layers and the number of neurons in each hidden layer are design hyperparameters and can vary depending on the complexity of the application and the available computation resources for training.

The model DeepMind authors utilized for all experienced Atari games consists of three convolutional layers, succeeded by two fully connected layers [9].

Here are details of their neural network architecture and the hyperparameters they used throughout in all experiments:

RL discount factor $\gamma = 0.99$, Neural network learning rate $\alpha = 0.00025$
Update rate of the target network $C\ every\ 10{,}000\ times$,
The episode length for training = 50 M steps Neural network learning rate $\alpha = 0.00025$
The size of the experience replay memory is $1M\ tuples$.
The mini-batches size = $32\ tuples$. The replay memory gets sampled every $4\ steps$.
ϵ linearly decaying from 1 (100 % random action) to 0.1 (10% random action) over $1M\ steps$.

4.6.3 Training of the DQN

The main goal of DQN training is to enable the agent to update its online neural network and ensuring the convergence and robustness of the learning.

During the training loop, the DQN agent interacts with the environment by playing a high number of episodes. It obtains the feedback from the environment in the form of the next state and the reward received after taking an action in the current state. The agent uses this feedback to learn from its experiences and train its Q network.

The DQN training loop typically follows these steps[6]:

Training Algorithm of the DQN

1. Environment initialization:
   The environment is initialized and the DQN agent is forced to begin with state s0.

2. Action Selection:
   In an epsilon-greedy manner, the agent selects an action. The epsilon-greedy method implies that with the probability of ϵ < 1 the agent selects a random action (exploration); otherwise, it chooses the action with the highest Q-value (A(t) = $argmax\ Q(s,a)$)[7] from the online network (exploitation).

   In epsilon-greedy algorithm the epsilon is decayed to a minimum, causing the agent to take maximum exploration in the beginning and maximum exploitation in the end phase of the training. This ensures that the efficiency of training the Q (s, a) approximation has reached an appropriate maturity level.

3. Environment Interaction:
   The agent takes the chosen action (a) and observes the environment's response in terms of the next state (s') and the reward (r) received from the environment.

[6] Adapted from [2,3,4,5]. The algorithm adaptation presented in this dissertation offers a more accessible and straightforward implementation.
[7] argmax: the action with the highest Q value, given the current state is selected

4. Experience Replay
   The agent stores the environment's response in the experience replay buffer for later use. The experience replay buffer is a data structure that stores a history of transitions. See Formula 4.6.2.
5. Batch Sampling:
   A batch is a small set of transitions that are randomly sampled from the experience replay buffer to train the Q Network. The batch size is a hyperparameter.

6. DQN Forward Pass:
   The forward pass in deep learning is the process of computing the output, which, in DQN reinforcement learning, represents the expected total reward for a given state.

   For each iteration through the batch data, the <u>target Q-value</u> $\hat{Q}(s', a', \hat{\theta})$ is calculated for the next state (s'), best possible action (a') using neural network parameters $\hat{\theta}$.

$$y = R(a) + \gamma \underset{a \in A}{argmax} \hat{Q}(s', a', \hat{\theta}) \qquad (4.6.3)$$

7. Online Q-Value Estimation:
   The optimal Q-Value is estimated from the online network: $Q\ (s, a, \theta)$

8. Loss Computation:
   The temporal difference (or the Bellman error) is defined as the difference between two Q values:

$$\sigma \doteq (Q(s, a, \theta) - y) = Q(s, a, \theta) - \left[R(a) + \gamma \underset{a \in A}{argmax} \hat{Q}(s', a', \hat{\theta})\right] \qquad (4.6.4)$$

   The aim of the DQN is to minimize the temporal difference.
   Google DeepMind authors chose a simple form of Mean Square Error [2] of the temporal difference as the loss function.

$$MSE \doteq \left(Q(s, a, \theta) - \left[R(a) + \gamma \underset{a \in A}{argmax} \hat{Q}(s', a', \theta)\right]\right)^2 \qquad (4.6.5)$$

   In practice, others like Pytorch RL experts [10] alternatively applied the Huber loss instead of MSE as the loss function and obtained good results.

9. DQN Backward Pass: Backpropagation:
   The weighting factors of the online neural network will be updated by the stochastic gradient descent (SGD) of the loss function (5) and the loss is continuously minimized.

   In the original DQN paper [9], the authors used RMSProp optimizer to update the weights of the neural network. They found that RMSProp was more effective than other SGD variants, such as AdaGrad and AdaDelta. However, the Adam optimizer[8] is becoming a more popular choice [36] for training deep neural networks, including DQNs- for instance Google DeepMind used it in their AlphaGo algorithm, due to its better optimization performance and its faster convergence speed [11], [12].

10. Target Network Update
To stabilize the learning process in DQN the Target neural network which is used for DQN forward pass (step 6) is updated at a slower rate directly from the online network. The Target network's parameters are updated with the online neural network's parameters.

Every C episodes reset $\hat{Q}(s', a', \hat{\theta}) = Q(s, a, \theta)$ (4.6.6)
C is a hyperparameter of the DQN.

4.6.4 DQN Convergence criteria

In Deep Q-Network (DQN) and Reinforcement Learning (RL) algorithms, convergence criteria refer to the conditions that determine when the function approximation neural network has successfully learned an optimal or near-optimal policy. After training, the RL agent is expected to interact with the environment at its highest possible level and obtain the maximum possible total reward.

The choice of convergence criteria depends on the specific RL application and the objectives of the training process. According to [1, 10, 16] some of the known and most applied convergence criteria in DQN include:

1. ***Average Reward***: One common convergence criterion is to monitor the average reward achieved by the agent over a specific number of episodes. Once the average reward stabilizes or reaches a satisfactory level, the training process is considered to have converged.

2. ***Maximal Episode Length***: In some cases, RL agents are trained to achieve a goal within a specific number of time steps (episode length). When the agent consistently achieves the goal within the predefined episode length, convergence is achieved.

3. ***Exploration-Estimation Trade-off***: DQN involves a trade-off between exploration and exploitation. Convergence can be linked to finding the right balance between exploration and exploitation to avoid under-exploration or over-exploration.

4. ***Stability of Loss Function***: Monitoring the stability of the loss function during training can be indicative of convergence. A decrease in the loss function indicates the network is learning. Once the loss function reaches a minimum plateau and

[8] Adam stands for Adaptive Moment Estimation and was introduced by Kingma and Ba (2014). The Adam optimizer computes adaptive learning rates for each parameter by tracking the first and second moments of the gradients. The first moment is the average of the gradients, and the second moment is the variance of the gradients.

further iterations or epochs do not lead to significant changes in the loss value then the network is considered to be trained.

Assuring DQN convergence to an optimal policy is not always an easy task. It is highly dependent on the DQN hyperparameters, and any fluctuation of the average total reward or the loss function can be a sign of training instability and divergence. Other challenges during DQN training include slow convergence speed and the possibility of converging to a suboptimal policy.

Zhikang T. Wang and Masahito Ueda (2021) [13] conducted an in-depth investigation into the inefficiency of minimizing the mean squared Bellman error (MSBE) within the deep Q network (DQN) algorithm. They argued that the DQN Q-learning approach does not ensure convergence, and, therefore, its success is largely based on empirical observations. Moreover, they highlighted that achieving optimal performance in DQN heavily depends on meticulous hyperparameter tuning and technical intricacies.

To overcome these problems, they proposed a minimally modified version of the DQN algorithm called Convergent and Efficient Deep Q Network (C-DQN) that is guaranteed to converge and can work with large discount factors (~ 0:9998). They found out that upon the update of target network (step 10 in prev. section) the loss value significantly changes and makes the training instable. Their algorithm constructs a loss function that does not increase upon the update of the target network, and therefore, the algorithm converges in the sense that the loss monotonically decreases.

4.6.5 Optimizing Safety in Autonomous Driving: Challenges and Potentials of DQN RL

While Deep Q-Networks (DQN) have demonstrated remarkable success across diverse domains, including classic board games, complex video games, robotics, and finance, their application in autonomous driving introduces unique challenges and promising potentials.

Computational Demands:

The training of DQN algorithms is inherently computationally demanding. Notably, it often requires access to high-performance hardware such as Graphics Processing Units (GPUs) and large amounts of memory to accelerate the learning process. Even with hardware accelerators and large amounts of memory, the training of a DQN agent can still be time-consuming, often taking days or even weeks to converge. However, advancements in hardware technologies and ongoing algorithmic optimizations hold promise for enhancing efficiency and reducing training times.

Potential in Autonomous Driving:

1. In the context of autonomous driving, one of the notable strengths of DQN algorithm and its extensions lies in their ability of neural networks to model intricate

policy networks. This capability is crucial for ensuring that the DQN RL agent meet the safety requirements of autonomous driving systems.

2. DQN offers a wide range of design and implementation variations, serving as a versatile toolkit for fine-tuning in the context of autonomous driving. This adaptability empowers researchers and practitioners to customize DQN algorithms to address specific challenges and requirements presented by autonomous driving scenarios. Chapter 5 of this thesis extensively explores these variations and evaluates their impact on agent performance and training stability.

4.6.6 DQN Extensions

Deep Q-Network (DQN) was a major advancement in reinforcement learning (RL). It was the first RL algorithm to achieve superhuman performance in Atari games. Since then it has successfully paved its way into other complex domains such as finance, and robotics. However, while DQN showcased its vast potential, reinforcement learning researches unveiled certain limitations and imperfections in the algorithm.

The main issues such as overestimation bias of Q values, slow convergence, instability in learning, and sensitivity to hyperparameters, etc. were identified as obstacles that could hinder its effectiveness in real-world applications.

These limitations led to the development of several refinements and extensions to improve DQN, seeking to enhance its performance, speed, stability and overcome the observed issues.

In this section, an exploration of the motivations driving the three most notable extensions to the DQN algorithm will be conducted, with an investigation into their enhancements and observable improvements. These DQN extensions of the DQN framework include:

- Double DQN – DDQN
- Prioritized experience replay
- Rainbow DQN

4.6.6.1 Double DQN (van Hasselt et al, Google DeepMind 2016)

Paper: "Deep Reinforcement Learning with Double Q-learning"

Deep Q-Networks (DQN) face challenges related to the overestimation bias inherent in Q-learning due to bootstrapping.

The widely used Q-learning algorithm tends to overestimate action values in specific conditions, attributed to the maximization step over estimated action values. If these overestimations are unevenly distributed and not concentrated on states of interest, they can adversely impact the resulting policy's quality.

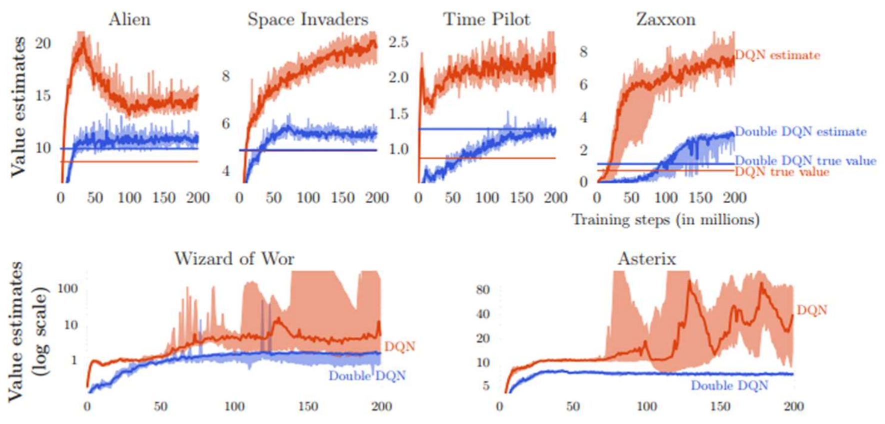


Fig. 4.6: value estimates by DQN (orange) and Double DQN (blue) on six Atari games [14]

The Double Q-learning algorithm, introduced by van Hasselt in 2010 and initially proposed in a tabular setting, has been successfully extended to accommodate arbitrary function approximation, including the use of deep neural networks [14]. The fundamental concept behind Double Q-learning involves mitigating overestimations by breaking down the max operation in the target into action selection and action evaluation.

To tackle the overestimation bias issue in Deep Q-Networks (DQNs), a second network is employed to select the action during the Q-value estimation step. Double DQNs have demonstrated enhanced stability and performance in addressing this challenge.

4.6.6.2 Prioritized Replay Buffer

To enhance learning efficiency, Tom Schaul et al. (2016) proposed a framework for prioritizing experiences, allowing the more frequent replay of crucial transitions [15]. The intuition behind the Prioritized Experience Replay (PER) mechanism is to assign priorities to experiences based on their learning potential, determined by the magnitude of the temporal difference (TD) error. Experiences with higher TD errors indicate a larger discrepancy between predicted and actual outcomes, suggesting a more significant contribution to the learning process.

The probability of sampling each transition is concretely defined using a mathematical distribution formula provided by Schaul et al.

$$P(i) = \frac{{P_i}^{\alpha}}{\sum_k {P_k}^{\alpha}} \tag{4.6.7}$$

Where P_i> 0 is the priority of transition i. The exponent α determines how much prioritization is used, with α = 0 corresponding to the uniform case.

This formula, which governs the sampling probabilities, ensures a targeted focus on transitions with higher priorities.

By prioritizing the replay of high-priority experiences, PER aims to focus the learning algorithm on the most informative and challenging samples, potentially accelerating convergence and improving the overall performance of the reinforcement learning system. This approach addresses issues related to the uniform random sampling used in traditional experience replay.

The selective prioritization of experiences in Prioritized Experience Replay, when compared to the baseline DQN, results in a more effective and targeted learning mechanism. Prioritized Experience Replay DQN outperformed the baseline DQN in 41 out of 49 Atari games, reaching a new state-of-the-art level.

4.6.6.3 Rainbow DQN

Hessel et al. (2017) introduced a seminal contribution to the field of deep reinforcement learning with their paper titled "Rainbow: Combining Improvements in Deep Reinforcement Learning" ([16]).

This work presented a suite of six pivotal already existing extensions to the Deep Q Network (DQN), including double Q-learning, prioritized experience replay, duelling networks, distributional, and Noisy DQNs.

Through empirical experiments, the authors found that the combination of all extensions yields state-of-the-art performance on the Atari 2600 benchmark, both in terms of data efficiency and final performance.

In the course of their empirical investigations, the researchers faced the challenge of optimizing a substantial number of hyperparameters and fine-tuning each component. However, the study systematically explored various configurations of the Rainbow architecture, selectively excluding specific enhancements, such as the duelling network, to determine the contribution of each component to the overall performance.

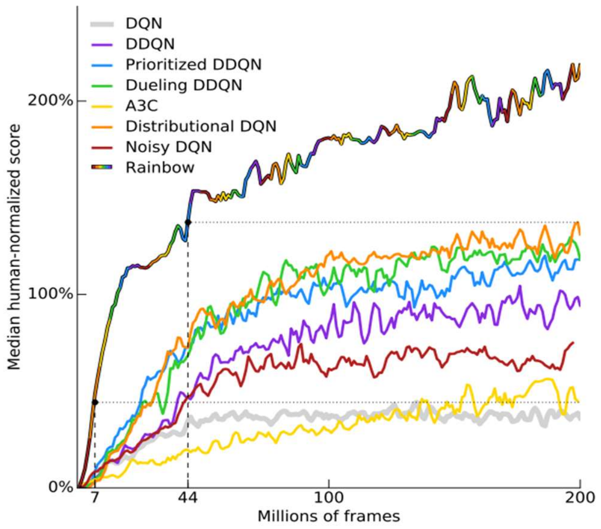


Fig. 4.7: Median human-normalized performance across 57 Atari games [16]

4.6.6.4 A summary of other DQN extensions:

Duelling network architecture

The duelling network architecture extends DQN by enhancing convergence speed through a network architecture that closely represents the problem being addressed.

Separation of the Q-function into two streams:

one stream is for estimating the value function and the other is for estimating the advantage function. This allows the network to learn the value of being in a certain state regardless of the action and the advantage of taking different actions [17].

Noisy DQN

The NoisyNet DQN approach introduces parametric noise to the weights of a deep reinforcement learning agent. This noise induces stochasticity in the agent's policy, encouraging exploration in the action space without relying solely on epsilon-greedy exploration.

The parameters of the noise are learned through gradient descent, alongside the remaining network weights. NoisyNet is easy to implement and incurs minimal computational overhead.

By replacing traditional exploration heuristics in A3C, DQN, and Duelling agents with NoisyNet, the study demonstrates significantly improved performance, achieving higher scores in various Atari games [18].

Distributional DQN: The distributional DQN introduces a novel algorithm that applies Bellman's equation to learn approximate value distributions. Instead of predicting the Q-values directly, it estimates the probability distribution over possible returns for each state-action pair. In an empirical evaluation carried out on the Arcade Learning Environment's comprehensive game suite, the algorithm demonstrates state-of-the-art results compared to the DQN baseline and DQN extensions [19].

4.7 Policy Gradient Reinforcement Learning Methods

Policy gradient methods are a subset of reinforcement learning techniques falling under the broader category of Policy-based RL. Unlike value-based methods, e.g. DQN RL, that emphasize estimating the value of states V(s) or state-action value function Q (s, a), policy gradient methods work by directly improving the agent's policy to maximize cumulative rewards over time.

A policy in a stochastic environment is a probability distribution, whereas in a deterministic environment, it is a mapping that the agent uses to select actions.

The policy π is parameterized by a vector of weights θ, and the goal is to find the optimal weights that maximize the expected discounted sum of rewards over an episode, which is denoted as expected return G_t. [1]

$$G_t = \sum_{k=0}^{H} \gamma^k R_{t+k+1} \tag{4.7.1}$$

Where H is the length of the episode, γ is the discount factor, and $R_t$is the reward at time step t.

The idea of optimizing policies by gradient methods dates back to Bellman (1957), but the first policy gradient algorithm for reinforcement learning was proposed by Williams (1992). [1]
The gradient algorithms he originally developed were for discrete action spaces, but in later variants have since been extended to continuous action spaces too.

Policy gradient methods have several advantages over value-based methods. They can incorporate prior knowledge into the policy class, operate effectively in stochastic environments, manage complex high-dimensional and continuous state/action spaces, and ensure convergence to at least a local optimum [1].

However, they also have some drawbacks, such as being slow, sample-inefficient, suffering from high variance in gradient estimates, and sensitive to hyperparameters, converging to local optima only [1]. Furthermore, they might encounter difficulties in off-policy settings, when the data distribution does not align with the current policy.

Due to their ability to optimize policies directly for complex tasks and handling continuous state/action spaces, Policy gradient methods have been successfully applied to various domains, such as robotics, game playing, natural language processing, and computer vision, etc. [1], [21], [22], [23], [24]

Some of the major variants of policy gradient reinforcement learning methods that have been developed to address different challenges and optimize learning efficiency include:

- Vanilla Policy Gradient (REINFORCE): The classic policy gradient method that updates policies based on the gradient of expected rewards, often utilizing Monte Carlo sampling.

- Trust Region Policy Optimization (TRPO): Enforces a constraint on policy updates to ensure stable learning while aiming to improve policy performance.
- Proximal Policy Optimization (PPO): Balances stability and sample efficiency by using a clipped surrogate objective to prevent drastic policy updates.
- Actor-Critic Methods: Combine policy-based and value-based approaches by having an actor (policy) and a critic (value function) work together to improve learning stability.
- Deterministic Policy Optimization (DPO): Focuses on learning deterministic policies for cases where deterministic actions are preferable or noise is undesirable.

4.7.1 Key concepts

Depicting the core principles of policy gradient reinforcement learning methods, the key concepts can be summarised as illustrated in Figure 4.8.

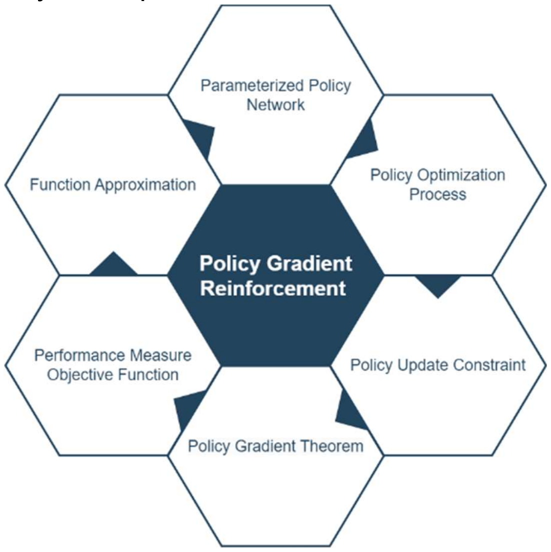

Parameterized Policy Network
Function Approximation
Policy Optimization Process
Policy Gradient Reinforcement
Performance Measure Objective Function
Policy Update Constraint
Policy Gradient Theorem

Fig. 4.8: Key Aspects of Policy Gradient Reinforcement Learning methods

4.7.1.1 Parameterized Policy Network

In Policy Gradient Reinforcement Learning, a parameterized policy network defines a stochastic or deterministic policy, mapping states or observations from the environment to actions that an agent can take. The term "parameterized" signifies that the policy network is characterized by a set of learnable parameters θ, typically represented by weights in a neural network architecture. During training, these parameters are optimized through techniques such as gradient ascent to satisfy an objective function that maximizes the expected cumulative rewards.

4.7.1.2 Function approximation

The goal of reinforcement learning is to find an optimal policy $\pi^*(s, a)$ that guides the agent to consistently select the best possible action and maximize its total reward.

In Policy Gradient methods, this policy is approximated by a parameterized function of actions and states, denoted as $\pi_\theta^*(s, a; \theta)$, where $\theta \in R^d$ represents the learnable parameters that are updated through gradient-based optimization techniques.

This function approximator can be mathematically formulated as: [1]

$$\pi_\theta^*(s,a;\theta) \approx \pi_\theta(s,a;\theta) = \pi_\theta(a|s;\theta) = \Pr\{A_t = a|\, S_t = s, \theta_t = \theta\} \qquad (4.7.2)$$

The probability that agent takes action: a at time t given that the environment is at state : s at time t with the parameter set θ.

The function approximator receives environmental observations (states) as input and generates actions chosen based on currently approximated policy with respect to θ.

The optimization of policy parameters aims to maximize the expected cumulative rewards. This optimization process is driven by the Policy Gradient Theorem, which provides a mathematical link between a policy performance metric and the gradient of the expected reward with respect to policy parameters θ.

Sutton and Barto [1] argue in chapter "13.1 Policy Approximation and its Advantages" that when the action space is discrete and not very large, an exponential softmax distribution can be seen as a suitable natural approximation.

$$\pi_\theta(a|s;\theta) \doteq \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}} \qquad (4.7.3)$$

The rationale behind this is that the softmax function assigns a higher probability of selection to an action with the highest preferences (function h). For continuous action spaces, a Gaussian distribution can perform a similar role.

Alternatively, for environments with larger state/action spaces and more complex dynamics the Policy can be approximated by an Artificial Neural Network ANN.

4.7.1.3 Policy Optimization process

Policy gradient methods in reinforcement learning utilize gradient-based optimization. While they often involve derivatives of a performance measure (usually called the "objective" or "loss" function), they don't always require "backpropagation" in the same way as supervised learning does.

As usual in reinforcement learning, these methods face the challenge of the exploration-exploitation dilemma, where the agent needs to balance between selecting an action with the highest expected reward (exploitation) with selecting a random action (exploration) to discover potentially higher rewards.

By iteratively computing gradients of the performance measure (objective function) and updating the policy parameters through gradient ascent[9], Policy gradient methods refine their policy functions and converge towards policies that yield higher rewards. The stability of the optimization process and convergence of the policy parameters need to be taken care of. In the subsequent chapter, these techniques will be covered in more detail.

[9] The goal of the optimization is to increase the performance metric over time. Hence, we employ gradient ascent, differing from gradient descent used to minimize loss functions.

4.7.1.4 Performance Measure $J(\theta)$

The Performance measure is a quantitative metric used to evaluate how well a particular policy is performing in a reinforcement learning task. It's a way of quantifying the quality of the agent's decisions and actions based on the rewards it receives over time.

The performance measure in policy gradient is defined as the expected discounted return of a policy (formula (4.7.1)), serving as the objective function for iterative maximization to enhance this measure.

The Policy Gradient Theorem states that by iteratively adjusting the policy parameters θ along the direction of the gradient of J(θ), the policy can converge towards one that yields the highest cumulative rewards.

Update rule:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)} \tag{4.7.4}$$

4.7.1.5 Policy Update Constraint

Policy Update Constraint is often essential for maintaining stability in learning, preventing excessively large policy updates that could lead to divergence or poor convergence.
Further details regarding the policy update constraint will be explored in the subsequent chapters, particularly when we delve into Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO) reinforcement learning methods in chapters 4.7.4 and 4.7.5.

4.7.1.6 Policy Gradient Theorem

The Policy Gradient Theorem provides the mathematical foundation for computing the gradient of the expected rewards with respect to policy parameters θ in reinforcement learning. It can be formulated as follows: [1]

Let $\pi_\theta(a|s,\theta)$ be a parameterized policy that specifies the probability of taking action "a" in state "s" with parameters θ.

Consider the objective function- Performance measure - J(θ) that represents the expected cumulative reward that an agent can achieve from state s0 onwards under policy π_θ:

$$J(\theta) \doteq V\pi_\theta(s0) \tag{4.7.5}$$

The Policy Gradient Theorem can be mathematically expressed as:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a)\, \nabla \pi_\theta(a|s,\theta) \tag{4.7.6}$$

Where:

∇J(θ) represents the gradient of the expected cumulative reward.

μ(s) is the mean of the policy. It is the probability that the policy will take any action in a given state.

q_π is the state-action value function under the given policy π.

$\nabla\pi_\theta(a|s,\theta)$ is the gradient of the policy's probability of selecting action "a" in state "s" with respect to the policy parameters θ.

The proof can be found in Sutton & Barto [1].

The Policy Gradient Theorem states that the gradient of the expected cumulative reward with respect to the policy parameters (J(θ)) will be proportional to the gradient of the policy corresponding to parameter vector π_θ and we can optimize the policy by adjusting θ in the direction that increases the probability of actions that lead to higher rewards. The derivation of the formula is presented in the subsequent chapter.

The policy gradient theorem is a powerful instrument for all gradient based reinforcement learning methods. It allows us to directly optimize the policy, without having to estimate the value function.

In the subsequent chapters, a few of the most important Policy Gradient algorithms, including REINFORCE and Proximal Policy Optimization (PPO), will be covered, utilizing the policy gradient theorem.

4.7.2 Vanilla Policy Gradient – REINFORCE

"REINFORCE" is an acronym that stands for "REward Increment = Nonnegative Factor × Offset Reinforcement × Characteristic Eligibility." It is a policy gradient algorithm commonly used in reinforcement learning and primarily introduced by Williams [20] in 1992. The algorithm was later extended and improved by various researchers, such as Sutton et al. (1999) and Peters and Schaal (2008).

REINFOECE, in its simplest form, is also called Vanilla REINFORCE.

The term "vanilla" is often used in the context of software development to refer to a basic or standard version of a technology or framework. In the case of reinforcement learning, "vanilla" refers to the simplest and most straightforward form of an algorithm, without any additional modifications or enhancements.

Before delving into the details of the REINFORCE algorithm, let's revisit the Policy Gradient Theorem (PGT) because it provides the necessary groundwork for understanding the formulation of the REINFORCE algorithm.

We recall from the PGT (4.7.6) that it gives an expression proportional to the gradient:

$$\nabla J(\theta) \;\propto\; \sum_s \mu(s) \sum_a q_\pi(s,a)\, \nabla\pi_\theta(a|s,\theta)$$

In order to change the proportionality to an equation and ease the implementation, Sutton and Barto [2] modified the formula as:

$$\nabla J(\theta) = \mathrm{E}[\textstyle\sum_a q_\pi(s_t,a)\nabla\pi_\theta(a|s_t,\theta)] = \mathrm{E}[\textstyle\sum_a \nabla\pi_\theta(a|s_t,\theta) q_\pi(s_t,a)\,] \qquad (4.7.7)$$

The rationale behind this modification is because the right hand side of PGT is a sum over states weighted by how often the states occur under Policy π.

4.7.2.1 Derivation of the Gradient estimate and the update rule:

$$\nabla J(\theta) = \mathrm{E}[\textstyle\sum_a q_\pi(s_t,a)\nabla\pi_\theta(a|s_t,\theta)] = \mathrm{E}[\textstyle\sum_a \pi_\theta(a|s_t,\theta)\, q_\pi(s_t,a)\, \frac{\nabla\pi_\theta(a|s_t,\theta)}{\pi_\theta(a|s_t,\theta)}]$$

$$= \mathrm{E}\left[\sum_a q_\pi(s_t,A_t)\frac{\nabla\pi_\theta(a|s_t,\theta)}{\pi_\theta(a|s_t,\theta)}\right] = \mathrm{E}[\textstyle\sum_a G_{t_\pi} \frac{\nabla\pi_\theta(a_t|s_t,\theta)}{\pi_\theta(a_t|s_t,\theta)}] \qquad (4.7.8)$$

Where $G_t$is the return.

Hence, the REINFORCE gradient-based update rule for the parameter vector θ is given by:

$$\theta_{t+1} \doteq \theta_t + \alpha \,.\, G_{t_\pi} \frac{\nabla\pi_\theta(a|s_t,\theta)}{\pi_\theta(a_t|s_t,\theta)} \qquad (4.7.9)$$

In practice, there exist slightly modified versions of this formula that simplify gradient computations at each step.

As we know from mathematics the logarithmic gradient is defined as: $\frac{\nabla x}{x} = \nabla \ln x$

Therefore, we can express the update rule using the log probability of the policy gradient. $\nabla J(\theta) = \mathrm{E}[\sum_a G_{t_\pi} \nabla\log\pi_\theta(a|s,\theta)]$ (4.7.10)

By taking the logarithm of the policy gradient, the optimization problem becomes more manageable, and the gradient computation often gains stability.

$$\theta_{t+1} = \theta_t + \alpha . \nabla_\theta Log\ \pi_\theta(a|s,\theta) . G_t \tag{4.7.11}$$

4.7.2.2 The REINFORCE Policy Gradient Algorithm

The REINFORCE Policy Gradient Algorithm has the following steps:

Training Algorithm of the REINFORCE Policy Gradient[10]

<u>Initialization:</u>

1. Initialize the policy parameters θ with random weights
2. Initialize the total cumulated reward to 0.
3. <u>Roll out N full episodes - Sampling and collection of transition Data</u>
4. For each episode:
5. Reset the environment to an initial state s0
6. Follow the stochastic policy π and select an action with the highest probability.
7. Take the action and observe the environment response.
8. Collect transitions samples (s, a, s', r) and store them in memory.
9. Repeat until episode terminates
10. <u>Training of the Policy Network Using Collected Transition Data based on the Monte Carlo Method</u>
11. Update the cumulative return: $G_t += r_t$
12. Compute the gradient of performance measure (objective function) with respect to the policy parameters for this sample
13. Perform stochastic gradient ascent on the objective function using the computed gradients
14. Update the policy parameters θ in the direction that maximizes the expected cumulative reward.
15. Repeat with the next episode until it converges

[10] Adapted from [20]. The algorithm adaptation presented in this dissertation offers a more accessible and straightforward implementation.

The REINFORCE Policy Gradient is a Monte Carlo sampling approach algorithm [1] that operates without a bootstrapping technique.

This means that you collect multiple trajectories by interacting with the environment and compute sample averages to estimate the expected gradient. In this case, the “Expected Return” is approximated using sample averages

4.7.2.3 A brief comparison between REINFORCE Policy Gradient and the DQN:

In his book "Deep Reinforcement Learning Hands-On - 2020 – chapter 11" [2], Lapan, M. brings attention to key differences between policy gradient reinforcement learning methods and the value-based Reinforcement learning methods such as DQN.

Unlike DQN, policy gradient methods do not explicitly require exploration through random actions for environment understanding. Additionally, in their training process, policy gradient methods do not need to have a replay buffer or a target network.

He then, implemented a REINFORCE and a DQN version of Reinforcement learning and applied both solution to the identical environment (CartPole within the Gym framework) and compared their convergence behaviour.

According to his benchmarking and by analysing their convergence patterns, he concluded that REINFORCE converges faster and requires fewer training steps and episodes to successfully solve the CartPole environment.

4.7.3 REINFORCE algorithm with Baseline

The conventional REINFORCE algorithm encounters challenges due to elevated variances during training, resulting in instability and slow convergence behaviour. If the learning process is unstable, the policy may not converge to an optimal solution.

As observed in equation (4.7.9) the gradient-based update rule of the REINFORCE algorithm can be expressed as follows:

$$\theta_{t+1} \doteq \theta_t \ + \alpha \,.\, G_t \frac{\nabla \pi_\theta(a_t|s_t,\theta)}{\pi_\theta(a_t|s_t,\theta)} \qquad (4.7.9)$$

However, the gradient estimate that relies on the cumulative reward, G_t, frequently experiences significant variances in the training loop leading to substantial instability in the training process.

To mitigate the high variance associated with the policy gradient estimates and to improve the stability of learning, Williams [20] introduced the REINFORCE Policy Gradient with baseline.

4.7.3.1 Update Rule in REINFORCE algorithm with Baseline

In the Baseline Policy Gradient algorithm, the cumulated reward G_t is subtracted by a baseline value function b(s). This simple technique serves to reduce the variance of the gradient estimates and improves the stability of the learning process.

$$\nabla \mathrm{J}(\theta) = \mathrm{E}[\textstyle\sum_a \frac{\nabla \pi_\theta(a_t|s_t,\theta)}{\pi_\theta(a_t|s_t,\theta)} \,.\, (G_{t_\pi} - b(s_t))] \qquad (4.7.12)$$

If we rewrite 4.7.12 using logarithmic gradient $\frac{\nabla x}{x} = \nabla \ln x$ then

$$\nabla J(\theta) = \mathrm{E}[\sum_a \nabla_\theta Log\ \pi_\theta(a_t|s_t,\theta)(G_{t_\pi} - b(s_t))] \tag{4.7.13}$$

Hence, the update rule is stated as:

$$\theta_{t+1} = \theta_t + \alpha . \nabla_\theta Log\ \pi_\theta(a_t|s_t)(G_t - b(s_t)) \tag{4.7.14}$$

By subtracting the baseline, the variance of the gradient estimates is reduced, making the learning process more stable and enabling the algorithm to learn more efficiently.

The main idea behind using a baseline function, b(s), is to estimate the advantage of each action, which is the difference between the observed cumulated reward G_t and an expected baseline value b(s).

The baseline function, b(s), can be any function, even a random variable as long as it does not have any dependency on action a. [1] If b(s) = 0 then we have vanilla REINFORCE algorithm again.

By combining the REINFORCE algorithm with a baseline the Policy Gradient Method is able to minimizes the variance of the individual weight changes over time [20] and stabilizes the learning process. The rest of the algorithm will remain the same as for vanilla REINFORCE algorithm.

Sutton and Barto [1] demonstrated the advantage of the REINFORCE with baseline variant over the conventional REINFORCE method.

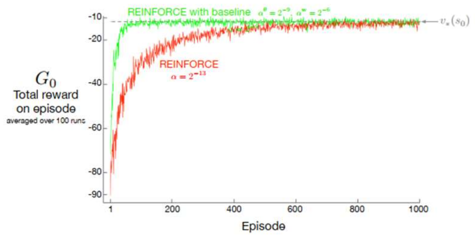


Fig. 4.9: Adding a baseline to REINFORCE can make it learn much faster [1]

The total reward G0 exhibits reduced fluctuations and the algorithm learns faster. See Figure 4.9.

4.7.3.2 Known Limitations of Vanilla REINFORCE and REINFORCE with Baseline Methods

Both Vanilla REINFORCE and REINFORCE with Baseline encounter significant challenges. They suffer from high variance in gradient estimates, leading to slow convergence. Additionally, they can struggle with exploration difficulties and show sensitivity to hyperparameters. They are also known to be sample inefficient and prone to converge to local optima.

In the Theory of policy gradient methods, researchers have introduced advanced alternatives to the REINFORCE and the baseline Policy gradient approaches. These advanced and more complex methods that offer improved stability and enhanced performance on a wide variety of applications. Among these alternatives, Trust-region (TRPO) and Proximal Policy Optimization (PPO) stand out as the most signification methods.

4.7.4 Trust Region Policy Optimization (TRPO)

To address several limitations of earlier gradient-based policy optimization methods, such as sample inefficiency, high variance of gradients, and slow convergence, researchers from Berkeley University introduced a novel policy gradient approach known as Trust Region Policy Optimization (TRPO). This advancement was presented in their 2015 paper titled "Trust Region Policy Optimization." [24]

TRPO is claimed to be similar to natural policy gradient methods and is effective for optimizing large nonlinear policies and has a Robust performance on a wide variety of tasks. [24]

Two major underlying factors contributing to these training challenges are:

- The Policy and the objective function are very complex and non-linear.
- High learning rates might lead to high rewards initially, but as the updates become less reliable, they might ultimately result in dis-convergence or sluggish convergence towards suboptimal policies.

The algorithm.

To enhance the efficiency and stability of the optimization process and to avoid dealing with the complexity of policies, TRPO substitutes the original policy gradient objective with a surrogate objective function.

The surrogate objective function is simpler and can be optimized more efficiently and reliably with monotonic improvements. The update of the surrogate objective function is limited within a trusted region.

4.7.4.1 TRPO Update rule

In TRPO, similar to all other gradient-based on-policy reinforcement learning method, the objective is to find the optimal policy that maximizes the expected cumulative reward.

The TRPO algorithm is frequently acknowledged for its mathematical intricacy. Interested readers can refer to the original paper [24] and related textbooks for a detailed derivation of the update rule.

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_{\tilde{\pi}(s)} \sum_a \tilde{\pi}(a|s) A_\pi(s,a) \qquad (4.7.15)$$

Where: η(π) denotes the expected discounted rewards of policy π.

$\tilde{\pi}$ represents the updated version of policy π at each time step. In TRPO $\tilde{\pi}$ must adhere to a trust region constraint. See Trust region bound.

$\rho_{\tilde{\pi}}(s)$is the discounted state visitation frequency and is defined as:

$$\rho_{\tilde{\pi}}(s) \doteq P(s0 = s) + \gamma\, P(s1 = s) + \gamma^2 P(s2 = s) + \cdots \qquad (4.7.16)$$

$A_\pi(s,a)$is the advantage function which is defined as:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \qquad (4.7.17)$$

As a quick reminder, in a deterministic environment, $\tilde{\pi}(a|s)$ is essentially the argmax function that we have encountered frequently in the past. $\pi(a|s) = \underset{a \in A}{argmax} Q(s, a)$; the function selects the action: a which in state: s returns the maximum cumulative reward.

4.7.4.2 Surrogate Objective Function:

Due to the complex dependency of $\rho_{\tilde{\pi}}(s)$ on $_{\tilde{\pi}}$ the update rule of The objective function in equation 4.7.15 is not directly computable [24]. To address this challenge, the creators of TRPO propose specific approximations to η that align with the theoretically justified approach of optimizing policies through gradient-based methods.

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_{\pi(s)} \sum_a \tilde{\pi}(a|s) A_\pi(s, a) \qquad (4.7.18)$$

The key point here is that the approximated objective function depends on visitation frequency ρ_π rather than $\rho_{\tilde{\pi}}$.

The distinctive property of TRPO is that it imposes a constraint on the policy update's proximity to the current policy.

4.7.4.3 Trust Region Bound

Let $\pi_\theta(a|s)$be the agent's parameterized policy to select an action. The trust region can be defined with respect to θ or with respect to π_θ as follows:[11]

1. Trust Region with Respect to θ:

   $$Trust\ Region \doteq \{\theta' | D(\theta, \theta') < \delta\} \qquad (4.7.19)$$

   Where, D (θ, θ') is a distance or dissimilarity measure between the current parameter vector θ and the updated parameter vector θ' and δ is a positive constant which represents the radius or size of the trust region.

   The trust region here is defined directly in terms of the parameters θ. The idea is to restrict how much the parameters can change from one iteration to the next. The optimization algorithm ensures that the update to θ stays within a certain region or neighbourhood.

   When θ represents a vector of weight factors in a neural network or any other complex function approximation, the trust region serves as a constraint to limit how large these weights can be updated during the update process.

2. Trust Region with Respect to π:

   $$Trust\ Region \doteq \{\theta' | D(\pi_\theta, \pi_{\theta'}) < \delta\} \qquad (4.7.20)$$

   In this formulation, the trust region is defined in terms of the policy function π_θ. The policy is parameterized by θ, and the trust region constrains how much the policy can change within each iteration.

[11] Formulas adapted from [24] with simplified notations for improved clarity and ease of implementation in the context of this dissertation.

The objective function is defined in terms of the policy's performance, such as expected rewards in reinforcement learning.

If the trust region is determined with respect to the policy then the limited change in a policy would usually correspond to a change in the probability of choosing some action, and that will lead to a gradual change in the value function V(s). On the contrary, if we select to determine the trust region with respect to the policy parameter θ, then a limited change in the weighting factor of the network might result in undesired large changes in the value function. [24]

4.7.4.4 KL Divergence

Kullback-Leibler (KL) divergence, also known as or relative entropy, is primarily used to measure the difference or distance between two probability distributions or density functions. It quantifies how one probability distribution differs from a reference or target distribution. In reinforcement learning policies are also probability distribution functions that describe the likelihood of choosing a specific action.

In the context of Trust Region Policy Optimization (TRPO), the fundamental principle revolves around constraining the magnitude of policy updates with respect to the previous policy. This essential constraint is achieved through the application of KL divergence. The KL divergence is employed as a distance metric to quantitatively regulate the distance between the old policy and the current policy for each state, ensuring monotonic improvement in the policy updates and facilitating good convergence.

4.7.4.5 The TRPO Surrogate Objective Function with the Region Policy Update Constraint:

In TRPO, the surrogate objective is maximized subject to a constraint on the magnitude of the policy update, as specified below:

$$\underset{\theta}{maximize}\, \hat{E}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}\, \hat{A}_t\right] \tag{4.7.21}$$

Subject to $\hat{E}_t[KL[\pi_{\theta_{old}}(.\,|s_t)\,,\; \pi_\theta(.\,|s_t)]] \leq \delta\,; for\ \forall s_t \in S$ (4.7.22)
where $\widehat{A}_t$ represents the estimate of the advantage function at time step t and $\widehat{E}_t$is the expectation operator over a finite batch of samples.

4.7.4.6 TRPO Algorithm

The Trust Region Policy Optimization Algorithm has the following steps[12]:

```
Training Algorithm of the TRPO
    Initialization:
    3. Initialize the policy parameters θ with random weights
    4. Initialize the total cumulated reward to 0.
    Roll out N full episodes - Sampling and collection of transition Data
    For each episode i:
    5. Reset the environment to an initial state s0
    6. Follow the stochastic policy π and select an action with the highest probability.
    7. Take the action and observe the environment response.
    8. Collect transitions samples (s, a, s', r)
    9. Compute all advantage values $A_{\pi_i}(s,a)$
    Policy Update
    10. Solve the constrained optimization problem
    $\pi_{i+1} = \underset{\pi}{argmax} [L_{\pi_i}(\pi) - CD_{KL}^{max}(\pi_i, \pi)]$
    $where: C = \frac{4\, \gamma_{s,a}^{max} |A_\pi(s,a)|}{(1-\gamma)^2}$
    11. Repeat with the next episode until it converges
```

4.7.4.7 Optimizing Safety in Autonomous Driving: Potentials of TRPO RL

In the context of autonomous vehicles, the trust region constraint within TRPO can be customized to conform to defined safety criteria. This adaptation ensures that policy updates are not only smooth but also exclusively adhere to the safety criteria, emphasizing an approach that prioritizes the overall safety of the autonomous system.

Moreover, TRPO's incorporation of KL divergence, assessing differences between two policies, serves as a potential model for conducting safety arbitration in scenarios involving two redundant agents.

[12] Adapted from [24] and [26]. The algorithm adaptation presented in this dissertation offers a more accessible and straightforward implementation.

4.7.5 Proximal Policy Optimization (PPO)

One of the most promising gradient-based policy optimization reinforcement learning algorithms is called Proximal Policy Optimization (PPO), proposed by Schulman et al at Open AI in 2017. [25]

PPO is considered to be an evolution of the Trust Region Policy Optimization (TRPO) algorithm, and while TRPO applies a trust region constraint to the updated policy, PPO uses a clipped objective function to enforce a more conservative policy update. In PPO, the clipped objective function limits the change in the policy's probability distribution, preventing it from straying too far from the previous policy in each iteration. This controlled policy update strategy makes PPO more stable and easier to implement than TRPO while still achieving competitive performance in reinforcement learning tasks.

PPO often outperforms TRPO despite its relative simplicity of implementation. Therefore, PPO is a popular choice for researchers and practitioners.

4.7.5.1 Stability in Training:

PPO uses a technique called proximal policy optimization to ensure that the updates to the policy are not too large. This helps to prevent the policy from diverging too far from the previous policy and enhances the training stability. [25]

4.7.5.2 Reduction of implementation complexity and hyperparameter sensitivity:

PPO is also relatively easy to implement and tune. One of its goals is to reduce sensitivity to hyperparameters such as learning rates and initialization. This is important because tuning of hyperparameters can be a very challenging task.

4.7.5.3 Sample Efficiency:

PPO addresses the problem of sample efficiency in reinforcement learning. It aims to learn a good policy with fewer samples by utilizing the collected experience more effectively. This is achieved by using each batch of experience for a single gradient update and then discarding it.

4.7.5.4 PPO objective function

The TRPO surrogate objective function is maximized subject to the KL divergence constraints between the current and old policies. This is shown in formulas 4.7.21 and 4.7.22.
The authors of PPO [21] argue that the theory justifying TRPO can also be used to derive a penalty in the objective function instead of a constrained update of the policy. This modification is presented below.

$$\underset{\theta}{maximize}\, \hat{E}_t[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}\, \hat{A}_t - \beta KL[\pi_{\theta_{old}}(.\mid s_t),\ \pi_\theta(.\mid s_t)]] \qquad (4.7.23)$$

Where ß is a hyperparameter that controls the strength of the penalty term.

Their experiments showed that it is not enough to simply choose a fixed penalty coefficient β and optimize the objective function with stochastic gradient descent. To address this challenge and guarantee that the policy always improves, they proposed the clipped surrogate objective function.

4.7.5.5 PPO Algorithm with the Clipped Surrogate Objective:

Schulman et al. at OPEN AI proposed the clipped surrogate objective, which is given in formulas 4.7.24 and 4.7.26.

$$L^{CLIP}(\theta) = \hat{E}_t[\min(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}\, \hat{A}_t\,, clip\,(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}\,, 1- \in, 1+ \in)\, \hat{A}_t] \qquad (4.7.24)$$

At each time-step t, where epsilon is a hyperparameter, say, ε = 0.2 to define the lower bound.

Let r(θ) denote the probability ratio between current and old policies and is defined as:

neural networks for two specific contributions in this disse

$$r(\theta) \doteq \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \qquad (4.7.25)$$

Then the clipped surrogate objective function is formulated as:

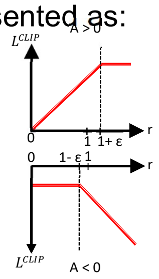
$$L^{CLIP}(\theta) = \hat{E}_t[\min(\,r(\theta)\, \hat{A}_t\,, clip\,(r(\theta)\,, 1- \in, 1+ \in)\, \hat{A}_t] \qquad (4.7.26)$$

It involves two terms. The first term calculates the advantage (A_t) of the new policy compared to the old policy. It measures how much better or worse the new policy is expected to perform compared to the old policy.

The second term limits the policy updates if the policy change exceeds a specified threshold. This clipping ensures that the policy doesn't deviate too far from its current estimate and prevents training instability or divergence.

Mathematically the clipping function of the ratio can be represented as:

$$clip\,(r(\theta)\,, 1- \in, 1+ \in)\, \hat{A}_t = \begin{cases} r(\theta)\ cliped\ at\ 1+ \in, & \hat{A}_t > 0 \\ r(\theta)\ cliped\ at\ 1- \in, & \hat{A}_t < 0 \end{cases}$$



The probability ratio r(θ) is clipped if it is less than (1 - ε) or larger than 1+ ε, else it is left unchanged.

4.7.5.6 PPO Algorithm with the adaptive KL Penalty coefficient

The clipped surrogate objective function enforces a lower bound on the KL divergence between the old and new policies using a clip function. Schulman et al. [7] proposed an alternative method called Adaptive KL Penalty Coefficient, which applies a penalty to the KL divergence directly in the objective function. This method automatically adjusts the penalty coefficient over the course of training to ensure that the KL divergence remains within a desired range.

$$L^{KLPEN} = \hat{E}_t[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t - \beta KL[\pi_{\theta_{old}}(. \mid s_t), \ \pi_\theta(. \mid s_t)]] \qquad (4.7.27)$$

The hyperparameter β, which controls the strength of the penalty, is not fixed. Instead, it is adapted based on the value of the KL divergence, as shown below. This is why the method is called Adaptive KL Penalty.

$$d \doteq \hat{E}_t\left[KL[\pi_{\theta_{old}}(. \mid s_t), \pi_\theta(. \mid s_t)]\right] \qquad (4.7.28)$$

$$If \ d < \frac{d_{targ}}{1.5} \Rightarrow \beta = \frac{\beta}{2}$$

$$If \ d > 1.5 \ d_{targ} \Rightarrow \beta = 2\beta$$

The hyperparameters 1.5 and 2 were chosen heuristically.

4.7.5.7 PPO Algorithms

The Proximal Policy Optimization (PPO) Algorithms have the following steps[13]:

Training Algorithm of the PPO clipped surrogate objective

Initialization:

1. Initialize the policy parameters θ with random weights
2. Initialize the clipping threshold ε.

Roll out N full episodes - Sampling and collection of transition Data

For each episode k:

3. Reset the environment to an initial state s0
4. Follow the current policy $\pi_k(\theta_k)$ and select an action with the highest probability.
5. Take the action and observe the environment response.
6. Collect transitions samples (s, a, s', r) and accumulate it in partial trajectory D_k
7. Repeat steps 4...6 until the episode terminates; T = length of the trajectory in this episode
8. Compute all advantage estimate values $\hat{A}_{\pi_k}(s,a)$

Policy Update

9. Compute the value of the constrained objective function:

$$L_{\theta_k}^{CLIP}(\theta) = \hat{E}_t[\sum_{t=0}^{T}[\min(r_t(\theta)\hat{A}_t, clip\left(r_t(\theta), 1-\in, 1+\in\right)\hat{A}_t]]$$

10. Update policy

$$\theta_{k+1} = \underset{\theta}{argmax}\ L_{\theta_k}^{CLIP}(\theta)$$

By taking K steps of mini-batch of trajectory D_k using Stochastic Gradient Descent (Adam optimizer)

11. $\theta_{old} = \theta$
12. Repeat with the next episode until it converges

Training Algorithm of the PPO adaptive KL Penalty coefficient

1. Initialize the policy parameters θ with random weights
2. Initialize the KL penalty β_0 and target KL divergence threshold δ.

Roll out N full episodes - Sampling and collection of transition Data

For each episode k:

3. Reset the environment to an initial state s0
4. Follow the current policy $\pi_k(\theta_k)$ and select an action with the highest probability.
5. Take the action and observe the environment response.
6. Collect transitions samples (s, a, s', r) and accumulate it in partial trajectory D_k
7. Repeat steps 4...6 until the episode terminates; T = length of the trajectory in this episode
8. Compute all advantage estimate values $\hat{A}_{\pi_k}(s,a)$

Policy Update

9. Compute the value of the constrained objective function:

$$L_{\theta_k}^{CLIP}(\theta) = \hat{E}_t[\sum_{t=0}^{T}[\min(r_t(\theta)\hat{A}_t, clip\left(r_t(\theta), 1-\in, 1+\in\right)\hat{A}_t]]$$

10. Update policy

[13]Adapted from [27]. The algorithm adaptation presented in this dissertation offers a more accessible and straightforward implementation.

$$\theta_{k+1} = \underset{\theta}{argmax}\, L_{\theta_k}(\theta) - \beta_k D_{KL}(\theta, \theta_k)$$

By taking K steps of mini-batch of Stochastic Gradient Descent (Adam optimizer)

11. $if\ D_{KL}(\theta_{k+1}, \theta_k) \geq 1.5\ \delta\ then\ \beta_{k+1} = 2\beta_k$
12. $else\ if\ D_{KL}(\theta_{k+1}, \theta_k) < \delta/1.5\ \ then\ \beta_{k+1} = \frac{\beta_k}{2}$
13. $\theta_{old} = \theta$
14. Repeat with the next episode until it converges

4.7.5.8 Advantages of PPO algorithm in practice

- Stability and Robustness:
    - PPO's clipped objective function prevents large policy updates that can cause training instability and divergence.
- Sample Efficiency:
    - PPO is relatively sample-efficient compared to some other policy gradient methods. It makes efficient use of collected experiences by reusing them multiple times through several epochs of training.
- Compatibility with Continuous Action Spaces:
    - PPO naturally handles continuous action spaces by parameterizing the policy as a probability distribution over actions.
- Code Efficiency:
    - PPO is designed with simplicity and ease of implementation in mind, and it often requires relatively few lines of code changes when compared to a vanilla policy gradient method like REINFORC. [25]

4.7.5.9 Optimizing Safety in Autonomous Driving: Potentials of PPO RL

PPO's inherent characteristics, including simplicity and computational efficiency, make it a suitable candidate for enhancing safety measures in autonomous driving scenarios.

In the context of autonomous vehicles, the PPO's clipped objective function can be customized to meet predefined safety criteria, preventing abrupt policy updates and thereby ensuring a more stable learning process.

This potential is explored in Chapter 7's safety layer, specifically in the safety-dependent policy optimization. A novel approach is proposed, incorporating a safety penalty in the clipped objective function to further enhance autonomous driving safety. For more details, please refer to section 7.2.4.

4.7.6 Actor-Critic Reinforcement Learning Methods

Policy gradient reinforcement methods primarily centre their agent training on optimizing the parameterized policy, with less emphasis on estimating state values or state-action values. Actor-Critic Reinforcement Learning represents a distinct class of policy-gradient reinforcement learning methods, characterized by the integration of both an actor network responsible for policy optimization and a critic network dedicated to value estimation. Hence, Actor-Critic Reinforcement learning is considered to be both policy-based and value-based learning. See Figure 4.10.

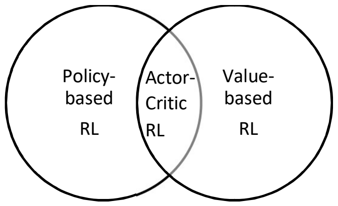


Fig. 4.10: Intersection of Policy-Based and Value-Based RL: Actor-Critic Approach

The idea of combining value-based and policy-based methods, which is fundamental to actor-critic architectures, has been explored by multiple researchers over several decades. Richard S. Sutton, a prominent figure in the field of reinforcement learning, made significant contributions to the development of actor-critic methods, including his work on temporal difference (TD) learning, which forms the foundation for the critic network Chapter 13.5 [1].

Actor-critic methods are known for their sample efficiency and they have been applied to a wide range of applications, including robotics, game playing, autonomous vehicles, and more, due to their ability to handle both continuous action spaces and large state spaces effectively. The Actor-Critic architecture comprises two parameterized networks, often represented as neural networks with parameters θ (for the actor) and φ (for the critic). The actor network is responsible for learning the policy (policy-based), while the critic network is responsible for learning state-values (value-based) based on the received rewards from the environment.

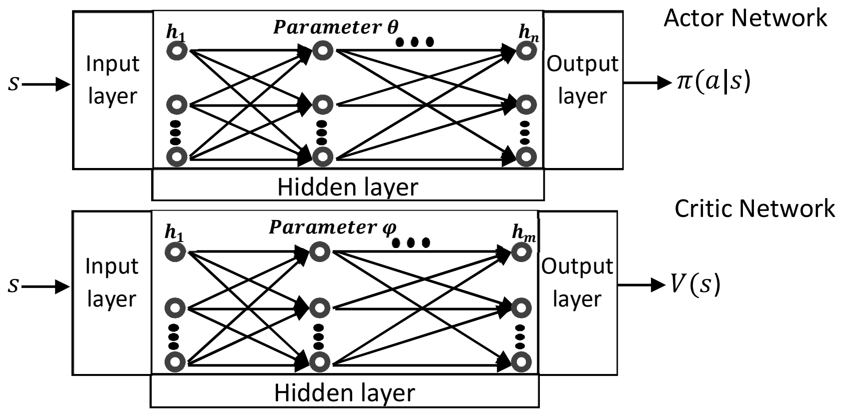


Fig. 4.11: The Actor (policy) and Critic (value function) networks.

4.7.6.1 The Architecture of Actor-Critic Method:

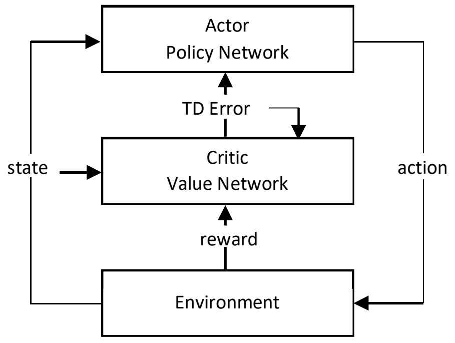


Fig. 4.12: The architecture of Actor-Critic Reinforcement Learning

The actor-critic architecture and the concept of actor-critic reinforcement learning is illustrated in Figure 4.12.

The Actor and Critic networks work in tandem.

- The Actor generates actions probabilistically (in the case of stochastic policies) or deterministically (in the case of deterministic policies) based on the current policy.
- The agent receives rewards and transitions to new states from the environment.
- The critic network evaluates the reward and estimates the expected returns and calculates the TD error.
- The TD error is then used as a feedback signal by the actor to calculate the gradient estimate and update the actor's policy. If an action resulted in higher-than-expected returns (positive TD error), the actor strengthens its policy to favour similar actions in the same states. Conversely, if the returns were lower than expected (negative TD error), the actor adjusts its policy to avoid such actions in those states.
- This feedback loop continues iteratively, with the actor and critic networks gradually updating their networks through reinforcement learning.

In Actor-Critic RL policy and the value function updates are de-coupled. The critic network estimates state values or action values, allowing for online updates during the trajectory.

This online update capability (on-policy) means that the agent doesn't have to wait until the end of the trajectory to update its policy. It can adjust its policy continuously as it interacts with the environment.

This is in contrast to some off-policy Policy Gradient algorithms (e.g., REINFORCE) that collect a batch of experiences before updating the policy. [1]

Actor-Critic's online updates can lead to quicker policy improvements and more sample-efficient learning.

4.7.6.2 Actor learning

The Actor objective function:

In Actor-Critic reinforcement learning, the actor is a Policy Gradient REINFORCE with baseline.

Let’s recap from section 4.7.2 three major steps of Policy gradient REINFORCE with baseline learning:

1. Roll out N full episodes – run the current policy $\pi_\theta(a_t|s_t)$; sample and collect transition Data
2. Compute the gradient estimate as:
   $\nabla J(\theta) = \mathrm{E}[\sum_a \nabla_\theta Log\ \pi_\theta(a_t|s_t,\theta)(G_t - b(s_t))]$ (from 4.7.13)
3. Run the policy update rule:
   $\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}$ (from 4.7.4)

The baseline function $b(s_t)$ is as often set to the state value function of the current state which is estimated by the critic network $V(s_t;\phi)$

Now if we define the advantage function as the return term minus the baseline term in 4.7.13 then:

$$A(a_t,s_t) = \left(G_t - b(s_t)\right) = (G_t - V(s_t;\phi)) \tag{4.7.29}$$

The Gradient of the objective function J(θ) with respect to the actor's parameters θ can be formulated as:

$$\nabla J(\theta) = \mathrm{E}[\textstyle\sum_a \nabla_\theta Log\ \pi_\theta(a_t|s_t,\theta)\ \mathrm{A}(\mathrm{a}_t.s_t)] \tag{4.7.30}$$

The expression ∇J(θ) is used in policy gradient methods to update the actor's parameters θ based on 4.7.4 formula: $\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}$

4.7.6.3 Critic learning

The Critic Loss function:

In Actor-Critic reinforcement learning, the critic is a value-based temporal difference (TD) Q - learning algorithm that estimates the expected return or value function for different states or state-action pairs in order to guide the actor's policy improvement.

Temporal Difference (TD) Error: The TD error measures the difference between the values of successive states. The Temporal difference (TD) error at time step t is defined as:

$$TD(0) = \delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{4.7.31}$$

R_{t+1} is the reward received at time t+1, γ is the discount factor, $V(s_t)$ is the estimated value of the current state, and $V(s_{t+1})$ is the estimated value of the next state s_{t+1}.

To create a loss function for the critic, the mean squared TD(0) error is used, which penalizes the difference between the estimated state value ($V(s_t)$) and the target value $R_{t+1} + \gamma V(s_{t+1})$.

$$\lambda(\phi) = d\phi = \textstyle\sum_{t=1}^{N} \nabla_\phi \left(R_{t+1} + \gamma V(s_{t+1};\phi) - V(s_t;\phi)\right)^2 \tag{4.7.32}$$

The loss function is minimized by adjusting the critic network's weights to improve the value prediction of the critic network. This is typically done using gradient descent.

The gradient of the critic loss with respect to the critic network parameters is used to update the network. Hence, the update rule for the critic network weighting factors is:

$$\phi_{t+1} = \phi_t + \beta d\phi \qquad (4.7.33)$$

4.7.6.4 Actor-Critic Training Algorithm

Training Algorithm of the Actor-Critic Reinforcement Learning. Adapted from [29] [14].

```
Initialization:
1. Initialize the policy parameters θ of the Actor π (a|s; θ) with random weights
2. Initialize the value parameters ϕ of the Critic V(S; ϕ) with random weights
Roll out N full episodes - Sampling and collection of transition Data
For each episode step:
3. Reset the environment to an initial state s0
4. Follow the Actor network stochastic policy $\pi_\theta$ and select an action with the highest probability.
5. Take the action and observe the environment response (s, a, s', r).

Training of the Policy and Value Networks
6. Compute the actor loss based on the gradient of the expected return with respect to the actor's policy.
7. Update the cumulative return: $G_t = R_{t+1} + \gamma V(s_{t+1}; \phi)$
8. Compute the advantage function $A(a_t, s_t) = (G_t - V(s_t; \phi))$ based on the critic estimated value function
Update the Actor Network (Policy) and the Critic Network (Value Function)
9. Compute the policy gradient of the actor network (Actor objective function) with respect to the policy parameters for this sample $d\theta = \sum_{t=1}^{N} \nabla_\theta Log\ \pi_\theta(a_t|s_t, \theta)\ \mathrm{A}(\mathrm{a}_t . s_t)$
10. Compute the gradient of the critic network (Critic objective function) with respect to the value parameters for this sample $d\phi = \sum_{t=1}^{N} \nabla_\phi \left(R_{t+1} + \gamma V(s_{t+1}; \phi) - V(s_t; \phi)\right)^2$
11. Update the actor parameters by applying the gradients $\theta_{t+1} = \theta_t + \alpha d\theta$
12. Update the critic parameters by applying the gradients $\phi_{t+1} = \phi_t + \beta d\phi$

13. Perform stochastic gradient ascent on the objective function using the computed gradients
14. Update the policy parameters θ in the direction that maximizes the expected cumulative reward.
15. Repeat with the next episode until it converges
16. Compute the gradients for the critic network by minimizing the mean squared error loss between the estimated value function $V(s_t; \phi)$ and the cumulative return: $G_t$across all N experiences.
```

[14] The algorithm adaptation presented in this dissertation offers a more accessible and straightforward implementation.

4.7.6.5 Bootstrapping in Actor-Critic vs. Monte Carlo Approach in REINFORCE Method

A fundamental distinction exists in the underlying methodology between the REINFORCE algorithm and the Actor-Critic (AC) reinforcement learning framework.

The REINFORCE algorithm employs a Monte Carlo approach to estimate the policy gradient. It gathers complete trajectories and adjusts the policy after an episode concludes. This adjustment is based on the average cumulative returns observed throughout those trajectories.

In contrast, the actor within actor-critic methods adopts a bootstrapping approach. This approach empowers the actor to update its policy at every step of the episode, facilitating a more continuous and dynamic learning process. [28]

4.7.6.6 Actor-Critic Extensions

Advantage Actor-Critic (A2C) and Asynchronous Advantage Actor-Critic (A3C) represent notable extensions of the Actor-Critic framework, providing significant advantages in terms of training efficiency, optimal sample utilization, and enhanced scalability.

4.7.6.6.1 Advantage Actor-Critic (A2C)

A2C extends the AC framework by introducing the concept of the Advantage function, which quantifies how much better or worse an action is compared to the average action in a given state. The critic computes the advantage value A(a,s) = Q(a,s) – v(s) and not the state value V(s)

Parallelization: A2C can be efficiently parallelized, allowing multiple agents to interact with the environment and collect experience simultaneously. This parallelization accelerates training and improves sample efficiency.

Synchronous Updates: In A2C, the updates to the actor's policy and the critic's value function are done synchronously, meaning that they occur at the same time step. This synchronous training can lead to more stable learning.

4.7.6.6.2 Asynchronous Advantage Actor-Critic (A3C)

Mnih et al. at DeepMind (2016) introduced Asynchronous Advantage Actor-Critic (A3C) algorithm in paper “Asynchronous Methods for Deep Reinforcement Learning” [31]. A3C maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V\ (s_t;\ \theta_v)$. Both functions are updated after every t_{max} actions. It is a popular variant of actor-critic RL that uses multiple actor-learner threads to parallelize training, enabling faster convergence and better performance.

Asynchronous Updates:

A3C extends the parallelization concept of Advantage Actor-Critic (A2C) by introducing asynchronous updates. A3C agents run in parallel, independently collecting experiences and updating their policy and value function asynchronously. Actor threads in A3C follow distinct exploration policies, improving robustness and performance through diversified exploration. The asynchronous updates in A3C reduce sample correlation, leading to faster convergence [31].

Increased Scalability:

A3C with its parallel reinforcement learning paradigm promises high levels of scalability and can efficiently utilize multi-core CPUs and distributed computing environments. It's particularly effective for handling complex environments and large state spaces.

A3C has been successfully applied in a variety of continuous motor control tasks, demonstrating stable training of neural networks through reinforcement learning. This success has been achieved using both value-based and policy-based methods, as well as off-policy and on-policy methods, in both discrete and continuous domains [31].

4.7.6.7 Optimizing Safety in Autonomous Driving: Potentials of Actor-critic Methods

Actor-critic RL methods can handle continuous action spaces, making them applicable to tasks such as steering or acceleration ratio control in autonomous vehicles. The use of A3C parallelism in training can accelerate learning by enabling multiple agents to explore various aspects of the environment concurrently, which is advantageous given the inherent dynamics in different driving scenarios. Furthermore, the Actor-Critic framework shows potential in the passive monitoring of human drivers, with the human acting as the actor. By employing the critic to evaluate human actions, the framework can dynamically update its policy to align with safer driving practices.

4.7.7 Deep Deterministic Policy Gradient (DDPG)

Lillicrap et al. at DeepMind (2016) introduced the Deep Deterministic Policy Gradient (DDPG) algorithm in their influential paper, "Continuous control with Deep Reinforcement Learning." DDPG is a model-free, off-policy, actor-critic algorithm based on the deterministic policy gradient (DPG) designed for tasks with high-dimensional continuous action spaces.

The algorithm combines the actor-critic approach with insights from the success of Deep Q Network (DQN) (Mnih et al., 2015).

In DDPG, the actor network defines the optimal policy by mapping states to continuous actions, while the critic network evaluates state-action pairs by estimating Q-values.

The actor network is parameterized as $\mu(a_t|s_t;\ \theta)$, representing the current policy, and the critic network learns $Q(a_t s_t; \varphi)$ using the Bellman equation from Q-learning.

The update rules for the actor and critic networks involve gradient ascent and minimizing the temporal difference error, respectively. The loss functions and update rules for the actor and critic networks are provided in [32]. Readers interested in more details can refer to the paper for further information.

DDPG employs a straightforward actor-critic architecture with batch normalization (Ioffe & Szegedy, 2015) applied to sampled experiences during training. Both the actor and critic networks have target networks (off-policy), which are softly updated at each time-step, enhancing stability and convergence during training.

In the DDPG algorithm, the target actor network (μ′) is initialized by adding noise to the weights of the online actor network (μ).

$$\mu'(a_t|s_t;\ \theta) \leftarrow \mu(a_t|s_t;\ \theta) + N \qquad (4.7.34)$$

This initialization improves the exploration and helps prevent overestimation during the early stages of training. [32]

The soft update and batch normalization methods are explained and implemented for a DQN RL agent in Chapter 5, Sections 5.3.1.3 and 5.3.1.5, respectively.

4.7.7.1 Optimizing Safety in Autonomous Driving: Potentials of DDPG methods

DDPG is particularly well-suited for tasks with continuous action spaces, rendering it applicable to critical functionalities such as steering and acceleration control in autonomous vehicles (AVs). In the context of AV safety, DDPG is regarded as a valuable choice because it can optimize policies for executing smooth and continuous actions, thereby minimizing the risk associated with abrupt accelerations or decelerations that might compromise safety.

4.7.8 Summary of policy gradient RL algorithms

Algorithm Name	Update Rule	On/Off -Policy	Use Cases	Remarks
REINFORCE	$\theta_{t+1} \doteq \theta_t + \alpha \,.\, G_{t_\pi} \frac{\nabla \pi_\theta(a\|s_t,\theta)}{\pi_\theta(a_t\|s_t,\theta)}$	On-Policy	continuous and discrete	sensitive to hyperparameters and initial conditions easy to implement
TRPO	$\pi_{i+1} = \underset{\pi}{argmax}[L_{\pi_i}(\pi) - CD_{KL}^{max}(\pi_i,\pi)]$	On-policy	continuous and discrete	Stability Complex algorithm
PPO	$L^{CLIP}(\theta) = \hat{E}_t[\min(r(\theta)\,\hat{A}_t, clip\,(r(\theta), 1-\in, 1+\in)\,\hat{A}_t]$	On-policy	continuous and discrete	Surrogate objective function stability Easy to implement
Actor-Critic	$\nabla J(\theta) = \mathrm{E}[\sum_a \nabla_\theta Log\, \pi_\theta(a_t\|s_t,\theta)\,\mathrm{A}(\mathrm{a_t}.\,s_t)]$ $\lambda(\phi) = d\phi = \sum_{t=1}^{N} \nabla_\phi \left(R_{t+1} + \gamma V(s_{t+1};\phi) - V(s_t;\phi)\right)^2$	On or off Policy	continuous and discrete	Parallelization of actor and critic agents
DDPG	Refer to [32]	Off-policy	Continuous actions	Advantages of DQN and Actor-critic architectures

Table 4.2: Summary of policy gradient RL algorithms

5 Chapter 5 - Optimizing DQN Reinforcement Learning: A Comprehensive Study on the Impact of Variations on performance and training stability

The fundamentals of DQN and its extensions are reviewed in section 4.6.

This section delves into the optimization of a Deep Q-Network (DQN) RL agent specifically employed for control in the CartPole Classic Control Environment.

Moreover, a comparative study explores the impact of convergence criteria, batch size, neural network architecture, and replay memory size, etc. variations on both the training dynamics and the maximum reward achieved by a DQN agent.

The classic control CartPole Gym environment [33] is a benchmark problem in reinforcement learning designed to test the ability of an RL agent to balance a pole on a moving cart. The CartPole Gym Environment is deterministic, and each state (s_t) in the state space consists of four elements: (See Figure 5.1)

- Cart position x (continuous): The horizontal position of the cart on the track.
- Cart velocity v (continuous): The rate at which the cart is moving horizontally.
- Pole angle θ (continuous): The angle of the pole with the vertical axis.
- Pole angular velocity θ' (continuous): The rate at which the pole is swinging.

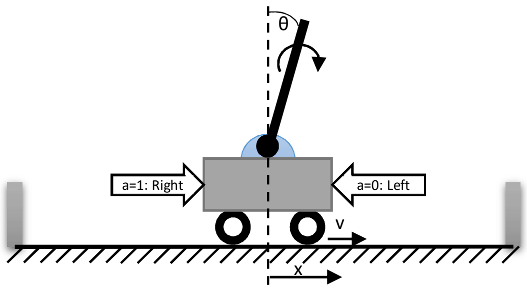


Fig. 5.1: Gym CartPole-v1 Environment

The action space and state space of the environment are represented in tables below.

State Space	State[x]	Min	Max
Cart Position (x)	$s_t[0]$	-4.8	4.8
Cart Velocity (v)	$s_t[1]$	-Inf	Inf
Pole Angle (θ)	$s_t[2]$	-24°	24°
Pole angular Velocity ($\dot{\theta}$)	$s_t[3]$	-Inf	Inf

Action Space	value
Push cart to the Left	0
Push cart to the Right	1

Table 5.1: Action and state spaces of Gym CartPole-v1 Environment

The agent's goal is to apply forces to the cart, either left or right, to prevent the pole from falling over. The task is considered successfully completed if the agent can maintain the pole in an upright position for a specified duration.

A “episode” is considered to be terminated if the agent is unable to maintain the pole angle below ±12° or keep the cart within ±2.4 units from the centre for consecutive time steps, with the optimal position being at zero.

The agent receives a reward of `+1` for each time step it successfully maintains the pole upright.

Reinforcement learning agents, such as Deep Q Networks (DQN), are commonly employed to learn optimal strategies through trial and error.

A wide-range of Deep Q-Network (DQN) variations is employed to solve the classic control problem of CartPole. To comprehensively explore the efficacy and robustness of the learning algorithm, the following variations are incorporated into the experimentation framework:

- Convergence Criteria:
  - all episodes played
  - maximum rewards over a specified number of consecutive episodes
- Optimizers:
  - Root Mean Square Propagation (RMSprop)
  - Adaptive Moment Estimation (Adam)
- Target Network Update:
  - Soft update
  - Hard update
- Neural Network Architecture:
  - with Neural Network Initialization
  - with Batch Normalization Layers
  - Reducing Number of hidden layer from 3 to 2
- Gradient Clipping:
  - Without gradient clipping
  - With gradient clipping
- Hyperparameter Tuning:
  - Learning rate adjustment
  - Gamma Discounting
- Replay Memory:
  - Batch size
  - Size of the replay memory

5.1 Implementation of DQN RL agent for Cartpole Control task

The implementation of the Deep Q-Network (DQN) algorithm is realized through the utilization of key Python libraries, namely PyTorch [34] and Gym [33], [35].[15,16,17] PyTorch, a powerful deep learning framework, is employed to construct and train the neural network model, facilitating efficient computation of complex functions. Gym, an open-source toolkit, provides standardized and customizable environments for developing and testing reinforcement learning algorithms.

Importing requisite libraries:

[15] The entire source code is provided as a supplementary document.
[16] For extraction of "state" and "next_state" from a mini-batch, and calculation of "Q values" the Python code from [36] was used and adapted.
[17] For implementation of a circular buffer behaviour (overwriting old data) the Python code from https://www.kaggle.com/code/dsxavier/dqn-openai-gym-cartpole-with-pytorch was used and adapted.

The requisite libraries, including PyTorch and Gym, are imported

```
"""
@author: Farshad Mirzarazi
DQN Reinforcement Learning CartPole Environment
Variations: Basic DQN, MSE/Hoss Loss function, 2 / 3 Hidden layer FC Network
Hard / soft update of target network, with or without gradient clipping, ...
"""
from collections import namedtuple, deque
import gymnasium as gym
import matplotlib.pyplot as plt
import math

import numpy as np
import random
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

Installed Packages and Versions:
Python Version: 3.9.13 (collections, random, and math are standard libraries of Python interpreter)

- gym==0.26.2
- gymnasium==0.29.1
- matplotlib==3.8.1
- numpy==1.26.2
- torch==2.1.0

<u>Instantiation of the environment:</u>

```
# Step 1: Define the Gym environment
env = gym.make('CartPole-v1', render_mode="rgb_array")
```

<u>Policy and target neural networks:</u>

The policy and target neural networks (NN) are implemented with a three fully connected (FC) hidden layer architecture. The input to the networks corresponds to the state space, consisting of four elements: cart position (x), cart velocity (v), pole angle (θ), and pole angular velocity (θ'). The output layer of both networks provides state-action values for the two possible actions: pushing the cart to the left (action 0) or to the right (action 1). See Figure 5.2.

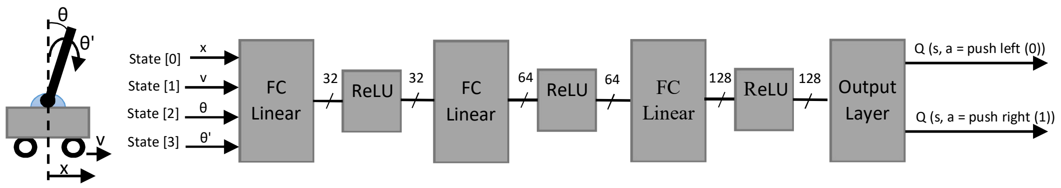


Fig. 5.2: Neural Network Architecture for DQN Agent with Three FC Hidden Layers (Policy and Target Networks)

```
# Step 2: Define the DQN model
class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 32)
        self.layer2 = nn.Linear(32, 64)
        self.layer3 = nn.Linear(64, 128)
        self.out = nn.Linear(128, n_actions)
       def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = F.relu(self.layer3(x))
        return self.out(x)
```

Replay Memory:

Replay memory is an integral component of the DQN implementation, storing tuples of agent experiences (s, a, s', r), where 's' represents the current state, 'a' is the action taken, 's'' is the resulting state, and 'r' denotes the received reward. During the training phase, a randomly selected set of previous experiences, called mini-batch, is sampled from the replay buffer. See Figure 5.3. This mechanism enhances the stability and efficiency of learning by breaking the temporal correlation between consecutive experiences and allowing for more diverse and effective training.

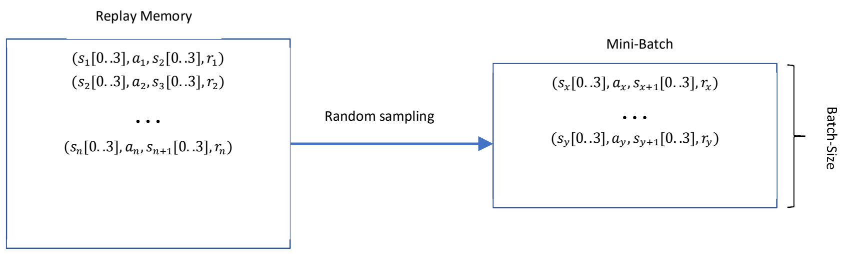

Replay Memory
$(s_1[0..3], a_1, s_2[0..3], r_1)$
$(s_2[0..3], a_2, s_3[0..3], r_2)$
...
$(s_n[0..3], a_n, s_{n+1}[0..3], r_n)$
Random sampling
Mini-Batch
$(s_x[0..3], a_x, s_{x+1}[0..3], r_x)$
...
$(s_y[0..3], a_y, s_{y+1}[0..3], r_y)$
Batch-Size

Fig. 5.3: Storage of agent experience in replay memory and creation of mini-batch for training

```
# Step 3: Define the Replay Memory
class ReplayMemeory:
    def __init__(self,capacity):
        self.BufferLimit = capacity
        # assign memory to the replay buffer
        self.memory = deque([], maxlen=capacity)
        self.storagecount = 0
    def storeExperiment(self, experience) -> None:
        if len(self.memory) < self.BufferLimit:
            self.memory.append(experience)
        else:
            self.memory[self.storagecount % self.BufferLimit] = experience
            self.storagecount += 1
    def CreateMiniBatch(self, batch_size: int):
        return random.sample(self.memory, batch_size)
```

Agent:

Deep Q-Network (DQN) agent class (DQNAgent) is initialized with input (4) for states and output sizes (2) for actions, epsilon-greedy exploration parameters, and two neural networks (policy_net and target_net).
The *select_action* method determines the agent's action based on the epsilon-greedy strategy, and the update_target_model method updates the target network by copying the weights from the policy network. The code also includes global variables for epsilon decay and tracking the episode where epsilon crosses a certain threshold.

```
# Step 4: Define the DQN agent
class DQNAgent:
    def __init__(self, input_size, output_size, epsilon=1.0, epsilon_decay=0.995, 
epsilon_min=0.01):
        self.input_size = input_size
        self.output_size = output_size
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min
        self.policy_net = DQN(input_size, output_size)
        self.target_net = DQN(input_size, output_size)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        # self.optimizer = optim.Adam(self.policy_net.parameters(), lr=0.001)

    def select_action(self, state):
        # Decay epsilon
        global DecayCount
        global epsilon_crossed50Percent
        randNr = random.random()
        self.epsilon = EPS_END + (EPS_START - EPS_END) * math.exp(-1. * DecayCount /
EPS_DECAY)
        if self.epsilon <= 0.5 and epsilon_crossed50Percent is None:
            epsilon_crossed50Percent = episode
        DecayCount += 1
        if self.epsilon > randNr:
            return torch.tensor([[env.action_space.sample()]], device=device,
dtype=torch.long)
        else:
            with torch.no_grad():
                return self.policy_net(state).max(1)[1].view(1, 1)
    def update_target_model(self):
        model_state_dict = self.policy_net.state_dict()
        target_model_state_dict = {}
        for key, value in model_state_dict.items():
            # Convert numpy arrays to tensors
            if isinstance(value, np.ndarray):
                value = torch.from_numpy(value).float()
            target_model_state_dict[key] = value
        self.target_net.load_state_dict(target_model_state_dict)
```

Hyperparameters

```
"""
Hyperparameters:
"""
# Size of mini-batch
batch_size =256
replayMem_size = 10000
# update_rates of target network.
TAU_softUpdate = 0.05
TAU_hardUpdate = 10
# Discount factor
gamma = 0.9
# Epsilon greedy parameters
EPS_START = 1
EPS_END = 0.01
EPS_DECAY = 200
# Optimizer learning rate
LearningRate = 0.0001
num_episodes = 1000
max_reward = 500
```

Training DQN Agent:

The training of the DQN agent involves several key steps, as outlined in section 4.6. The agent undergoes training over a predefined number of episodes, each comprising a maximum of 500 time-steps.

```
# Step 5: Train the DQN agent
num_episodes = 1000
for episode in range(num_episodes):
# reset the env.
initial state: s0 = initial position, velocity of Cart and Pole after reset
state, info = env.reset()
state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)
total_reward = 0
for time_step in range(500):  # number of time steps
```

At each time-step, the agent interacts with the CartPole environment, selecting actions based on an epsilon-greedy strategy.

```
action = agent.select_action(state)  # action selection 0: push left, 1: push right
observation, reward, terminated, truncated, _ = env.step(action.item())
reward = torch.tensor([reward], device=device)
total_reward += reward.item()
```

The resulting experiences (s, a, s', r) are stored in a replay memory, from which a random mini-batch is sampled to create training data.

```
# populate replay memory with agent observation: (state, action, next_state, reward)
ExperienceReplayBuffer.storeExperiment(Experience(state, action, next_state, reward))
########## Training loop - start ##########
if len(ExperienceReplayBuffer.memory) >= batch_size: # BatchSizeReached
    MiniBatch = ExperienceReplayBuffer.CreateMiniBatch(batch_size)
    batch = Experience(*zip(*MiniBatch))

    states= torch.cat(batch.state)
    actions = torch.cat(batch.action)
    rewards = torch.cat(batch.reward)
    next_states = torch.cat([s for s in batch.next_state
                                            if s is not None])
```

In the forward pass, the Q-values for the actions and current states within the mini-batch are computed using the policy network. Simultaneously, the target Q-values for the next states (s') are estimated, considering the Bellman optimality equation. Refer to 4.3.2.2 and 4.6 for more details.

```
# Compute Q (s_t, a) values for each batch state s_t - according to policy_net
q_values = agent.policy_net(states).gather(1, actions)

# Compute Q(s_{t+1}) for all next states.
next_q_values = torch.zeros(batch_size, device=device)
with torch.no_grad():
    next_q_values[non_final_mask] = agent.policy_net(next_states).max(1)[0]

# Compute the target Q-values using the Bellman equation
expected_state_action_values = (next_q_values * gamma) + rewards
```

In the backward pass, the loss function quantifies the difference between the predicted and target Q-values. The chosen loss function (Huber or MSE) is then optimized using the backpropagation process with an Adam optimizer to update the policy network weights.

```
# TODO Loss function variations
# Compute loss function Var.1 Huber loss, Var.2 MSE
# criterion = nn.SmoothL1Loss() # Huber loss
criterion = nn.MSELoss()  # MSE
loss = criterion(q_values, expected_state_action_values.unsqueeze(1))

# Optimize the model
agent.optimizer.zero_grad()
loss.backward()
# TODO with or without gradient clipping
# In-place gradient clipping
torch.nn.utils.clip_grad_value_(agent.policy_net.parameters(), 4)
agent.optimizer.step()
########## Training loop - End ##########
```

<u>Target network:</u>

The target network is updated through a soft or hard update mechanism with update rate represented by hyperparameter Tau (τ).

```
# Step 6: Update target network
# TODO soft or hard update of target network
# Soft update of the target network's weights
# Var.1 Soft update
# θ′ ← τ θ + (1 −τ )θ′
# target_net_state_dict = agent.target_net.state_dict()
# policy_net_state_dict = agent.policy_net.state_dict()
# for key in policy_net_state_dict:
#     target_net_state_dict[key] = policy_net_state_dict[key] * TAU_softUpdate +
target_net_state_dict[key] * (1 - TAU_softUpdate)
# agent.target_net.load_state_dict(target_net_state_dict)
# Var. 2 Hard update
if episode % TAU_hardUpdate == 0:
    agent.target_net.load_state_dict(agent.policy_net.state_dict())
```

Save the model:

Upon completion of training, the trained model is saved for future use.

```
# Step 7: Save the trained network
torch.save(agent.policy_net.state_dict(),
'F:\\Brunel\\02_SourceCodes\\DQNVariations\\online_model.pth')
print("Model saved.")
```

DQN Evaluation

After training the model, the next step is to evaluate its performance by running 100 episodes. The trained model is reloaded to assess its behaviour in different scenarios. During these 100 episodes, the model's ability to achieve satisfactory results will be observed and analysed.

```
# Step 8: reload the trained network
agent.policy_net.load_state_dict(torch.load('F:\\Brunel\\02_SourceCodes\\DQNVariations\\online
_model.pth'))
print("Model loaded.")

# Step 9: Performance evaluation of DQN Agent
total_reward = 0
for _ in range(100):  # Run the environment 10 times for evaluation
    state, info = env.reset()
    state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)
    for _ in range(500):  # You can adjust the maximum number of time steps
        action = agent.select_action(state)  # 100% exploitation
        observation, reward, terminated, truncated, _ = env.step(action.item())
        # reward = torch.tensor([reward], device=device)
        done = terminated or truncated
        if terminated:
            next_state = None
        else:
            next_state = torch.tensor(observation, dtype=torch.float32,
device=device).unsqueeze(0)
        total_reward += reward
        state = next_state
        if done:
            break
average_reward = total_reward / 100
print(f"Average Reward over 100 episodes: {average_reward}")
```

Results:

The training progress is visualized through plots of total rewards and epsilon values over episodes, offering insights into the agent's learning dynamics and convergence.

Monitoring includes the visualization of total rewards and epsilon values over episodes and time-steps.

```
# Step 10: Plot the Results
def Plot_Result(episode_rewards, epsilon_values, epsilon_crossed50Percent, last_50_means,
episode_last_50_mean_gt_400, average_reward):
    if episode_rewards:
        episodes, rewards = zip(*episode_rewards)

        figsize = (12, 6)  # Adjust the width and height as needed
        fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=figsize)
        ############ ax1 ############
        ax1.plot(episodes, rewards, label='Total Reward', color='green')
        ax1.axhline(y=max_reward, color='orange', linestyle='--', label='Max Reward')
        ax1.set_ylabel('Total Reward')
        ax1.grid(True)

        # Plot Last 50 Mean Reward with label
        ax1.plot(episodes[-len(last_50_means):], last_50_means, color='blue', label='Mean Last
50 Reward')
        ax1.legend(loc='upper left')
        # Add a vertical line at the episode where last_50_mean > 400
        if episode_last_50_mean_gt_400 is not None:
            ax1.axvline(x=episode_last_50_mean_gt_400, color='purple', linestyle='--',
label='Mean Last 50 Mean > 400')
            ax1.text(0.5, -0.1, f'Training stop at episode: {episode_last_50_mean_gt_400}',
transform=ax1.transAxes,
                     fontsize=10, ha='center')
        ############ ax2 ############
        ax2.plot(episodes, epsilon_values, label='Epsilon', color='red')
        ax2.axvline(x=epsilon_crossed50Percent, color='gray', linestyle='--', label='Epsilon =
0.5')
        ax2.set_xlabel('Episode')
        ax2.set_ylabel('Epsilon')
        ax2.legend(loc='upper right')
        ax2.grid(True)
        ############ subplots and subtitle ############
        # Adjust the vertical position of the entire plot
        plt.subplots_adjust(left=0.06, bottom=0.12, right=0.97, top=0.95, wspace=0, hspace=.2)
        # Display average reward below ax2
        fig.text(0.5, 0.02, f'Average Reward of Trained model: {average_reward}', ha='center',
fontsize=10)
        plt.suptitle('DQN Training: Total Reward and Epsilon over Episodes')
        plt.show()
    else:
        print("No episode durations recorded. Cannot plot.")
```

And finally, the Gym environment is closed.

```
# Step 11: Close the environment
env.close()
```

5.2 Evaluation of DQN Performance, Learning Dynamics and Convergence

The presented plots in this section, offer a comprehensive view of the training progress for the Deep Q-Network (DQN) agent in the CartPole environment. The visualization of crucial metrics across episodes and time-steps provides insights into the learning dynamics and convergence behaviour of the agent.

Training Process Overview:

The training process is an iterative journey where the DQN agent refines its decision-making abilities in the CartPole environment. This refinement occurs through a series of episodes and time-steps, gradually enhancing the agent's understanding of optimal actions and strategies.

Training Configuration:

The DQN agent undergoes training for 1000 episodes, each consisting of 500 time-steps. The maximum achievable reward in a single episode is crapped at 500, reflecting the duration the agent can successfully keep the cart and the pole within the acceptable range.

Explanation of the result:

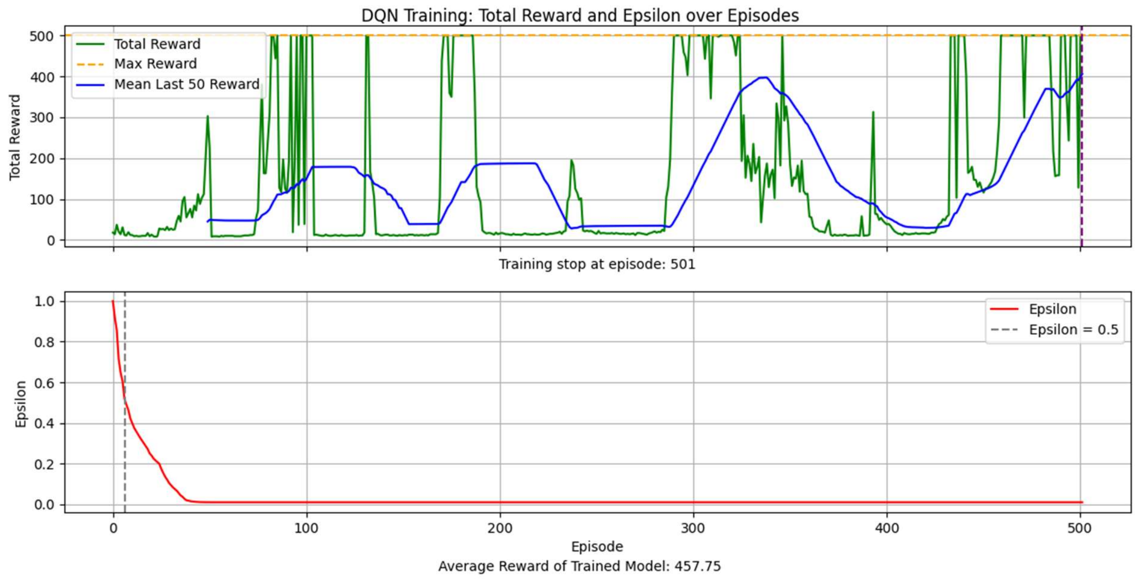


Fig. 5.4: DQN Training Performance: Total Reward Dynamics Over Episodes

Total Reward (AX1):

The first plot (AX1) depicts the total reward (green) achieved by the agent in each episode. The total reward is indicative of how effectively the agent can maintain the stability of the cart and the pole over time. The orange dashed line indicates the maximum achievable reward of 500 in each episode.

The Last 50 Mean Reward, shown in blue, offers a filtered representation of the agent's recent performance. Calculated by averaging the total reward across the last 50 episodes, this metric serves as a stable and informative measure for assessing the training quality of the agent in the task. Notably, the Last 50 Mean Reward acts as an alternative convergence criterion, leading to the termination of training once it surpasses the threshold of 400.

The legend text, "Training stopped at episode: **501**," below plot 1 indicates that the agent has achieved a performance level where the Last 50 Mean Reward is greater than or equal to 400.

Decayed Epsilon (AX2):

The second plot (AX2) illustrates the balance between exploration and exploitation through the decayed epsilon values. Epsilon is a parameter controlling the agent's exploration strategy.

The red line represents the decayed epsilon values over episodes. A higher epsilon implies more exploration, allowing the agent to freely experience the environment and discover optimal strategies.

The grey dashed line marks the episode at which epsilon crosses the 50% threshold. This point signifies a transition in the agent's behaviour, balancing exploration and exploitation. As epsilon decreases, the influence of Q-learning becomes more noticeable, emphasizing the exploitation of learned knowledge.

The trained model undergoes an additional evaluation by executing 100 episodes post-training, and its performance is assessed. The legend text, "Average Reward of Trained model: **457.75**" (out of a possible 500), situated below plot 2, serves as an indicator of the agent's performance. In this instance, the rating is designated as "very good" performance.

Observations:

1. Training stability - Volatility in Agent Performance:

The agent's performance exhibits frequent unexpected drops in total reward when confronted with unseen or extreme data (e.g. at episodes: {**102, 185,323,470,483**}.

Recovery from such drops requires a substantial number of episodes (e.g. at intervals: {**[185-289]**, **[324-433]**}, highlighting the limitation of the neural network to rapidly respond to significant loss values. This behaviour may be attributed to the sophistication of the neural network's architecture or factors such as the learning rate.

To address this stability issue, a close analysis of q-values and loss function values can provide insights for effective resolution. However, it's important to note that a comprehensive solution may go beyond what's covered in this section.

2. Steady Improvement of Mean-Last-50-Total-Reward Over Time

The Mean-Last-50-total-reward, on the other hand, demonstrates a consistent upward trend over time (e.g. intervals {**[287-333]**, **[431-501]**}), indicating an overall improvement in the agent's performance.

3. Hyperparameter Sensitivity:

The agent's performance is highly sensitive to hyperparameters and their variations. While adjusting hyperparameters may address specific training issues, it introduces the challenge of potentially negligible negative impacts on other aspects of training. More information will be provided in 5.3.1.6.

4. Transition from Exploration to Exploitation:

The total reward experiences a positive trend when the agent shifts from exploration to exploitation (e.g., at episode **24**, where the exploration rate is around **20%**, and at episode **75**, where the exploration rate is almost **0%**), showcasing the effectiveness of Q-learning. However, the lack of sufficient experience becomes evident as the agent struggles to control the cart and pole in subsequent episodes, leading to next sudden drops in total reward.

5. Experiment Randomness:

The experiment exhibits a significant level of randomness, making it challenging to achieve consistent reproducibility in results and trends. The outcomes vary, including instances of early convergence, late convergence, and occasional non-convergence. The performance of the trained model ranges from very good to poor, with the latter being an infrequent occurrence.

Early convergence:

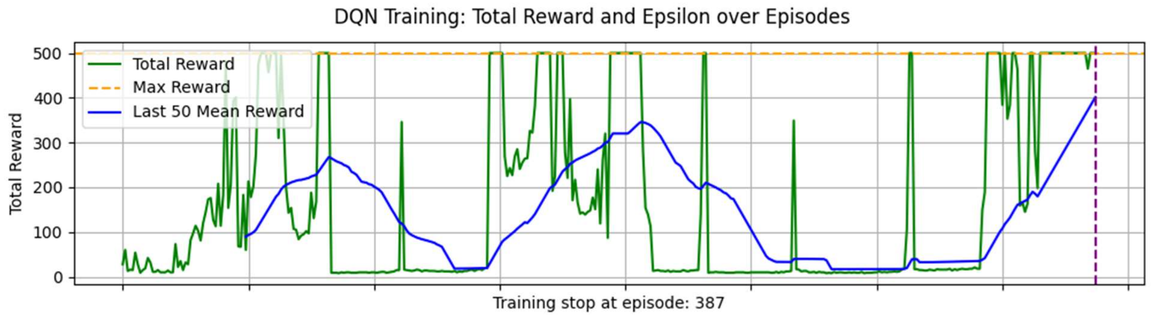


Fig. 5.5 (a): DQN Training Performance: example of early convergence

non-convergence:

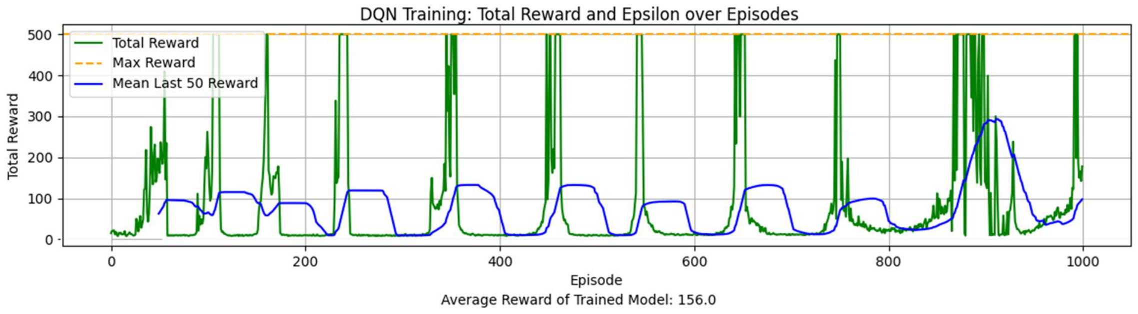


Fig. 5.5 (b): DQN Training Performance: example of non-convergence

6. Correlation Between Average Reward and Total Reward:

The average reward of the trained agent is closely correlated with and reflective of the total reward achieved before the training concludes. This correlation underscores the significance of the total reward as a crucial metric for evaluating the agent's training, indicating its capacity to achieve satisfactory performance. See 5.3.1.1.

5.3 DQN Variations

In this subsection, an array of variations is applied to the previously implemented DQN for the CartPole environment. The impact of these variations on the learning curve and overall performance is empirically examined. These variations include convergence criteria, loss functions, optimizers, target network update, neural network architecture, gradient clipping, hyperparameter tuning, and replay memory.

It is important to note that, due to the inherent randomness in the experiment, achieving consistent reproducibility in results and trends proves challenging. Hence, the subsequent analysis considers this challenge and provides a generalized evaluation of the variations.

5.3.1.1 Convergence Criteria

- Variant 1: All Episodes played
- Variant 2: Maximum rewards over a specified number of consecutive episodes

Utilizing Variant 1 implies that the training process continues until the agent has experienced all episodes, ensuring a comprehensive exploration of the environment. This may lead to a more exhaustive learning process.

Variant 2 is considered to be a more targeted approach. This method not only reduces the training duration, but also minimizes the risk of overwriting better model weights with potentially misleading data. However, it's important to note that Variant 2 comes with a potential risk of premature convergence when the maximum reward is quickly reached. This may expose the agent to a significant amount of unseen data, posing a challenge for robust generalization.

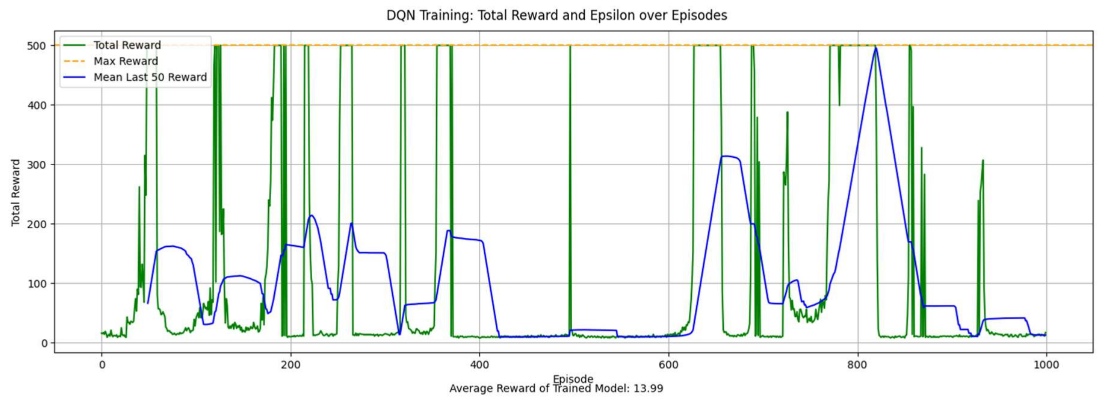


Fig. 5.6: DQN Training Performance: all 1000 episodes played

The DQN agent exhibits poor performance after training (average reward = 13.99 out of 500) because the network weights are far from optimal values. Alternatively, the agent could halt the training process at episode 807, where the mean last 50 rewards surpass the threshold of 400 rewards, potentially updating the model to achieve significantly better performance.

5.3.1.2 Optimizers

- Variant 1: Root Mean Square Propagation (RMSprop)
- Variant 2: Adaptive Moment Estimation (Adam)

Optimization algorithms are briefly explained in 3.3.3. The most commonly used optimizers in backpropagation process of neural networks include Gradient Descent, Stochastic Gradient Descent (SGD), AdaGrad, RMSProp, and Adam [37]. RMSProp (Tieleman & Hinton, 2012; Graves, 2013), and Adam (Kingma & Ba, 2015) are considered to be advanced variants of optimizers as they adaptively adjust the learning rate and incorporate momentum—exponentially weighted averages of past gradients—into their optimization algorithms [38].

The advantage of RMSProp is that it reduces oscillations, enabling the potential for higher learning rates or larger algorithm steps, which aids in convergence [39]. However, very large step sizes in RMSProp can be disadvantageous and often lead to divergence [40]. Adam is known for its fast convergence and robustness to different types of datasets and architectures. It demonstrates better convergence than other methods [40].

Readers interested in a detailed explanation of both algorithms can refer to relevant sources.

Code Snippet to select an optimizer variation:

```
# TODO variation of agent optimizer
# Instantiate the optimizers with learning rate
# SGD optimizer
optimizer_sgd = optim.SGD(agent.policy_net.parameters(), lr=LearningRate)
# Adam optimizer
optimizer_adam = optim.Adam(agent.policy_net.parameters(), lr=LearningRate)
# RMSprop optimizer
optimizer_rmsprop = optim.RMSprop(agent.policy_net.parameters(), lr=LearningRate)
# choose which optimizer to use
# agent.optimizer = optimizer_sgd
# agent.optimizer = optimizer_adam
agent.optimizer = optimizer_rmsprop
```

Empirical experimentation with the DQN Agent using LR = 0.001 and LR = 0.0025 (figures 5.7 and 5.8) shows that the Adam optimizer exhibits a faster convergence behaviour.

In both cases, Adam converges faster, with LR = 1e-4 and LR = 25e-4, and its model evaluation demonstrates good results. Increasing the learning rate to 25e-4 (25 times larger) results in divergence for DQN when using the RMSProp optimizer, as demonstrated in the 2nd plot in Figure 5.8.

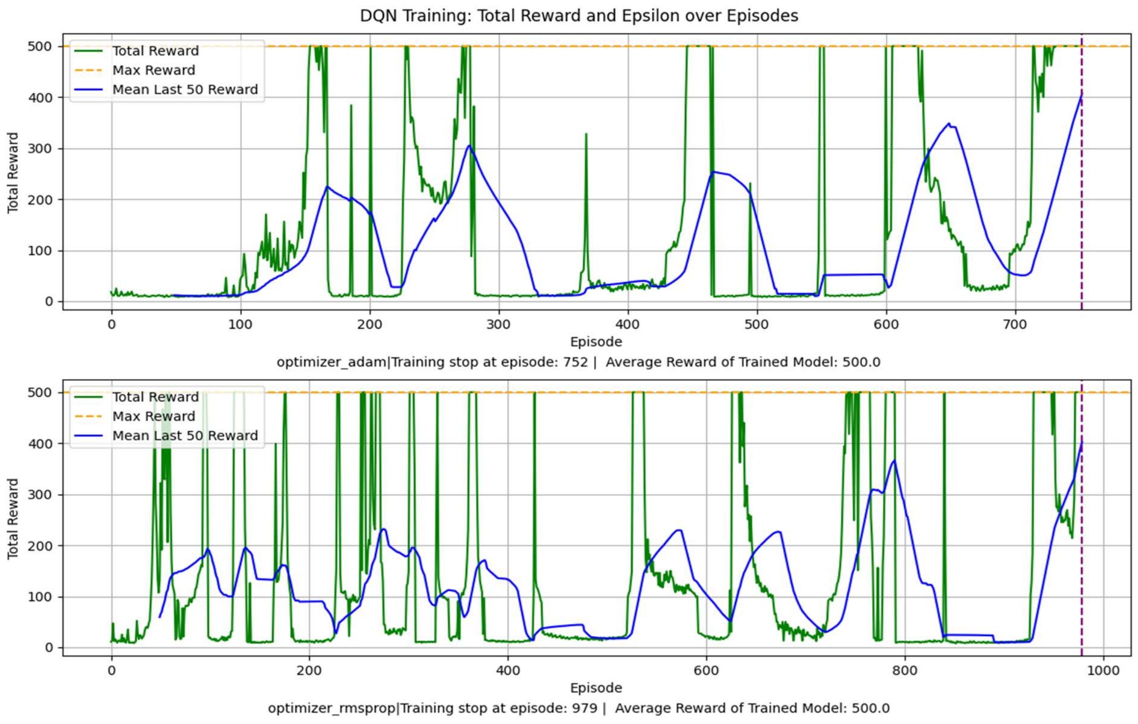


Fig. 5.7: DQN Training Performance: Optimization ADAM vs RMSProp; LR = 0.0001

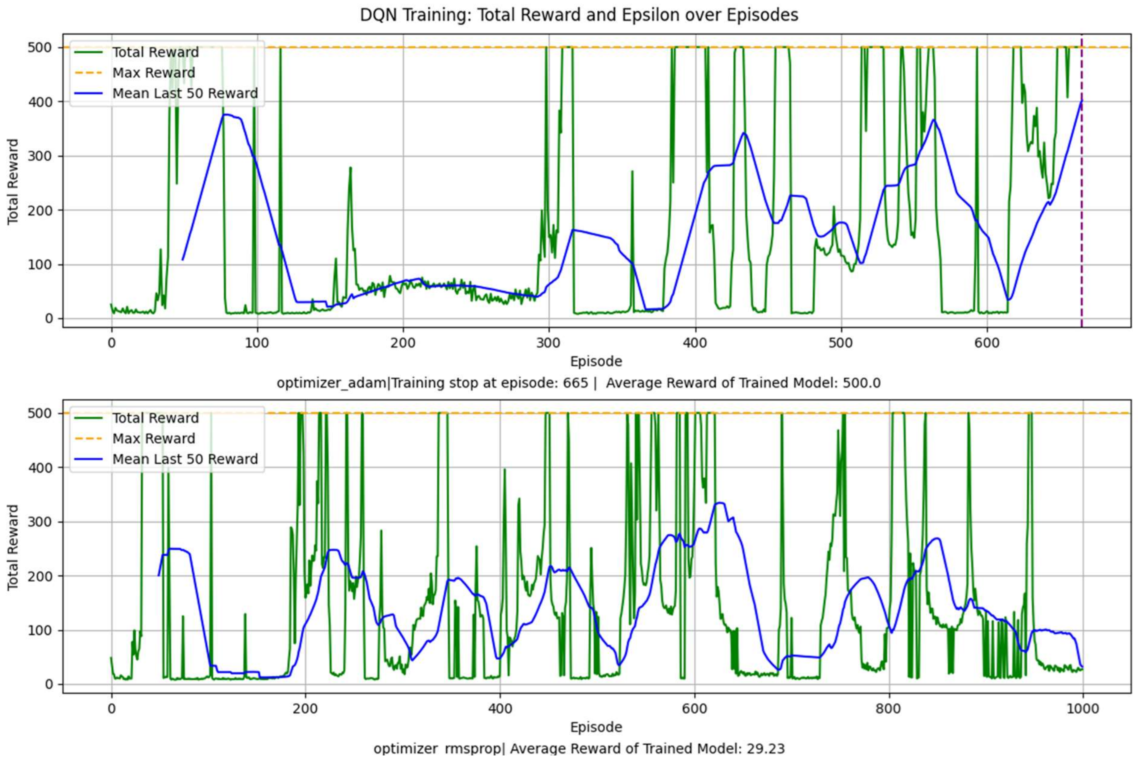


Fig. 5.8: DQN Training Performance: Optimization ADAM vs RMSProp; LR = 25 e^-4

5.3.1.3 Target Network Update

- Variant 1: Hard update
- Variant 2: Soft update

In DQN (Deep Q-Network) training, the target network is employed at each time step to estimate the expected action-state values. Continuous updates to the target network are crucial for effective learning. Two prominent paradigms for updating the target network are soft updates and hard updates, each providing distinct strategies for synchronization with the policy network.

In their original work, DQN inventors - Minh et al. - applied a hard-update mechanism every c steps to the target network [9]. See section 4.6.1. The hard update mechanism copies the complete parameter set of the policy network to the target network. In the current implementation, TAU_hardUpdate represents the frequency of hard updates, meaning a hard update occurs every TAU_hardUpdate episodes.

In contrast, soft-update mechanism, proposed by Timothy P. Lillicrap, et al, blends the parameters (θ) of the policy network (agent.policy_net) into the parameters of the target network (agent.target_net) represented by θ' in every time-step [41]. This blending is controlled by the hyperparameter τ as follows:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'; \text{with } \tau \ll 1. \tag{5.3.1}$$

Code snippet for soft / hard target network update:

```
# TODO soft or hard update of target network
# Soft update of the target network's weights
# Var.1 Soft update
# θ' ← τ θ + (1 −τ )θ'
target_net_state_dict = agent.target_net.state_dict()
policy_net_state_dict = agent.policy_net.state_dict()
# for key in policy_net_state_dict:
#     target_net_state_dict[key] = policy_net_state_dict[key] * TAU_softUpdate +
target_net_state_dict[key] * (1 - TAU_softUpdate)
# agent.target_net.load_state_dict(target_net_state_dict)
# Var. 2 Hard update
if episode % TAU_hardUpdate == 0:
    agent.target_net.load_state_dict(agent.policy_net.state_dict())
```

The authors in [42] argue that their soft-update mechanism significantly enhances training stability, and they assert that any potential sluggishness in training is negligible. In our experiments, while acknowledging the inherent randomness in results, a similar trend was observed (see Figure 5.9): (a) hard-update, (b) soft-update. Additionally, it was observed that the hard-update variation is more sensitive to hyperparameters.

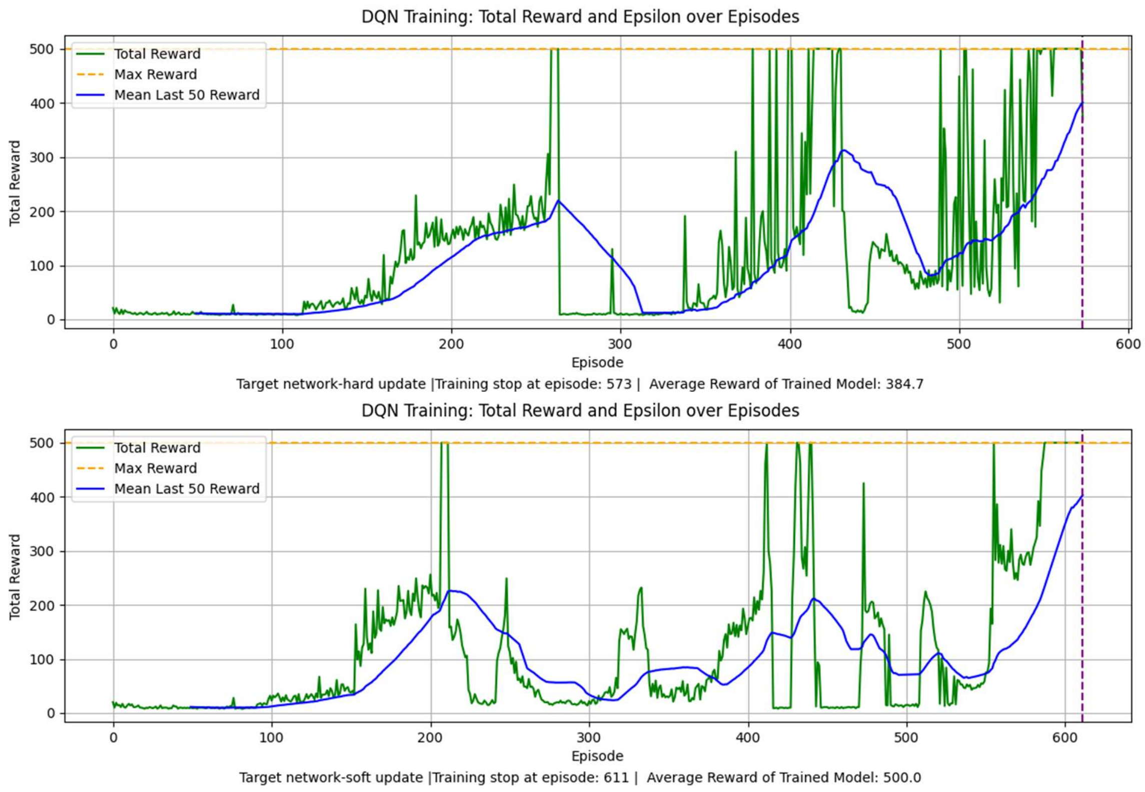


Fig. 5.9: DQN Training Performance: Target Network update. (a) hard-update, (b) soft-update

5.3.1.4 Gradient Clipping

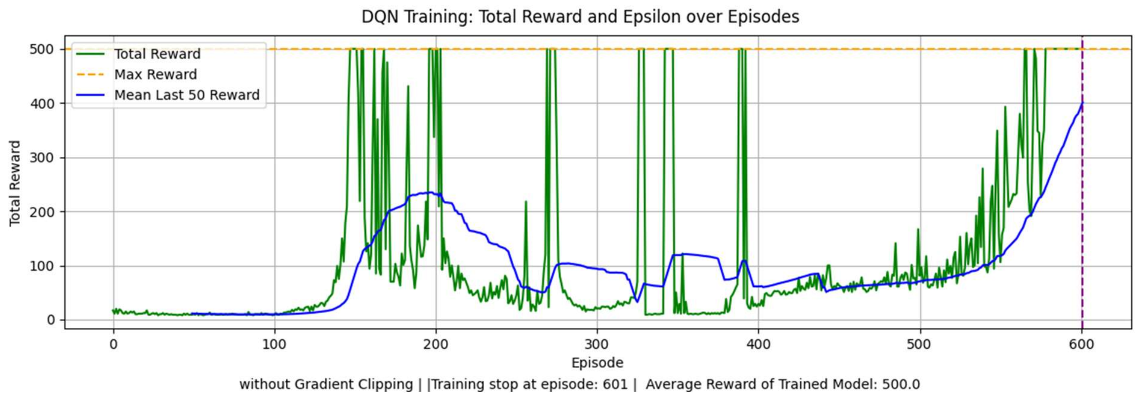
- Variant 1: Without gradient clipping
- Variant 2: With gradient clipping (moderate, intense)

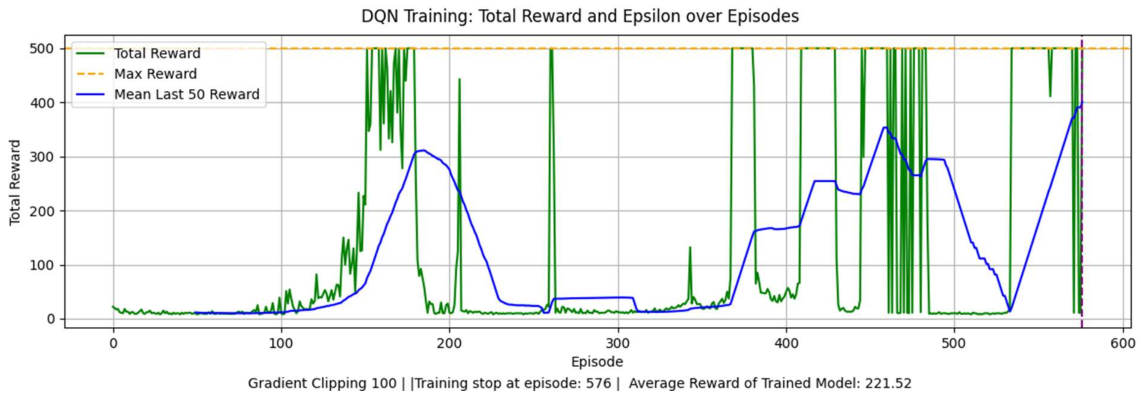
In DQN algorithm proposed by Mnih et al., the clipping of the temporal difference (TD) error, which is used to calculate the loss, is recommended. The clipping is typically applied to ensure that the magnitude of the TD error remains within a certain range, often between -1 and 1 [9]. This helps to mitigate issues related to the instability of training that may arise from large TD errors. They also suggested the clipping of rewards to limit the impact of extreme values on the learning process.

Clipping of the gradient is considered good practice in various DQN variants, such as [4], aiming to restrict aggressive gradient updates. While large gradients might contribute to faster adaptation in the training process, they pose a potential risk of instability.

Code snippet to apply gradient clipping:

```
# Optimize the model
agent.optimizer.zero_grad()
loss.backward()
# TODO with or without gradient clipping
# In-place gradient clipping
torch.nn.utils.clip_grad_value_(agent.policy_net.parameters(), 40)
agent.optimizer.step()
```





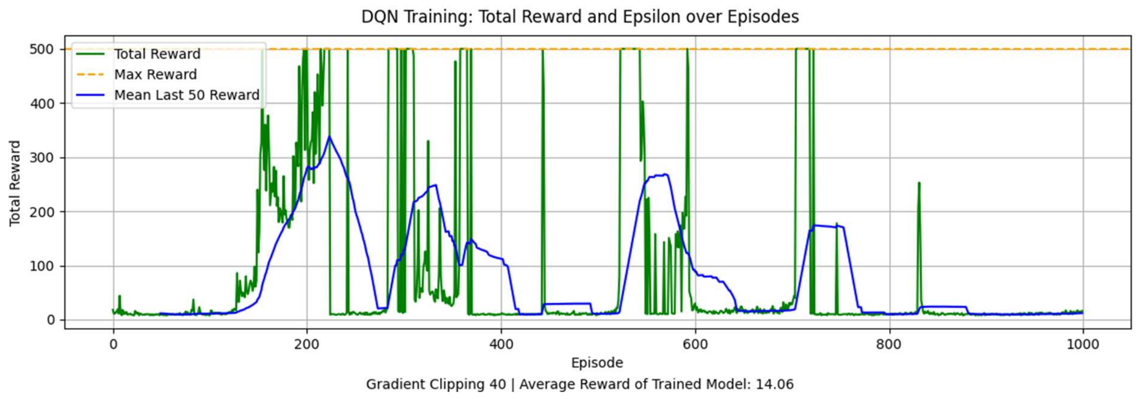


Fig. 5.10: DQN Training Performance: Gradient Clipping. (a) w/o (b) with moderate Gradient Clipping, (c) with intense Gradient Clipping

In the experimentation, the application of gradient clipping is observed to have a favourable impact on the stability of training, as evidenced by Figure 5.10 (b) and (c). However, this positive effect comes at the cost of a slower or non-convergent training process. Notably, in scenario (a), where gradient clipping is not applied, the training speed is faster, and the average reward successfully reaches the 400 level.

5.3.1.5 Neural Network Architecture

- Variant 1: with Neural Network Initialization
- Variant 2: with Batch Normalization Layers
- Variant 3: Reducing Number of hidden layer from 3 to 2

<u>Neural Network Initialization</u>

It has long been known that initialization of neural networks plays a crucial role in determining the success of training. Inadequate or non-optimized initialization can lead to issues such as the vanishing gradient problem, where the gradients during backpropagation become extremely small, hindering the learning process. This challenge is discussed in detail in Section 2.6, emphasizing its impact on the convergence and effectiveness of neural networks.

To address initialization challenges, researchers have proposed various techniques, and one notable advancement is He initialization, introduced by Kaiming He et al. He initialization is specifically designed for rectified linear units (ReLUs) [42], which is used activation function in our neural network.

Code snippet to initialized the neural network with He method:

```
def weights_init_he(m):
    if isinstance(m, nn.Linear):
        nn.init.kaiming_uniform_(m.weight.data, nonlinearity='relu')
        if m.bias is not None:
            nn.init.constant_(m.bias.data, 0)
class DQNAgent:
…….
        self.policy_net = DQN(input_size, output_size)
        self.policy_net.apply(weights_init_he)  # Apply He initialization
        self.target_net = DQN(input_size, output_size)
        self._hard_update_target_net()
```

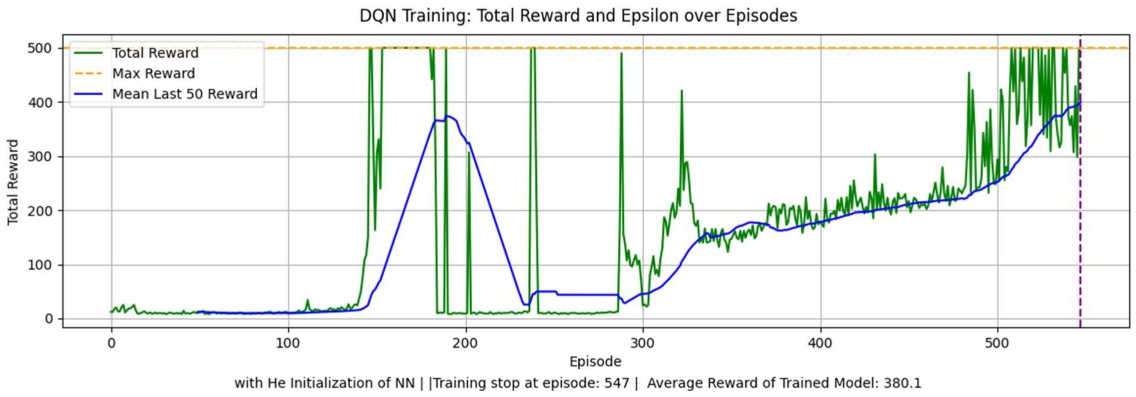


Fig. 5.11: DQN Training Performance: Effect of Neural Network Initialization

After applying the He Initialization scheme, a positive effect on the total reward (green plot shifted upwards) was observed. Notably, after the exploration phase (see Decayed Epsilon (AX2) in Figure 5.3), the neural network with this initialization exhibits a rapid increase in total rewards.

<u>Batch Normalization</u>

Training Neural Networks is challenging due to the dynamic input distribution across layers during training. This necessitates lower learning rates and meticulous parameter initialization. In 2015, Google researchers (Ioffe and Szegedy) introduced an innovative approach to enhance the training speed of feed-forward and convolutional neural networks through batch normalization (BN), allowing for a significant increase in the learning rate [43]. Batch normalization in training neural networks involves normalizing the output of layers before they are fed into the subsequent layer. One common normalization approach is adjusting each input variable to have a zero mean and unit variance [1].

A batch normalization layer is commonly positioned between the fully connected/convolution layer and its associated activation function.

Code snippet to add Batch Normalization to the network:

```
# Step 2: Define the DQN model
class DQN_BN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 32)
        self.batch_norm1 = nn.BatchNorm1d(32)
        self.layer2 = nn.Linear(32, 64)
        self.batch_norm2 = nn.BatchNorm1d(64)
        self.layer3 = nn.Linear(64, 128)
        self.batch_norm3 = nn.BatchNorm1d(128)
        self.out = nn.Linear(128, n_actions)

    def forward(self, x):
        x = F.relu(self.batch_norm1(self.layer1(x)))
        x = F.relu(self.batch_norm2(self.layer2(x)))
        x = F.relu(self.batch_norm3(self.layer3(x)))
        return self.out(x)
```

After implementing batch normalization, the experiment did not exhibit improvement; rather, it posed challenges for the DQN agent to converge and attain a satisfactory level of rewards. Despite extensive efforts and thorough adjustment of hyperparameters such as the learning rate, we encountered challenges in achieving success in our experimentation. While batch normalization is considered advantageous for training of neural networks, the lack of success in the experiments may be attributed to our implementation and not proper use of the PyTorch library.

Number of hidden layer

Varying the number of hidden layers can impact the agent's ability to capture complex relationships in the data, with deeper architectures potentially learning more complex features. Different methods, such as those outlined in [44], aim to standardize the selection of the optimal number of hidden layers for neural networks. However, it's essential to note that the optimal number of hidden layers depends on the objective of the application. Researchers in [45] explored how varying the number of hidden layers affects the efficiency of neural networks in terms of time complexity and accuracy. Their findings indicate that employing a large number of hidden layers can slow down the training process, and unnecessary increases in neurons or layers may lead to overfitting issues. Nevertheless, if model accuracy is a priority, selection of a larger number of hidden layers is considered more suitable.

In this experiment, the number of hidden layers in the Policy and Target networks is reduced from 3 layers to 2 layers by removing the last hidden layer in the current implementation.

Code snippet to reduce the number of hidden layers from 3 to 2:

```
class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 32)
        self.layer2 = nn.Linear(32, 64)
        #self.layer3 = nn.Linear(64, 128)
        self.out = nn.Linear(64, n_actions)

    # Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        #x = F.relu(self.layer3(x))
        return self.out(x)
```

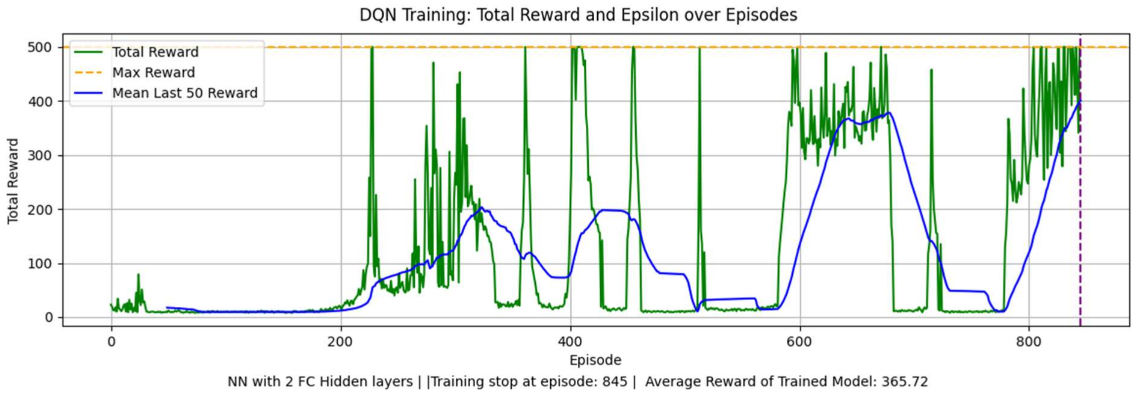


Fig. 5.12: DQN Training Performance: Policy/Target Networks with 2 hidden layers

The decrease in model accuracy is evident in Figure 4.18 when using only 2 hidden layers, as the agent fails to achieve the maximum reward (500, orange plot) in many episodes, unlike its performance with 3 hidden layers.

5.3.1.6 Hyperparameter Tuning

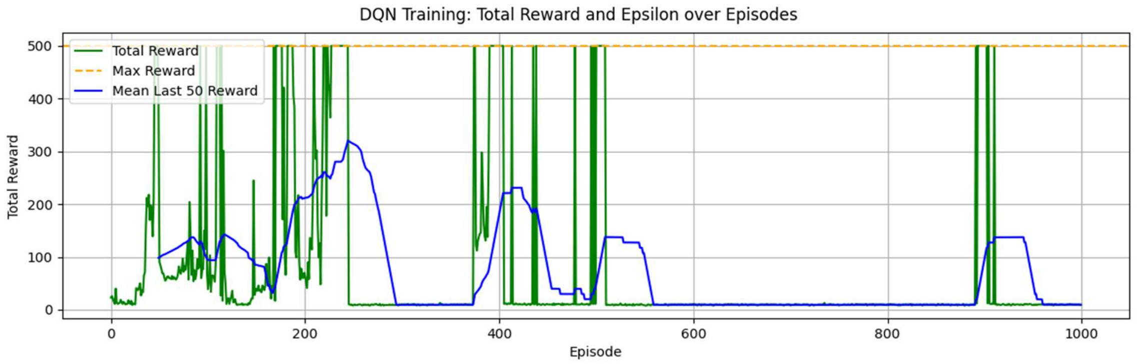
- Variant 1: higher and lower Learning rates
- Variant 2: lower Gamma Discounting

In Section 3.3.2 of this dissertation, which delves into the influential role of the learning rate (η) in neural network training, it is emphasized that tuning the learning rate influences the step size during optimization and significantly affects the speed and outcome of the training process. A lower learning rate enhances precision during training, allowing for meticulous weight adjustments. Conversely, higher learning rates may risk overlooking optimal weight settings.

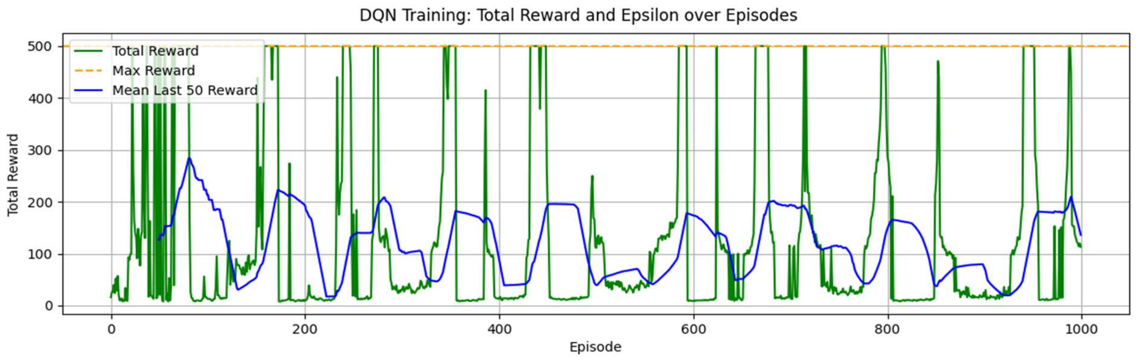
The learning rate in this implementation has been fixed at a predefined value of LR = 0.0001. Figure 5.13 a, b, and c present experiments conducted with higher learning rates—0.01, 0.001, and 0.0008, respectively. These rates are 100, 10, and 8 times larger than our predefined LR. Additionally, Figure 5.13 (d) represents LR = 0.00005, which is half of our predefined LR, demonstrating the impact of a lower learning rate on training dynamics.

Despite their higher step sizes, agents with elevated learning rates (a) and (b) struggle to efficiently update network weights and fail to achieve high rewards within 1000 episodes. This leads to evident training instability, resulting in consistently low performance of the trained model. On the other hand, the agent with LR = 0.0008 (c) performs better, converging at episode 671. However, repeated experiments at this learning rate revealed a potential risk of premature convergence, leading to suboptimal performance after training.

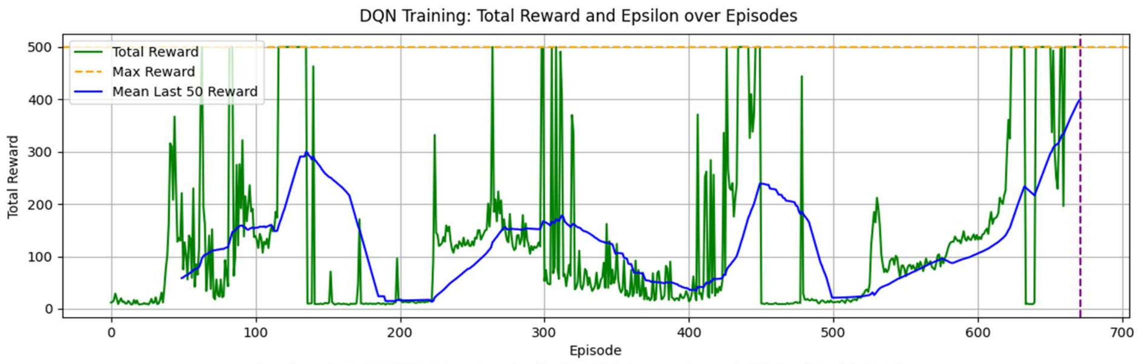
The agent with LR = 0.00005, half of the predefined LR, understandably takes more time to offset smaller rewards given its reduced step size. However, no substantial improvement in performance accuracy was observed with this lower learning rate.



learning rate 0.01 | | Average Reward of Trained Model: 9.59



learning rate 0.001 | | Average Reward of Trained Model: 113.72



learning rate 0.0008 | |Training stop at episode: 671 | Average Reward of Trained Model: 500.0

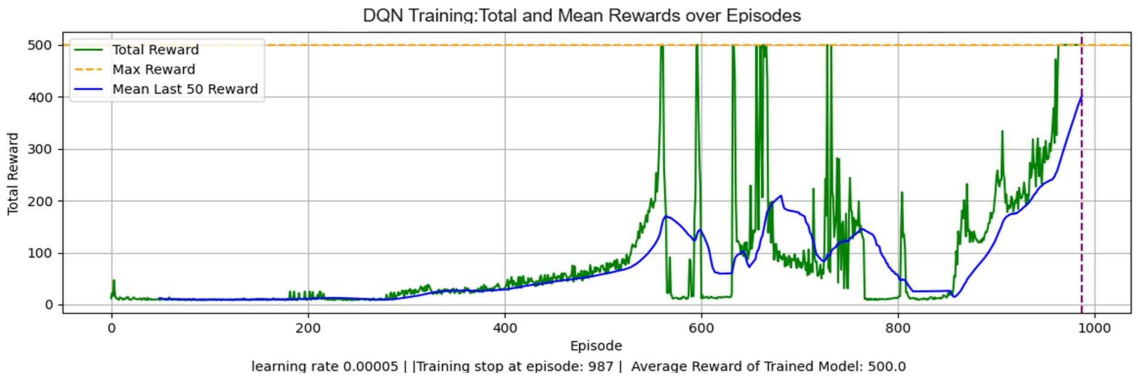
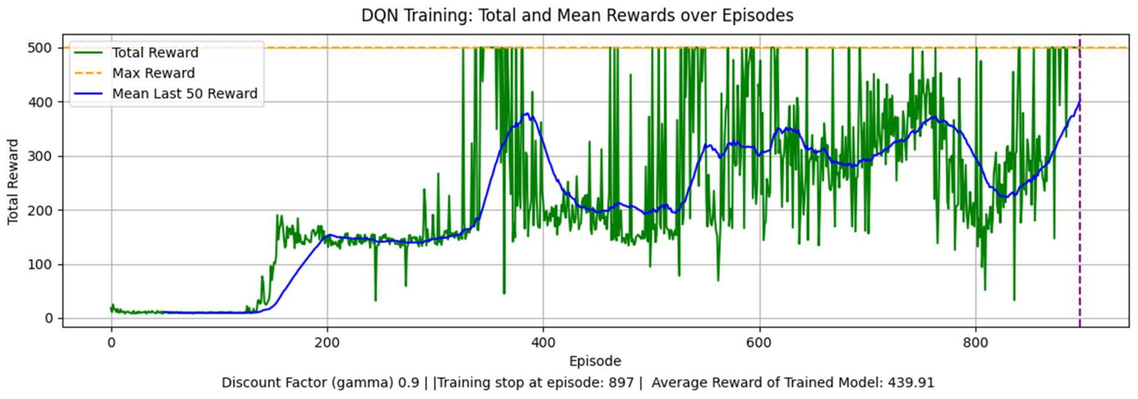


Fig. 5.13: DQN Training Performance: impact of Learning rate on training. (a) LR = 0.01, (b) LR = 0.001, (c) LR = 0.0008, (d) LR = 0.00005

The discount factor (gamma) in reinforcement learning plays a crucial role as a hyperparameter, influencing how an agent values future rewards. It directly affects the trade-off between immediate and future rewards in the agent's decision-making process. As evident in formula 4.1.6 (section 4.3.1), this factor, ranging from 0 to 1, determines the weight assigned to previous rewards. A higher gamma emphasizes long-term consequences, encouraging actions that lead to greater cumulative rewards, while a lower gamma reduces the impact of future rewards, focusing more on immediate rewards.

The choice of the discount factor (gamma) depends on the specific characteristics of the environment and the task. In this implementation, gamma is fixed at a predefined value of 0.999 (almost 1). Adjusting gamma to lower values (0.9 and 0.5) allows tailoring the agent's behaviour to prioritize immediate rewards more than previous rewards.



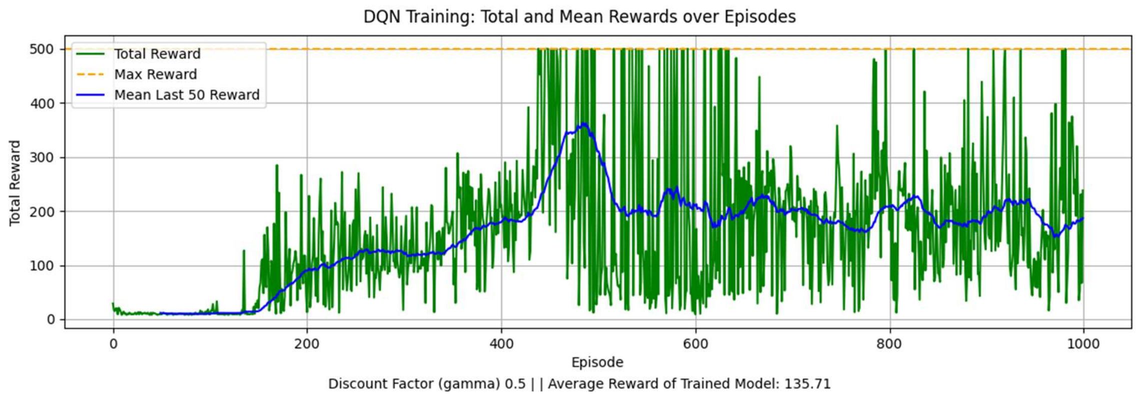


Fig. 5.14: DQN Training Performance: impact of Discount factor on training. (a) gamma = 0.9, (b) LR = 0.05

Repeated iterations prioritizing immediate rewards with gamma set to 0.9 demonstrate a robust progressive trend. The DQN agent continuously learns and exhibits better performance than with gamma = 0.99 (Figure 5.14 a). Further reducing the discount factor to 0.5 devalues previous rewards, leading to increased training instability, and the agent struggles to converge and achieve higher levels of reward. (Figure 5.14 b)

5.3.1.7 Replay Memory:

- Variant 1: smaller Batch size
- Variant 2: smaller Size of the replay memory

<u>Batch size</u>

The intuition behind employing a randomly selected batch of past experiences in DQN and its decorrelation effect on training stability were elaborated in section 4.6.1. A reduction in batch size in DQN worsens the correlation between consecutive experiences, leading to a higher dependency that impairs the learning process and negatively impacts the model's generalization ability, thereby hindering overall DQN performance. See Figure 5.15.

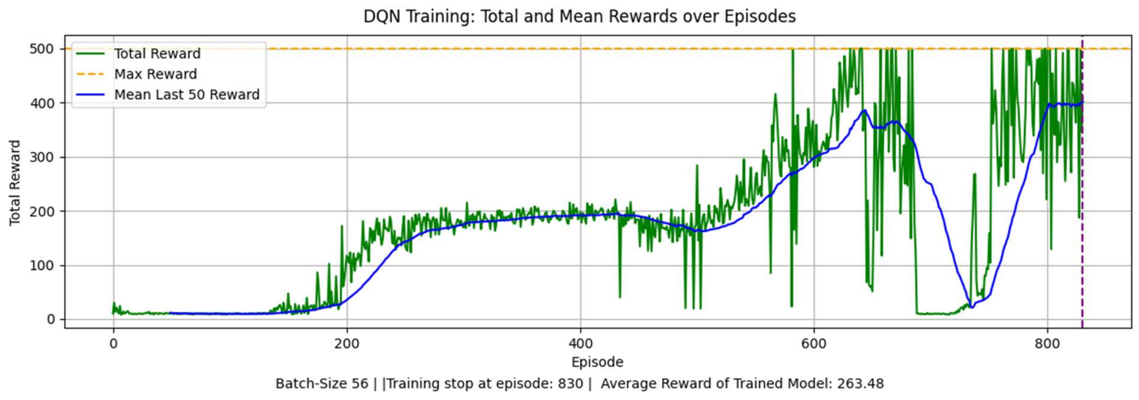


Fig. 5.15: DQN Training Performance: mini-batch size reduction from 256 to 56 samples

A larger batch size, on the other hand, can be computationally expensive due to the increased computational workload in processing a higher amount of batch data during both the forward and backward passes in each training iteration.

Replay memory

The latest experiences of the DQN agent are added to the end of the replay memory.

When the memory reaches its maximum capacity (replayMem_size = 10000), new data will overwrite the oldest data in the memory (A circular buffer behaviour).

Reducing the replay memory size in a DQN implementation limits the total number of experiences retained by the agent for sampling in the mini-batch. It can negatively impact the diversity of experiences available for learning, potentially affecting the agent's ability to effectively generalize from past interactions.

In our experiments, reducing the size of the replay memory from 10k to 5k and even to 2k and 1k did not result in any noticeable negative impact on training dynamics and agent performance. This could be attributed to the inherent simplicity and efficiency of the CartPole task, which may not necessitate a larger replay memory to adequately capture a diverse set of experiences for effective learning.

5.4 Conclusion

Various configurations and strategies in training Deep Q-Networks (DQN) for the CartPole environment has been explored. Our experimentation covered diverse aspects, including convergence criteria, optimizers, target network update mechanisms, neural network architecture variations, gradient clipping, hyperparameter tuning, and replay memory considerations.

Key Findings:

Mitigating Experiment Randomness: To address the inherent randomness in the experiments, multiple iterations were conducted, ensuring the reliability and consistency of results. This approach establishes a robust foundation for drawing meaningful conclusions from the observed variations.

Algorithm Sophistication: Winning all episodes consistently often demands sophisticated algorithms, nuanced training approaches, and carefully designed architectures. Tweaking hyperparameters requires practical experience to achieve optimal results, and identifying training issues is a challenging task that necessitates intensive hands-on work.

Comparison with DeepMind's Approach: In a comparative context, it reveals that DeepMind's training of DQN for Atari 2600 games involve extensive experience totalling 50 million frames, a replay memory of 1 million most recent frames, and specific hyperparameter settings. Such benchmarks provide a reference point for the complexity and resources required in training sophisticated models.

Exploring Training Dynamics: For a deeper analysis of training dynamics and insights into weight updates in neural networks, it is crucial to examine Q-values and the gradient of the loss function. However, it's important to note that this aspect was beyond the scope of the current experimentation.

Hyperparameter Sensitivity: DQN training exhibits extreme sensitivity to hyperparameters and variations. While tweaking hyperparameters may positively impact certain aspects of training, it can simultaneously introduce negligible negative impacts on other facets. Maintaining a subtle equilibrium is a nuanced and continual challenge.

Future Directions:

The results of the experiments can be used for future investigations. Potential variations could include exploring different neural network types, experimenting with alternative loss functions, adopting diverse action selection strategies (e.g., Thompson sampling or Upper confidence bound (UCB) methods), and implementing learning rate annealing. Learning rate annealing, specifically, could enhance the fine-tuning of the learning process over time.

6 Chapter 6 - Safety Framework for ADAS Systems – Deep Neural Network Classifiers

6.1 Introduction

This chapter aims to provide a framework for mitigating safety risks associated with the integration of state-of-the-art deep neural network (DNN) classifiers within Advanced Driver Assistance System (ADAS) systems. Additionally, it proposes actionable measures to ensure compliance with automotive safety standards set by the International Organization for Standardization, including ISO 26262, Safety of the Intended Functionality (SOTIF) ISO 21448, and ISO Publicly Available Specification 8800.

It introduces an innovative comparative analysis, contrasting DNN classifiers with the obligatory prerequisites of automotive safety standards. The objective is to address the gaps and propose practical measures for enhancement.

After presenting a short overview of ADAS systems,5 levels of autonomous driving, and automotive safety standards such as ISO26262 and SOTIF 21448, a review of influential papers in this field is conducted.

Subsequently, a safety framework for DNN classifiers is introduced. It involves the identification of safety risks across the design, development, and implementation lifecycle, as well as the proposal of instrumental solutions for risk mitigation and compliance with the safety standards.

By presenting this comprehensive safety framework and proposing practical solutions, it contributes to a better estimation of unresolved remaining risks in the deployment of DNN solutions in ADAS systems.

6.2 Overview of Advanced Driver Assistance Systems (ADAS) in Automotive

Autonomous driving has become one of the most important topics of research and development in the automotive industry in recent years. Advanced Driver Assistance Systems (ADAS) are the foundation for autonomous vehicles and are designed to provide drivers with enhanced safety, comfort, and convenience, as well as reduce the overall risk of accidents. ADAS encompass a wide range of features, including lane keeping systems, adaptive cruise control, blind spot detection, emergency braking, obstacle detection, and predictive navigation [1].

The challenge is to develop ADAS technologies that are robust and reliable enough to safely handle complex driving scenarios [2].

For highly automated driving vehicles ADAS systems must integrate a large number of intricate sensing components, from radar to optical sensors such as camera and LIDAR sensors, with sophisticated algorithms in order to detect and respond to any potential road hazards. This makes the development of ADAS systems challenging for OEMs (Original Equipment Manufacturers) [2].

Furthermore, the development and deployment of ADAS systems require an additional layer of design, namely the Operation Design Domain (ODD), which defines the operational context of the system [3] in terms of boundaries and constraints and its features.

The OEMs continue to face the challenge of developing reliable and cost-effective systems that accurately detect and respond to any potential dangers on the road. Despite the challenges, ADAS systems have become increasingly important to the automotive industry, as they are seen as a critical step towards achieving full autonomy. As the industry continues to focus on developing more sophisticated systems, it is expected that more advanced technologies, such as artificial intelligence and machine learning, will be incorporated into the design of these systems [3].

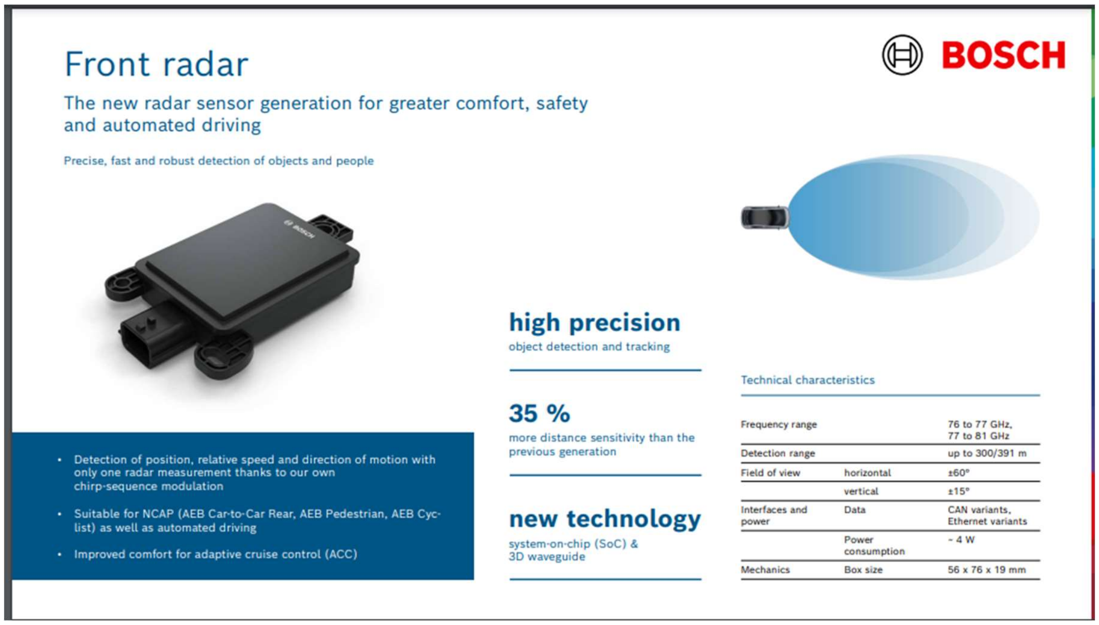
Car manufacturers continue to face intensive challenges to develop safe and cost-effective systems that can -in real time and under all circumstances- detect and respond to any potential dangers on the road. In addition to the technological challenges, OEMs must also contend with the various regulatory and safety requirements that must be met for ADAS systems to be used in production vehicles.

6.2.1 ADAS Sensor Technologies

6.2.1.1 Long-range Front or near-range Corner Radar

Radar sensors play a crucial role in Advanced Driver Assistance Systems (ADAS) and highly automated driving systems. These sensors use radio waves to detect the presence and location of objects and pedestrians around the vehicle within a long-range distance and an extensive field of view, ensuring comprehensive and precise object detection in the vehicle's surroundings. In the example provided in Figure 6.1, the radar sensor operates at a frequency range of 77 to 81 GHz, and exhibits an impressive detection range of up to 391 meters, coupled with a broad field of view covering +/-60 degrees horizontally and +/-15 degrees vertically.

Radar sensors in automotive are robust against many environmental factors, including adverse weather conditions, low visibility scenarios, and interference from other electronic devices. Radar sensors have extensive applicability in various ADAS functions, including Automatic Emergency Braking (AEB), Adaptive Cruise Control (ACC), Blind Spot Detection, and more.



Front radar

The new radar sensor generation for greater comfort, safety and automated driving

Precise, fast and robust detection of objects and people

- Detection of position, relative speed and direction of motion with only one radar measurement thanks to our own chirp-sequence modulation
- Suitable for NCAP (AEB Car-to-Car Rear, AEB Pedestrian, AEB Cyc-list) as well as automated driving
- Improved comfort for adaptive cruise control (ACC)

high precision
object detection and tracking

35 %
more distance sensitivity than the previous generation

new technology
system-on-chip (SoC) & 3D waveguide

Technical characteristics

Frequency range		76 to 77 GHz, 77 to 81 GHz
Detection range		up to 300/391 m
Field of view	horizontal	±60°
	vertical	±15°
Interfaces and power	Data	CAN variants, Ethernet variants
	Power consumption	~ 4 W
Mechanics	Box size	56 x 76 x 19 mm

Fig. 6.1: Technical Specification of a Long Range Radar ECU (source: https://www.bosch-mobility.com/)

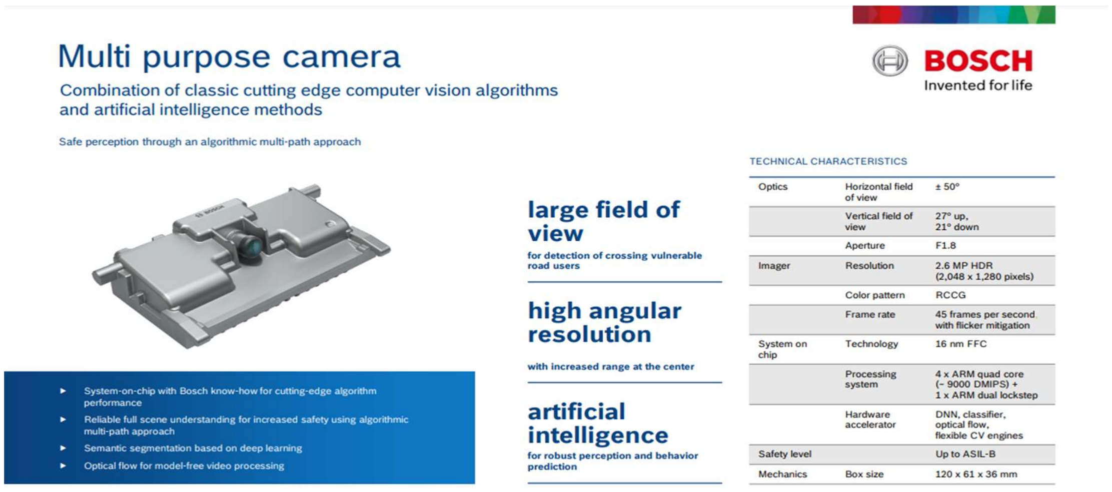
6.2.1.2 Camera sensors

Camera sensors are integral components in ADAS and highly automated driving systems, playing a crucial role in enhancing vehicle perception. These sensors employ optical technology to capture visual information, enabling the detection of stationary and moving objects, lane markings, traffic signs, and pedestrians in the vehicle's vicinity.

In the example depicted in Figure 6.2, the camera sensor features a field of view of +/-50 degrees horizontally and 21 to 27 degrees vertically, with a resolution of 2.6 megapixels. This high resolution allows for detailed and accurate image processing, contributing to the precision of object recognition and scene interpretation.

Operating within the visible spectrum, camera sensors excel in recognizing traffic signs, classifying objects, and capturing intricate details of the road environment.

Camera sensors have extensive applicability in various ADAS functions, including Automatic Emergency Braking (AEB), Adaptive Cruise Control (ACC), Lane Keeping Assist System (LKAS), Traffic Sign Recognition (TSR), and more



Multi purpose camera

Combination of classic cutting edge computer vision algorithms and artificial intelligence methods

Safe perception through an algorithmic multi-path approach

BOSCH
Invented for life

- System-on-chip with Bosch know-how for cutting-edge algorithm performance
- Reliable full scene understanding for increased safety using algorithmic multi-path approach
- Semantic segmentation based on deep learning
- Optical flow for model-free video processing

large field of view
for detection of crossing vulnerable road users

high angular resolution
with increased range at the center

artificial intelligence
for robust perception and behavior prediction

TECHNICAL CHARACTERISTICS

Optics	Horizontal field of view	± 50°
	Vertical field of view	27° up, 21° down
	Aperture	F1.8
Imager	Resolution	2.6 MP HDR (2,048 x 1,280 pixels)
	Color pattern	RCCG
	Frame rate	45 frames per second with flicker mitigation
System on chip	Technology	16 nm FFC
	Processing system	4 x ARM quad core (~ 9000 DMIPS) + 1 x ARM dual lockstep
	Hardware accelerator	DNN, classifier, optical flow, flexible CV engines
Safety level		Up to ASIL-B
Mechanics	Box size	120 x 61 x 36 mm

Fig. 6.2: Technical Specification of a Multi-Purpose Camera ECU (source: https://www.bosch-mobility.com/)

6.2.1.3 LIDAR

LIDAR (Light Detection and Ranging) sensors are integral components in ADAS Systems, contributing to enhanced vehicle perception and safety. Operating on the principle of emitting laser beams and measuring their reflections, LIDAR sensors accurately map the surrounding environment in three dimensions.

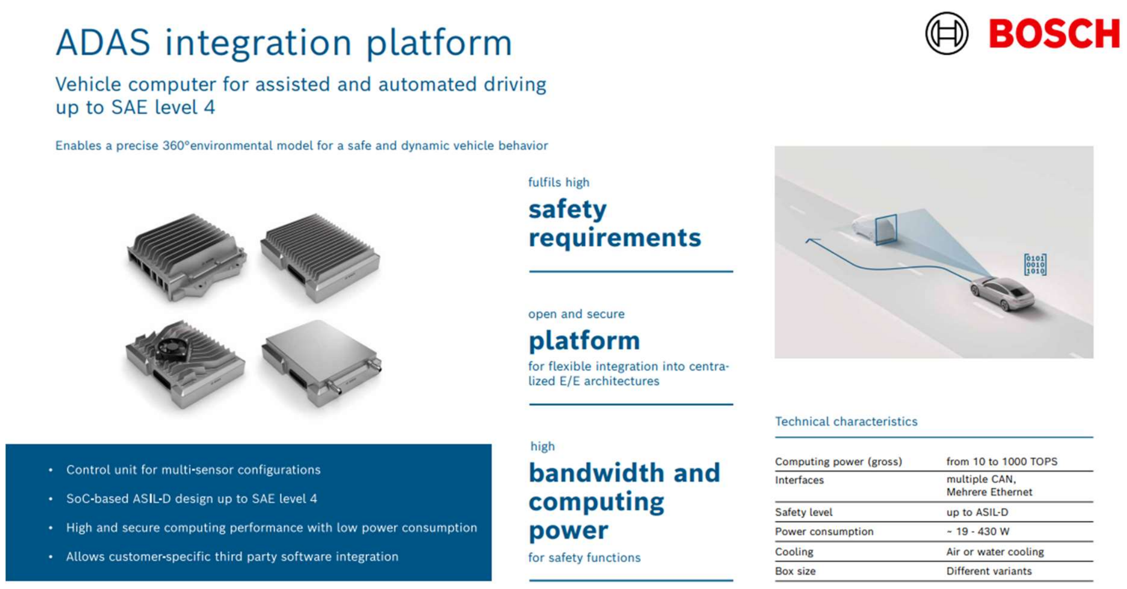
6.2.1.4 ADAS Fusion ECU

The ADAS Fusion Electronic Control Unit (ECU) is at the forefront of advanced automotive technology, integrating data from various sensor types through sensor fusion. Through sensor fusion, this ECU combines information from diverse sensors, employing sophisticated software algorithms to create a highly comprehensive and accurate environmental model. By merging inputs from cameras, radar, LIDAR, and other sensors, and utilizing powerful algorithms to handle this big amount of data the ADAS Fusion ECU generates a comprehensive and accurate representation of the vehicle's surroundings.

Additionally, the redundancy provided by multiple sensors contributes to an extra layer of reliability. For instance, in scenarios of impaired visibility, where cameras may falter, the ADAS fusion ECU leverages data from other sensor types to compensate for these weaknesses.

This redundancy is a critical feature that further solidifies the ADAS Fusion ECU's role in advancing the safety and efficiency of modern automotive technologies.

The ability of the ADAS fusion ECU to access data from various sensors ensures a continuous and reliable flow of information and significantly enhances the safety of ADAS and autonomous driving systems. Figure 6.3 presents an example of ADAS fusion ECU system.


ADAS integration platform

BOSCH

Vehicle computer for assisted and automated driving up to SAE level 4

Enables a precise 360°environmental model for a safe and dynamic vehicle behavior

- Control unit for multi-sensor configurations
- SoC-based ASIL-D design up to SAE level 4
- High and secure computing performance with low power consumption
- Allows customer-specific third party software integration

fulfils high **safety requirements**

open and secure **platform** for flexible integration into centralized E/E architectures

high **bandwidth and computing power** for safety functions

Technical characteristics

Computing power (gross)	from 10 to 1000 TOPS
Interfaces	multiple CAN, Mehrere Ethernet
Safety level	up to ASIL-D
Power consumption	~ 19 - 430 W
Cooling	Air or water cooling
Box size	Different variants

Fig. 6.3: Technical Specification of a Central ADAS fusion ECU (source: https://www.bosch-mobility.com/)

6.2.2 ADAS Functions

ADAS systems offer a wide range of functions contributing to the comfort and safety of driving in modern automotive technology. In this section, three key functions—Automatic Emergency Braking (AEB), Adaptive Cruise Control (ACC), and Lane Keeping Assist System (LKAS)—are briefly explained.

<u>Automatic Emergency Braking (AEB)</u>

Automatic Emergency Braking (AEB) is a sophisticated driver assistance feature aimed at elevating vehicle safety. This technology employs a network of sensors, such as radar and cameras, to monitor the surroundings for potential collision threats. When a potential collision is detected, AEB autonomously applies the brakes, mitigating or even preventing the impact. This vital Advanced Driver Assistance System (ADAS) significantly enhances the overall safety of the vehicle by providing an additional layer of protection against potential accidents.

<u>Adaptive Cruise Control (ACC)</u>

Adaptive Cruise Control (ACC) is a cutting-edge technology within the realm of ADAS that optimizes driving comfort and safety. Using a combination of sensors, including radar and cameras, ACC maintains a set speed while also adjusting the vehicle's speed based on the distance to the vehicle ahead. This intelligent system not only enhances convenience during highway driving but also contributes to overall road safety by automatically adjusting the vehicle's speed to maintain a safe following distance.

<u>Lane Keeping Assistant (LKAS)</u>

Lane Keeping Assist System (LKAS) is an advanced driver assistance technology designed to enhance vehicle safety and stability. Utilizing a combination of cameras and other Advanced Driver Assistance Systems (ADAS) sensors, LKAS is engineered to detect lane markings on the road. Once engaged, this system actively intervenes by steering the vehicle, ensuring it remains cantered within the designated lane. LKAS is considered a function of highly autonomous driving at AD Level 3 contributing to the safety of automated driving.

6.2.2.1 Summary of Safety Critical Hazards

Table 6.1 presents a list of safety hazards associated with ADAS functions: AEB, ACC, and LKAS.[18]

ADAS Function	Hazard
AEB	Unintended Braking in non-critical situations
	Intended but too weak/strong/late deceleration
	Undetected obstacle in the vehicle's path
	Complete or partial Loss of braking power
ACC	Unintended activation of the function
	Too high or low vehicle deceleration/ acceleration
	Unintended modification of set speed
	Not taking over driver’s command to override the set speed
LKAS	Oversteering
	Understeering
	Misdetection or non-detection of lane markings
	Unintended deactivation of the function

Table 6.1: A list of safety hazards with respect to ADAS functions: AEB, ACC, and LKAS

6.2.3 Machine learning solution in ADAS systems

In recent years, there has been a significant surge in the adoption of advanced technologies, including machine learning (ML) and artificial intelligence (AI), within the automotive industry [11]. While these advancements hold great promise, their integration necessitates a careful consideration of safety concerns. The application of machine learning in ADAS systems, encompassing perception and computer vision, has transformed the vehicle's awareness of its surroundings. This section explores the evolving landscape of machine learning solutions in ADAS, shedding light on their potential benefits and the imperative need for a nuanced approach to safety within this rapidly advancing technological domain.

Machine learning (ML) and artificial intelligence (AI) can significantly increase road safety and traffic productivity through the exploitation of advanced driver assistance systems (ADAS). These technologies enable vehicles to train from vast amounts of data and make complex decisions in real time, allowing them to expect and respond to various driving scenarios more adequately. However, the integration of ML and AI also introduces new safety concerns and challenges that must be addressed carefully. A deep understanding of the possible risks and strict adherence to safety standards is essential for ensuring the safe and reliable operation of such systems.

Researchers and practitioners are working on new safety standards that specifically address how ML and AI will be utilized in the automotive industry in order to overcome these challenges.

[18] This information is provided for general reference and is not intended to be exhaustive or comprehensive.

6.3 SAE Levels of Autonomous Driving

ADAS systems are typically classified into different levels according to the Society of Automotive Engineers (SAE) International level of automation scale [4]. These levels range from level 0, which is no automation, to level 5, which is full automation. Automation levels 1 and 2 typically involve driver assistance systems such as Adaptive Cruise Control or Lane Keeping Assist, while levels 3 and 4 are able to take over most of the driving tasks and involve partial automation, and level 5 represents full automation. See Table 6.2.

Level 0	No Automation The driver is completely in control of the vehicle and all its functions.
Level 1	Driver Assistance Certain functions of the vehicle are automated in specific situations, such as adaptive cruise control and lane centring.
Level 2	Partial Automation vehicle executes acceleration and braking, while the human driver is responsible for steering.
Level 3	Conditional Automation Vehicle is capable of performing all dynamic driving tasks, but the human driver must be ready to take control when the system requests it.
Level 4	High Automation Vehicle performs all dynamic driving tasks without the human driver. It will warn the driver when it nears its operational limits. If the driver does not react, then it brings the vehicle to a safe condition.
Level 5	Full Driving Automation Vehicle monitors the driving surroundings without the human driver and performs all dynamic driving tasks autonomously.

Table 6.2: SAE's Autonomous Driving Levels

The automotive industry is quickly moving towards level 5 automation, and it is expected that by 2030, the automotive industry will move towards level 4 and 5 automations [5].

6.4 Driving Scenarios

To gain a deeper understanding of the diverse challenges that autonomous driving and ADAS system may face, some of the driving scenarios are elaborated here:

Highway: highway driving consists of long stretches of well-maintained roads with high-speed traffic. Using long-range sensors (e.g., Radar), vehicles in this environment must maintain a constant high speed, change lanes, and respond to occasional traffic variations such as slowing down or overtaking vehicles.

Parking Lot: Parking lot scenarios include manoeuvring at low speed through tight spaces, avoiding obstacles, and precise parking using high-precision and near-range sensors.

Off-Road: Off-road driving means driving on rough and unpaved roads covered with dirt or mud. ADAS systems must adapt to these unpredictable conditions to offer robust off-road vehicle control and obstacle detection.

Stop and Go: Complex traffic patterns, frequent stops, and variable speeds characterize city driving scenarios. ADAS systems must manage congestion, navigate tight streets, and handle stop-and-go traffic, ensuring passenger safety and efficient navigation.

Intersections and Roundabouts: Roundabouts are circular intersections where vehicles must yield and merge. In navigating roundabouts, ADAS systems rely on both front sensors as well as side and rear sensors to detect and respond to colliding traffic and to ensure safe navigation.

6.5 Automotive Safety Standards

In the automotive industry, safety is of utmost importance. To ensure the safety of road users and pedestrians, the International Organization for Standardization (ISO) and the Special Interest Group for automotive safety have released safety standards ISO26262 and ISO21448-SOTIF- [6].

These standards have been in place for a while now but they have evolved over time to take into account the ever-changing demands of the automotive industry.

6.5.1 International Organization for Standardization (ISO) 26262

ISO 26262 is a normative guideline that provides a framework for the entire development process of electrical and/or electronic systems that are installed in road vehicles. It outlines specific approaches to identify and assess hazards related to systematic failures as well as random Hardware (HW) malfunctions, and how to mitigate the impact of the risks within the framework of the product lifecycle. This includes the design, implementation, validation, and verification of safety-related systems. ISO 26262 offers a comprehensive approach to the development of safety-related systems, helping engineers to identify and reduce risks throughout the entire development process.

6.5.2 SOTIF 21448

The ISO 21448 - Road vehicles — Safety of the intended functionality (SOTIF) standard is a newer automotive safety standard and was released in 2019 as an extension of ISO26262, aiming to address risks that are not covered by the original standard. It provides guidance on how to ensure the safety of road vehicles when they perform their intended functions without faults [7]. The standard focuses on preventing unreasonable risks that may arise from functional limitations of the intended functions or performance insufficiency of the system or from predictable misuse by people or any environmental influences [8].

Some key aspects of the SOTIF standard are:

- The definition of intended function and malfunction.
- The identification and assessment of SOTIF scenarios and risks.
- The establishment of requirements and validation strategies for SOTIF.
- The documentation and traceability of the SOTIF process.

SOTIF validation strategy considers two types of hazards: known area hazards and unknown area hazards. Known area refers to the set of scenarios where the system's behaviour is well-defined and predictable. Unknown hazard area refers to the set of scenarios where the system's behaviour is uncertain or unexpected due to unforeseen situations or limitations [9].

For example, a lane keep assist function may work well in a known area where the road markings are clear and visible, but it may fail in an unknown hazard area where the road markings are faded or covered by snow [9]

SOTIF validation aims to identify and mitigate potential hazards in both known and unknown areas by applying various methods such as functional specification, functional hazard analysis, risk assessment, verification and validation [10]

6.5.3 ISO PAS 8800

<u>Inadequacy of ISO26262 and SOTIF standards for Automated driving and ML solutions</u>

The ISO 26262 and 21448 (SOTIF) standards have been widely adopted in the automotive industry for developing safety-critical systems. However, when it comes to the development of Machine Learning (ML) solutions and autonomous driving, these standards are inadequate.

This inadequacy is due to the fact that these standards are designed for deterministic systems and not for ML applications with stochastic behaviour [12]. Furthermore, due to the unpredictable nature of ML and autonomous driving, it is difficult to ensure safety in these systems using the existing safety standards [13]. Thus, it is important to develop safety standards that are specific to ML and autonomous driving.

ISO PAS 8800 Road Vehicles – Safety and Artificial Intelligence [14] -drafted in 2022- is a publicly available specification for the safety assurance of automated driving systems, providing compliance guidance on various aspects of safety assurance for automated driving systems. These include safety requirements, safety management, validation, and verification methods. It is intended to be used in combination with existing safety standards, such as ISO 26262 and SOTIF, to ensure the safety and reliability of automated driving systems.

Overall, the combination of these safety standards is essential to ensure the safety and reliability of ML solutions in the automotive industry. They provide a common framework for the development process of ML solutions, as well as guidance on how to ensure safety and reliability in the development of autonomous driving systems.

ISO PAS 8800 is a proposed standard for road vehicles that define safety-related properties and risk factors impacting the insufficient performance and malfunctioning behaviour of artificial intelligence (AI) within a road vehicle context [14]. It is currently under development by ISO/TC 22/SC 32/WG 14 "Safety and Artificial Intelligence" and is expected to be published in 2023[15].

The motivation for developing ISO PAS 8800 is to address the challenges and gaps that arise from applying AI systems in road vehicles, such as perception, decision-making, learning, adaptation, etc.

ISO PAS 8800 aims to establish a common approach for developing and testing AI systems that are used in road vehicles [16]. It will cover topics such as:

- Derivation of Safety requirements for AI systems
- Conducting Safety analysis methods for AI systems
- Verification and validation methods for AI systems
- Safety assurance cases for AI systems
- Safety management processes for AI systems

6.6 Safety Framework for Deep Neural Network (DNN) Classifiers

Deep neural networks (DNNs) are a type of machine learning model that can be used to solve a variety of tasks, including object classification. In recent years, the application of DNN classifiers has extended to ADAS functions which are an integral component of Autonomous Driving systems. DNN enable ADAS functions to not only detect but also classify the objects in the vehicle's vicinity. Accurate object classification is necessary for the system to respond appropriately to potential hazards, providing timely warnings or taking corrective actions. If the autonomous system inaccurately classifies the drivability of an object or fails to recognize road markings, it can pose serious safety risks and jeopardizes the lives of people.

The importance of accurate classification in LKA systems is illustrated in figure 6.4.

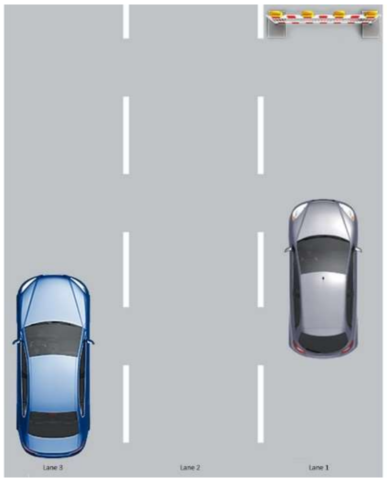


Fig. 6.4: The significance of accurate object classification for ADAS Lane-Keeping Assistant systems

The ego vehicle is situated in lane 1, several hundred meters ahead lies a stationary road closure obstacle. Lane 2 is currently unoccupied, while in lane 3, there is an approaching vehicle. If the system misclassifies the drivability of these objects or fails to recognize the road markings, it could result in incorrect lane-keeping decisions, such as unnecessary lane changes or abrupt braking, potentially leading to hazardous situations. Hence, in this particular driving context, the precision of object classification, coupled with road marking identification, remains essential for the Lane Keeping Assist system (LKAS) to operate safely and efficiently in the presence of both stationary and moving obstacles.

Before delving into the specifics of our proposed safety framework for deep learning classifiers, an initial step involves reviewing influential papers in this field.

6.6.1 Review of related works

Authors in [17] represent how crucial it is to develop methods for quantifying the risks related to deep neural networks, especially as they become more prevalent in safety-critical applications, such as medical diagnosis systems and autonomous vehicles. They defined a new class of risk metric called “uncertainty example” based on a probabilistic modelling approach and developed a framework which allows quantification of both the likelihood and severity of safety-critical metrics in a computationally effective algorithm. They evaluated the framework on several image classification tasks and demonstrated its effectiveness in identifying safety risks associated with specific neural network architectures and training procedures. They also demonstrate how the framework can be used to guide the design of more robust and reliable neural network systems. Their work has made a significant contribution to quantifying safety risk metrics such as robustness, reachability, and uncertainty metrics in DNNs. However, we believe that Quantitative risk assessment techniques are often difficult to apply to deep neural networks (DNNs) due to their complex architectures and a large variety of implementation algorithms. Therefore, a quantitative computation of safety risk metrics of such DNN networks with a high number of layers seems to be practically not feasible. Estimating the safety risks of these networks can be done more effectively by analysing the performance of the classifier through the metrics of false positives, true positives, false negatives, and true negatives. Performance measurement can give a repetitive way of examining the exactness of a classifier and spotting likely hazards in an economical and immediate fashion.

The DDE process, proposed in the paper [18] offers a systematic V-Model development process solution to ensure the quality and composition of data sets used for ML. This process is compliant with the System Processes in Automotive Engineering (ASPICE) standards, making it easy to integrate into existing development processes and gain acceptance in automotive engineering. The motivation of the authors to propose the DDE process is to address the challenges of developing high-quality machine learning (ML)-based systems in the industry. The authors recognize that the quality of ML-based systems depends on the quality of the data used for training, verification and validation tests. They proposed the DDE process as a systematic and structured approach to ensure that the generated data sets are of high quality and meet the requirements of the operational design domain (ODD). Despite its advantages, the DDE process does not cover other aspects of ML such as model selection or hyperparameter tuning and does not offer a comprehensive approach to functional safety.

The paper [19] provides an overview of available methods for supporting the safety argumentation of machine learning solutions in safety-critical systems in accordance with the ISO26262 safety standard. It identifies open challenges in this area and argues that the development and certification of safety-critical software using machine learning (ML) is different from traditional approaches. Since ML models are data-driven and automated their design, verification, and validation require new methods. To address this, the authors suggest that ISO 26262-part 6 processes for software development can be applied, and the main focus must be placed on the requirement engineering, development, verification, and validation parts. Regarding requirement engineering, the authors emphasize that the incorporation of available expert knowledge and experience into the formulation of use-case, system, and function requirements should be expressed in specialized key performance indicators (KPIs). To provide a proper safety argumentation for the design and development part, they recommend that domain experts should reason all general design choices -which are not specified in the requirements- related to NNs model design and the training objectives. To enhance the robustness of the design, authors believe that measures such as regularization and training data preparation might be particularly useful.

The authors in [20] contend that conventional neural networks are generally trained based on the principle of minimizing the training error without taking into account the

impact of outliers or misclassified samples. This can lead to high risk in real-world applications, where the cost of misclassification can be noteworthy. To address this issue, the authors propose a new training algorithm that speculates both the training error and the risk associated with each sample. The proposed algorithm involves two steps: first, the neural network is trained using the conventional approach to minimize the training error. Then, the risk of misclassification for each sample is calculated and used to adjust the weights of the neural network. This ensures that the network is less likely to misclassify samples that are associated with higher risk. The authors [20] evaluate the proposed algorithm on several datasets and demonstrate that it outperforms conventional neural networks in terms of risk reduction. The algorithm has proven to be robust against various types of noise and outliers in the data. Overall, the paper presents a novel approach to pointing risk in neural network classifiers, which can have remarkable inferences for real-world applications where the cost of misclassification is high.

6.6.2 Risk Factors across Key Steps in DNN Classifier Development

This work introduces an illustration highlighting immediate risk factors and potentially risky design variations in key stages of DNN design, training, and the implementation of DNN classifiers within ADAS systems. The identified risks originate from direct hands-on experiences in the domain, contributing to a distinctive and practical perspective. See Figure 6.5.

Risk factors are marked in red and design variations are marked in blue.

It is crucial to emphasize that the careful selection of these design variations is essential to minimize the risk of frequent misclassifications in real-world applications.

The subsequent sections will elaborate on the identified risks and propose methods and solutions to mitigate the risk and comply with mandatory requirements of safety standards.

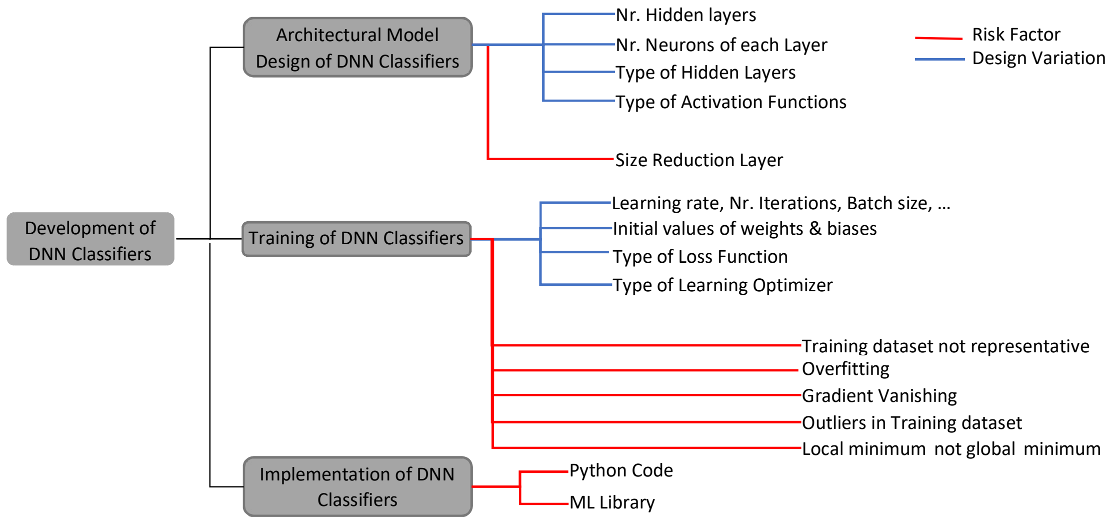

Development of DNN Classifiers
Architectural Model Design of DNN Classifiers
Nr. Hidden layers
Nr. Neurons of each Layer
Type of Hidden Layers
Type of Activation Functions
Size Reduction Layer
Risk Factor
Design Variation
Training of DNN Classifiers
Learning rate, Nr. Iterations, Batch size, …
Initial values of weights & biases
Type of Loss Function
Type of Learning Optimizer
Training dataset not representative
Overfitting
Gradient Vanishing
Outliers in Training dataset
Local minimum not global minimum
Implementation of DNN Classifiers
Python Code
ML Library

Fig. 6.5: Variations and Risk Factors across Key Steps in DNN Classifier Development

6.6.3 Architectural Model Design of Deep Neural Network Classifiers

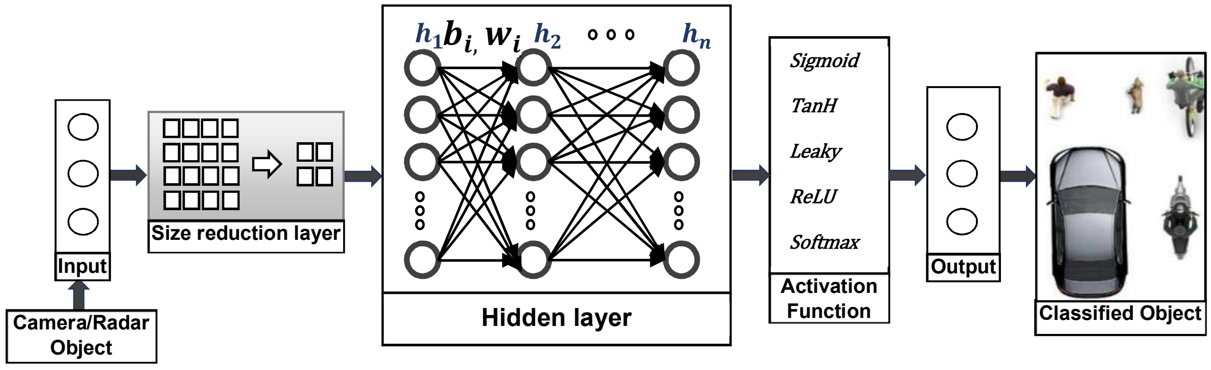


Fig. 6.6: Architectural components of a Deep Neural Network Classifiers for ADAS functions

An example of a neural network classifier consists of five functional layers: the input, the hidden layer, the activation function, and the output layer is shown in Figure 6.6. A DNN classifier may also utilize a size reduction layer, max pooling, and other elements.

In a practical implementation for ADAS ECUs, such as Radar ECU or Video ECU, the data acquisition process gathers image data from a camera or extracting object information (distance, azimuth angle, and radial velocity) from a radar ECU. After processing raw data and conducting related digital signal processing, the software's perception algorithms further refine the acquired information, preparing it for the subsequent classification task.

Neural Network Model:
The network design and architecture should be carefully chosen so that the model does not over fit or under fit the data. It is critical to note that, as none of the approaches offers an ideal solution, DNN architects have to trade-off between the robustness and accuracy of their model and the implementation overheads. In the context of autonomous driving, ConvNets, with their established success in image classification, remain a robust choice. Therefore, the selection of the most appropriate design and architecture largely depends on the specific application and the resources available.

Input Layer:
The input layer in an ADAS system is responsible for taking in sensory data from cameras, Lidar, or radar subsystems [21]. In the case of a camera image, the input layer consists of pixels that are converted into grayscale values. The input layer can also be used to process radar objects, such as obstacles or other vehicles. In this case, the input layer takes the raw data from the radar and performs normalization and feature extraction (the pre-processing steps).

Size Reduction Layer:
The design decisions of DNN architect at this stage include choosing an appropriate image resolution, colour space, and data format. For radar data, the design decisions include choosing the appropriate data format and pre-processing

techniques. If the input layer is too small, the model may not be able to feed sufficient information to succeeding layers leading to misclassification. On the other hand, if the input layer is too large, the model may overfit the training data, resulting in poor classification of new data. The key aspects of the size reduction layer are compression, resizing, cropping, and scaling. Compression reduces the size of a file by removing redundant or unnecessary data.
Resizing changes, the dimensions of an image, allowing for changes in the aspect ratio.
Cropping reduces the size of an image by removing certain segments. Scaling adjusts the size of an image without changing its aspect ratio. While size reduction layers can reduce the size of a file, they can also reduce the accuracy and quality of the image. Compression, in particular, can reduce the detail and clarity of the image. Additionally, resizing and cropping can cause distortion or blurriness. On the other hand, scaling can be used to increase the size of an image without sacrificing accuracy. It is important to carefully consider the accuracy and computational requirements when selecting a size reduction layer.

Hidden Layer:
The hidden layers are responsible for transforming the input data into a more meaningful representation. FC layers are prone to overfitting, as they can easily learn redundant features. Additionally, they often require a lot of parameters to model the data, which can lead to longer training times. Convolutional layers are effective for image processing tasks, as they can easily extract features from images. They are also more efficient than FC layers, as they share weights across the input, reducing the number of parameters that need to be tuned. However, convolutional layers are not efficient enough to model relationships between input and output when they are highly complex. This would be a weakness for safety-critical tasks. Additionally, they can require significant amounts of training data to learn useful features.

Activation Function:
There are a variety of activation functions available for use in neural networks, each with its own strengths and weaknesses. ReLU, Leaky ReLU, sigmoid, Tanh and softmax are the most widely used activation functions.
ReLU (Rectified Linear Unit) is one of the most widely used activation functions in deep learning. ReLU has the advantage of being non-saturating and has a non-zero gradient, which allows for faster training of the network. However, it can suffer from the “dying ReLU” problem where the neuron’s output is 0 and can no longer be trained [22].

Leaky ReLU (LReLU) is an extension of ReLU that introduces a negative slope on the left side of the activation function to alleviate the dying ReLU problem
by allowing for a small positive gradient for negative inputs. However, it can suffer from
noisy gradients and can be difficult to tune [23].

Sigmoid is another popular activation function that has the advantage of introducing non-linearity into the network and can be used to approximate an arbitrary function. However, it can suffer from the “vanishing gradient problem” where the gradients become very small, and the network is unable to learn [24].
Tanh (Hyperbolic Tangent) is similar to the sigmoid function with a single difference in
that its output ranges between (-1,1). It is smooth and non-linear and has the advantage of allowing for faster training compared to the sigmoid due to its centred output. However, it can suffer from the same vanishing gradient problem as the sigmoid [23].

In this section, the construction components of a typical neural network model have been introduced, design diversity has been highlighted, and their strengths, weaknesses, and safety risks have been discussed. The subsequent focus will be on proposing architectural solutions to minimize risks and presenting systematic approaches for compliance with automotive safety standards.

Architectural Solutions

Plausibility checks, Degradation strategy, Fusion:
Architects of DNNs can reduce the risk of wrong classifications in their architecture design by incorporating plausibility checks, degradation, and fusion strategies. Plausibility checks can help detect outliers and false positives, while degradation strategies are needed to ensure that the model still performs safely in the face of various levels of data corruption or perturbations.
Fusion of radar and camera sensors, i.e., associating radar and camera information [25] in an Advanced Driver Assistance System (ADAS) decomposes the safety load between available perception sensors and hence can drastically reduce the risk of misclassification. A comprehensive guideline for data processing and fusion methodologies for autonomous driving has been proposed in [26]. By combining the data provided by redundant sensors, the system can take advantage of the complementary information provided by each to reduce the uncertainty of any given classification and better identify objects in the environment.

Software safety analysis, Review, Comprehensive documentation of design decisions:
DNN models are highly complex algorithms, not intuitive for humans, and their output may not be readily interpretable. Software safety analysis ISO 26262 is the recommended method and aims to identify and mitigate potential hazards and risks in software systems. These methods can be applied in a Hazard and Operability (HAZOP) based cause-effect relationship by identifying the root causes of potential safety-critical issues arising from these algorithms and implementing measures to prevent or mitigate their effects. By analysing the potential causes of software failures and their consequences, safety analysts can develop strategies to improve the safety and reliability of software systems.
Comprehensive documentation of detailed design decisions is a crucial step when designing DNN models for autonomous driving to ensure that the model is

reproducible and that its performance can be accurately assessed and tracked. This documentation should include all decisions made during the design process of the model architecture, the dataset used, the hyperparameters, and the training procedure

6.6.4 Training of Neural network

The training of Deep Neural Network classifiers involves a process known as supervised learning, where labelled data are used to train the model using a training dataset. During this process, the objective is to determine the weighting factors and biases of the network by iteratively adjusting them to minimize the loss function.

In the training phase the available labelled data is divided into two parts: the training dataset and the test dataset.

The process involves systematically adjusting the network hyperparameters and training the model on a training dataset to find the best configuration that minimizes the loss function. These hyperparameters include learning rates, batch sizes, and number of learning iterations, among others.

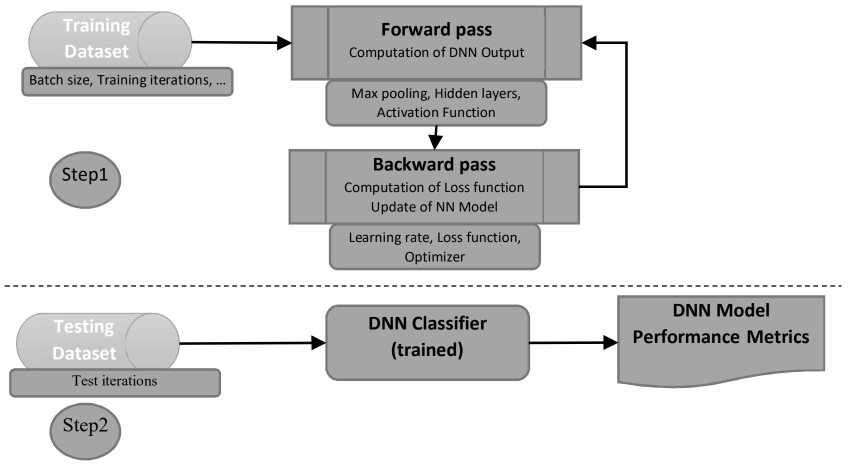

Training Dataset
Batch size, Training iterations, ...
Forward pass
Computation of DNN Output
Max pooling, Hidden layers, Activation Function
Backward pass
Computation of Loss function
Update of NN Model
Learning rate, Loss function, Optimizer
Step1
Testing Dataset
Test iterations
DNN Classifier (trained)
DNN Model Performance Metrics
Step2

Fig. 6.7: The training process of a DNN network

Figure 6.7 shows the two-phase process of training Deep Neural Network (DNN) classifiers. The first phase is the training phase, where a forward and a backward pass are implemented. The forward pass processes input data to make predictions, while the backward pass (backpropagation) adjusts network weights and biases to minimize the prediction error, aligning them with the actual output.

The second phase involves testing the trained model on unseen data and assessing its performance by measuring the accuracy of its predictions.

In the backpropagation phase, factors such as the learning rate, choice of loss function, and optimizer are crucial. Now, the focus shifts to the exploration of design variations and potential risks associated with these key training components.

<u>The Learning rate:</u>

The learning rate is one of the most important indicators of the performance and stability of DNN. It has a direct relation with the ability of the Neural Network to learn from the data and converge to an optimal solution. Learning rates determine the size of the steps taken during the optimization process, as the model iteratively updates its weights in order to minimize the loss function. A well-chosen learning rate ensures that the model converges efficiently and effectively, without overshooting the optimal solution or becoming stuck at local minima [27].

The choice of an appropriate learning rate is essential to strike the right balance between the speed of convergence and the accuracy of the model. If the model oscillates wildly around the optimal solution, a high learning rate may result in unstable training and potentially poor performance. A low learning rate may, in contrast, cause the model to converge very slowly, consuming considerable computational resources and increasing the risk of over-fitting as the model spends more time training on the same data. Therefore, DNN architects must carefully choose and tune the learning rate based on the specific

application and available resources, often employing techniques such as learning rate annealing or adaptive learning rate algorithms to optimize its value throughout the training process [28].

The Optimizer:

Optimizers are responsible for adjusting the weights and biases of the network during the training process to minimize the loss function. Different types of optimizers exist, each with its own strengths and weaknesses. The most commonly used optimizers include Gradient Descent, Stochastic Gradient Descent (SGD), AdaGrad, RMSProp, and Adam [29].

Gradient Descent is an optimization algorithm in which the network parameters are updated based on the steepest gradient of the loss function. In spite of its ability to converge to the global minimum, this method is computationally expensive and slow, especially when dealing with large datasets. Unlike Gradient Descent, the Stochastic Gradient Descent (SGD) optimization approach is based on a subset of the dataset randomly selected, which speeds up the training process. However, it can be less stable and converge to the global minimum with more fluctuations.

The AdaGrad is an adaptive learning rate optimizer that assigns individual learning rates to each parameter. It accelerates convergence in cases where the data is sparse or has varying scales, but it can lead to premature convergence due to its aggressive learning rate decay. The RMSProp addresses the limitations of AdaGrad. It utilizes a moving average of squared gradients to adjust the learning rate, which prevents the aggressive decay of the learning rate and leads to better convergence properties. The Adam optimizer maintains separate moving averages for the gradients and squared gradients, which allows for adaptive learning rates and momentum. Adam is known for its fast convergence and robustness to different types of datasets and architectures. The choice of an optimizer is essential for the performance and efficiency of a DNN. Each optimizer has its own set of advantages and disadvantages, and the choice depends on the specific application, dataset, and resources available. By selecting an appropriate optimizer, DNN architects can strike a balance between model robustness, accuracy, and implementation overheads.

The Loss function:

The loss function serves as a measure of how well the model performs during the training phase. It provides a quantitative assessment of the discrepancy between predicted and true outcomes. This assessment helps determine the most suitable design and architecture. It allows DNN architects to fine-tune the model, and balance the trade-offs between robustness, accuracy, and implementation overheads.

A loss function can take a variety of forms when applied to DNNs, catering to different types of problems and objectives. Some commonly used loss functions are Mean Squared Error (MSE) for regression tasks, Cross-Entropy Loss for classification problems, and Hinge Loss for support vector machines, among others. It is essential to carefully choose the appropriate loss function for the specific application, as it directly influences the model's learning process and the optimization algorithm's behaviour [30].

The role of the loss function goes beyond simply calculating a model's performance. During the training phase, it acts as a guide for the optimization algorithm, helping it adjust the model's parameters to minimize error. In essence, the optimization process seeks to reduce the loss function's value by locating the model's optimal configuration in the parameter space. This is typically done using gradient-based methods like Stochastic Gradient Descent (SGD) or advanced variants such as Adam and RMSProp [31].

Moreover, the loss function contributes to the prevention of overfitting or underfitting, which are common challenges in DNNs. By incorporating regularization terms, the loss function can penalize complex models that overfit the data or prevent models that are too simplistic and underfit the data. This enables the model to achieve a balance between fitting the training data and generalizing it to new, unseen data.

Some of known training challenges:
Deep learning training involves several significant challenges, as illustrated in Figure 6.5. These issues encompass unrepresentative training data, making it difficult to accurately detect all classes. Gradient vanishing can hinder the model's weight updates, while overfitting arises when models capture noise rather than genuine data patterns. The presence of outliers in training datasets can lead to classification inaccuracies, and there's the risk of training converging to local minimums instead of reaching global optima. Addressing these known issues is crucial to ensure the robustness and generalization of deep learning models.

6.6.5 Implementation and Integration of DNN Classifiers

Figure 6.8 illustrates how deep neural network (DNN) models are typically implemented and deployed. Machine-learning developers use open-source machine-learning libraries such as TensorFlow and PyTorch in their code to implement DNN models.

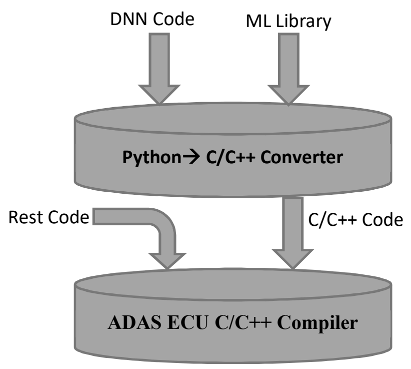


Fig. 6.8: Implementation of DNN in python and conversion to C/C++

Whilst C and C++ are considered to be industry-standard, Python has become increasingly popular for the development of DNN models in automotive applications. To manage the coding complexity, ISO 26262 has several normative orders such as having a specific coding guideline, avoiding dynamic memory allocation, using static analysis tools, and following a consistent coding style. Coding complexity can be measured using metrics such as cyclomatic complexity, nesting depth, and the number of parameters [32]. Furthermore, advanced-level programming features such as data structures, concurrency, polymorphism, global variables, and exception handling must also be regulated to ensure high-quality code. By adhering to these coding standards, developers can ensure their code is reliable, robust, and efficient. Despite the fact that machine-learning libraries are widely used in automotive engineering, it must be noted that they are general-purpose libraries and are not designed to comply with ISO 26262 standard.

6.6.6 Verification and Validation Methods for DNN Classifiers

Verification and validation are essential processes in developing deep neural networks (DNNs) for autonomous driving. They help to identify errors in DNN models, bugs in their implementation, or critical performance insufficiencies.

The Verification process ensures that the DNN model is correctly implemented and meets the design specifications. The Validation process on the other hand ensures that the DNN model is accurate and reliable in its predictions.

To ensure that DNNs are implemented and functioning correctly, several V&V techniques are proposed. These include unit testing, endurance run tests, scenario-based testing, fault injection with white noise on sensor data, adversarial attacks, and FP/FN and TP/TN metrics benchmarking.

Unit Testing:

Unit design testing is an essential step for verifying the functioning of individual components of a DNN model. By breaking down the system into smaller units, engineers can thoroughly test each one separately and guarantee its performance according to the specifications. Hardware-in-the-Loop (HIL) simulation is a great way to conduct automated and cost-effective tests that are reproducible and can detect any potential issues before deployment in the real world.

Endurance Run Testing:

This is one of the most common V&V techniques applied to automotive systems. In this type of testing, a DNN model is tested for its ability to process sensor data in a wide range of conditions over a long period of time. Endurance run tests help to validate the robustness of the model and its performance under various conditions.

Scenario-based Testing:

Scenario-based testing focuses on exploring the behaviour and functionality of a DNN model when presented with specific scenarios that an autonomous vehicle may encounter in the real world. The model response under these scenarios is analysed to assess its ability to detect overridable and non-overridable obstacles and take appropriate decisions. This helps to guarantee that the model is able to handle unforeseen circumstances in a safe manner, while accurately detecting and responding to obstacles and other objects in its environment.

Fault Injection with White Noise on Sensor Data:

This is a technique used to evaluate the performance of a DNN model when it receives noisy or corrupted sensor data. In this type of testing, noise is injected into the system and the model robustness is tested to see how sensitive it is to its sensor data quality. This kind of testing helps to identify any weaknesses in the model’s ability to accurately process and interpret the data.

Adversarial Attacks:

Adversarial attacks are a type of attack that can be used to fool DNN models and cause them to misclassify data or produce incorrect results. Some of examples of Adversarial attacks are:

Adversarial patch attack: This type of attack involves adding a small patch to an image that causes the DNN model to misclassify the image.

Adversarial perturbation attack: This type of attack involves adding small perturbations to an image that are not visible to the human eye but cause the DNN model to misclassify the image.

FP, FN, TP, TN benchmarking:

FP/FN, and TP/TN metrics are used to evaluate the accuracy of a DNN model when it is tested on known testing data. False Positive (FP) and False Negative (FN) metrics measure the number of incorrect predictions made by the model, while True Positive (TP) and True Negative (TN) metrics measure the number of correct predictions.

In 2019, some car manufacturers and Tier I suppliers released the white paper, SAFETY FIRST FOR AUTOMATED DRIVING [33], which offers an in-depth analysis of the verification and validation techniques for SAE L3 and L4 automated driving from a practical perspective. They demonstrate the positive risk balance of automated driving solutions compared to the average human driving performance and also provide guidance for potential methods and considerations in the V&V of Level 3 and 4 automated driving systems. However, it is not intended to serve as a final statement or minimum or maximum guideline or standard for automated driving systems.

6.7 Conclusion

An analysis of safety risks associated with deploying DNN classifiers in guiding autonomous road vehicles, conducted in this chapter, has unveiled a spectrum of safety risks. These risks are present in all development phases of the design, training, implementation, and deployment of DNN classifiers. Addressing these risks is crucial to ensure the safety of AI solutions. To mitigate these challenges, a set of measures and their alignment with safety standards is proposed, as summarized in Table 6.3.

Mapping Safety Measures to Safety Standard:

III.A ARCHITECTURAL MODEL DESIGN	
Selecting suitable AI technology DNN model, Activation function, etc.	ISO/AWI PAS 8800:2023 General requirements 7.4.1- 7.4.10
Function degradation Plausibility checks	ISO/AWI PAS 8800:2023 12.5 Measures to assure safety of the AI system during operation
System redundancy and fusion strategies	ISO/AWI PAS 8800:2023 7.6.1 Measures for Architectural Redundancy
Safety Analysis	ISO/AWI PAS 8800:2023 10.4 Safety analysis of the AI system
Comprehensive Review	ISO/AWI PAS 8800:2023

	11.5 Structuring Assurance Arguments for AI Systems
III.B TRAINING OF DNN CLASSIFIERS	
Hyperparameter tuning	ISO/AWI PAS 8800:2023 7.6.4 Training safety measures
Dataset Safety Analysis	ISO/AWI PAS 8800:2023 8.4.3 Dataset Safety Analysis
Adversarial attack testing	ISO/AWI PAS 8800:2023 7.6.4.2 Robust Learning
III.C IMPLEMENTATION AND INTEGRATION OF DNN ALGORITHM CLASSIFIERS	
Qualification of ML libraries	ISO 26262-8:2018 Software tool qualification report
Reinforcement of low complexity Coding guideline	ISO 26262-6:2018 Table 1 — modelling and coding guidelines
6.7.1 III.D. VERIFICATION AND VALIDATION METHODS FOR ML-BASED AUTOMATED DRIVING SYSTEMS	
Static code analysis Fault injection test Unit/ scenario-based / endurance testing	ISO 26262-6:2018 Table 10 — Methods for verification of software integration
False Negative / Positive benchmarking	ISO/AWI PAS 8800:2023 9.5.5.1 Performance evaluation methods

Table 6.3: Mapping of proposed methods to automotive safety standards ISO26262, PAS 8800 Normative Demands

7 Chapter 7 - Enhancing Safety in RL Agents: The 'Safety Override Layer' for Autonomous Driving

7.1 Introduction

Ensuring the safety of Reinforcement Learning (RL) methods in the context of Highly Automated Driving (HAD) is a paramount concern that demands innovative solutions. In this chapter, a novel approach is delved into, addressing the intricacies of safety within RL-based systems for highway driving. Beyond conventional methodologies, a comprehensive safety layer, comprising eight key features designed to enhance the safety of RL agents specifically for HAD, is introduced.

The first key feature of this safety layer is enhancing the training of RL agents, particularly emphasizing the maximization of safe explorations. In this regard, a methodology is proposed to quantify, visualize, and maximize the exploration behaviour trend of the RL agent across the continuous state spaces of environments such as autonomous driving.

Beyond this, the safety layer is designed to enable integrating guidance from human expert, prevent unsafe actions, impose a safety constraint on agent policy optimization, dynamically shape rewards based on the safe margins of automated driving, introduce redundant agents, and uphold a fail-safe strategy for situations where the Operational Design Domain (ODD) is violated, and safety cannot be ensured by the reinforcement learning agent alone. Additionally, a methodology is proposed to enhance the adaptability and safety of the RL agent by imitating the decision-making processes of human drivers.

One notable aspect of the proposed safety layer involves the definition of the safety margin scheme through the incorporation of key performance indicators (KPIs) related to the safety of automated driving.

This chapter also presents details regarding the implementation of the proposed algorithms, along with specifics of their realization within a software-in-the-loop simulation framework.

7.2 Safety Layer for Autonomous Driving RL Agent

In the context of our autonomous driving system, the proposed safety override layer in this dissertation serves as a crucial critical fail-proof support system for the RL agent, ensuring not only optimal performance but also adherence to safety standards. Eight key features for the proposed safety layer are suggested to addresses various dimensions of safety, enabling the RL agent to navigate the complex dynamics of real-world scenarios while prioritizing the safety of passengers, pedestrians, and other traffic participants.

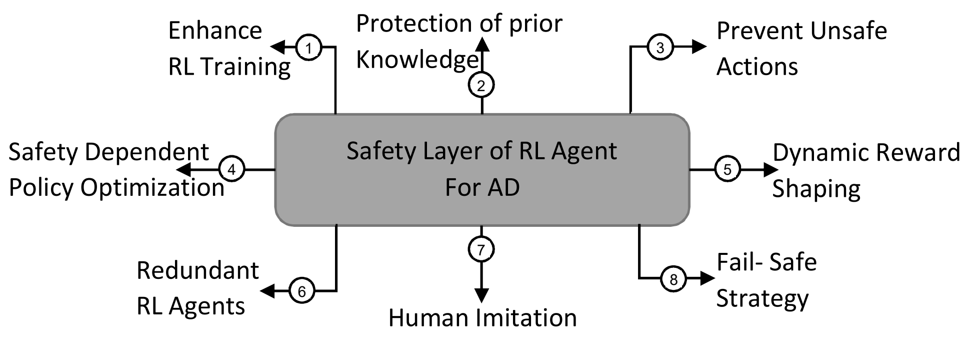


Fig. 7.1: Proposed RL agent safety layer with 8 key elements

- Enhance RL Training
    - Maximizing exploration:
      Encouraging max exploration of safe actions introducing exploration metric
    - Integrating Human Guidance to Enhanced Initial RL Training Performance
- Protection of Prior Knowledge
    - protection of Q-values and policy values
- Prevent Unsafe Actions
    - Rule based logics to prevent unsafe actions in the safety context based on safe margins during and after training phase.
- Safety Dependent Policy Optimization
    - Safety penalty for policy update constraints to return to safe state
- Dynamic Reward Shaping
    - Adjusting the reward based on driving conditions to influence the decision-making of the agent.
- Redundant RL Agents
    - Safety arbitration between two RL agents. E.g. (A3C, DDPG, Multi-agent, DQN) Policy1 vs Policy 2
- Human Driver in Loop
    - policy improvement after deployment: learning from human (monitoring human driving, estimating q values and rewards, and safety KPIs) and updating policy if human actions were safer
- Fail-Safe Strategy
    - HW or communication failure, ODD violated, input sensor data or actuators failed, …

Safety Margin

- Safety KPI, safety goals
- Safety margin, 2 levels of safety, road condition, vehicle control conditions, sensor availability….
- Behaviour of the RL agent based on the safety margin

7.2.1 Safety Layer- Enhance RL Training

The enhancement of RL training is accomplished by maximizing RL agent exploration and integrating human expert guidance during the training phase.

7.2.1.1 Maximizing Exploration of Reinforcement Learning Agents during Training

The objective here is to maximize exploration rate of safe actions in reinforcement learning. It includes the enforcement of robust policies so that the solution is able to adapt to diverse scenarios. For example, the discovery of rarely encountered critical states, addressing model inaccuracies, prevention of premature convergence to suboptimal policies, enhancement of safety margins, and preparation for handling unexpected scenarios, to name a few are the type of policies deployed.

7.2.1.1.1 Quantifying and Visualization of the RL Agent Exploration Metric in Continuous Action/State Spaces

The *"Exploration Rating"* introduced in this dissertation aims to parametrise and create relative quantification metric to measure the exploration trend of a learning agent during training throughout continuous and contiguous (discretization) action spaces. In the context of reinforcement learning, where agents must strike a delicate balance between exploiting known strategies and exploring new possibilities, the Exploration Rating reflects the agent's tendency to venture into various regions space.

By discretizing a x-dimension continuous action space into manageable segments and tracking the visit count and the taken actions for each segment, the Exploration Rating provides a relative measure of how extensively the agent explores different aspects of the action space with respect to events. A higher Exploration Rating suggests that the agent is actively seeking novel strategies, which can be advantageous for discovering optimal policies. In scenarios where exploration is not excessively costly or risky, higher exploration allows the agent to uncover more effective strategies, intelligent adaptation and robustness in handling diverse environmental conditions.

The Exploration Rating serves as a valuable tool for promoting and maximizing exploration, particularly in scenarios where random action selection is desired. By effectively quantifying the agent's tendency to venture into diverse regions of the action space, this metric contributes to the optimization of the learning process, enhancing the efficiency and effectiveness of reinforcement learning algorithms.

7.2.1.1.2 Methodology for Computing and Visualizing the Exploration Rating

Assumption: A state is defined with x-dimensional continuous factors.

State Definition: A state, denoted as {s[1], s[2], ..., s[x]}, consists of x continuous factors.

<u>Discretizing the state spaces:</u>[19]

In the presence of continuous action or state spaces within the environment, discretization emerges as a practical solution for facilitating exploration in

[19] Other resources such as [1] have addressed the topic from a distinct perspective. The methodology proposed in this dissertation, however, takes a fundamentally different approach.

reinforcement learning. Continuous spaces, characterized by an infinite range of possible actions or states, present computational challenges that can impede the learning process. Discretization involves partitioning these continuous spaces into discrete intervals or bins, rendering them more amenable to exploration by learning agents. By transforming the continuous space into a finite set of well-defined contiguous states or actions, discretization enables the agent to systematically explore various possibilities, contributing to a more efficient learning process.

Uniform Discretization of State Space:

Each factor, s[i], is uniformly divided into $n_i$segments, where i varies from 1 to x.

For example, S[1] has segments : $\{1, 2, \dots, n_1\}$ and S[x] has segments : $\{1,2, \dots, n_x\}$

Total number of segments:

The total number of segments in the discretized state space is calculated as follows:

$$N_{Total} = \prod_{i=1}^{x} n_i = n_1 . n_2 . \dots . n_x \tag{7.2.1}$$

This discretization process enables the representation of continuous state factors in a segmented form, forming the basis for computing the exploration metric.

<u>A simplified example of the discretizing approach:</u>
To elaborate on the discretization approach, two essential parameters in the modelling of an autonomous vehicle (AV) within a possible scenario is given. The distance to an obstacle and the velocity of the AV. The discretization process involves the representation of these continuous parameters in segmented form to facilitate a structured analysis of the environment.

The distance to the obstacle is characterized by a range spanning from 0 to 200 meters, simulating the capabilities of a long-range radar. Simultaneously, the AV's velocity ranges from 0 to 100 km/h. To achieve a fine-grained discretization, the approach employs 100 divisions for velocity, with each division representing a 1 km/h increment. Similarly, for distance, 200 divisions are utilized, and each division corresponds to a 1-meter increment.

This meticulous division of the continuous range into discrete segments results in a discretized grid comprising a total of 20,000 segments (100 by 200). Each segment encapsulates a unique combination of distance and velocity, providing a structured and comprehensive representation of the potential states within the given environment. This discretization approach lays the foundation for further analysis and the computation of the exploration metric.

7.2.1.1.3 Conversion of x-dimensional state space to segmented state space

The conversion of an x-dimensional continuous state space into a sequentially segmented discretized state space is a crucial step in the exploration metric computation process.

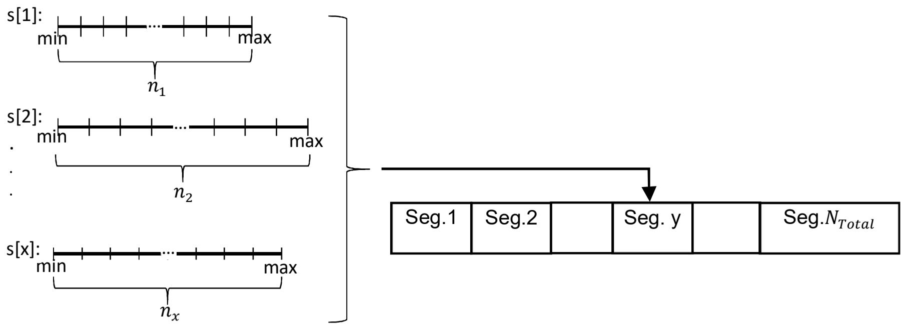


Fig. 7.2: Conversion of a x-dimensional continuous state space into a sequentially segmented discretized state space

By incorporating combinatorial principles, the methodology systematically arranges segments of discretized factors next to each other - s[1], s[2],…, s[x] - to uniquely transform discretized segments of the state space to a grid form, as depicted in Figure 7.2. This transformation facilitates a more granular representation of the state space, enabling a detailed analysis of the agent's exploration behaviour during the training phase.

Segment 1: Represents the starting point of each x-factor within the discretized state space.

Segment N_{total} Corresponds to the end point of each x-factor within the discretized state space.

Methodology to Compute Absolute Segment Number (seg. y)

1. **Relative Position Calculation:**

For each factor s[i], calculate the relative position ($relative\_position_i$) within the segmentation using the formula:

$$relative\_position_i = \max\left(1, \min\left(\left\lfloor \frac{s[i]-s[i]_{min}}{s[i]_{max}-s[i]_{min}} \times n_i \right\rfloor + 1, \mathrm{n_i}\right)\right) \quad (7.2.2)$$

Where, $s[i]_{max}$ *and* $s[i]_{min}$ denote the maximum and minimum values of the continuous factor s[i] and $\mathrm{n_i}$ represents the number of segments for factor s[i].

The symbol ⌊ ... ⌋ represents the floor function. The floor function takes a real number as input and returns the largest integer that is less than or equal to that number.

The formula ensures that $relative\_position_i$ is always between 1 and $\mathrm{n_i}$.

2. **Absolute Segment Number Calculation**

The Absolute Segment Number (y in Figure 7.2) is computed using the following formula:

$$y = relative\_position_1 + n1 \times (relative\_position_2 - 1) + n1 \times n_2 \times (relative\_position_3 - 1) + \cdots + n1 \times n_2 \times \ldots \times n_{x-1} \times (relative\_position_x - 1) \quad (7.2.3)$$

7.2.1.1.4 Data Structure to Store Taken Actions in State Space Segments:

In this section, the data structure designed for the storage of information regarding actions taken within each segment of the state space is introduced.

It is assumed that the action space consists of M actions (or is equivalently segmented into M segments).

For each segment x, denoted as Nr. Of Action 1, ..., Nr. Of Action M, a dedicated array is established to track the number of times each action/event in the action space (analysis span) has been taken during the training process. The structure of each segment, represented as segment[i], takes the form:

$$segment[i] = [n_{a1}, n_{a2}, \dots, n_{aM}] \qquad (7.2.4)$$

Where i is varied from $1, \dots, N_{Total}$ and n_{aj} signifies the number of times action a_j (j= 1, …, M) has been taken within this specific segment.

7.2.1.1.5 Exploration Metric

The Exploration Metric (EM) is a metric employed to quantify the exploration behaviour of a learning agent throughout the training process. The formulation and incorporation of this metric in the dissertation offer a novel perspective on assessing the exploration behaviour of a reinforcement learning agent.

The exploration metric is defined as the ratio of the number of taken actions to the total number of possible actions, expressed as a percentage:

$$Exploration\ Metric\ (EM1) \doteq \frac{Nr.\ of\ taken\ Actions}{Nr.\ of\ all\ possible\ Actions\ in\ all\ segments}\ x\ 100\% \qquad (7.2.5)$$

Or alternatively:

$$Exploration\ Metric\ (EM2) \doteq \frac{Nr.\ of\ taken\ Actions}{Nr.\ of\ all\ possible\ Actions\ in\ visited\ segments}\ x\ 100 \qquad (7.2.6)$$

The nominator and denominator components of the Exploration Metric are further detailed as follows:

$$Nr.\ of\ taken\ Actions = \sum_{i=1}^{N_{Total}} \sum_{j=1}^{M} Min(1, segment[i][j]) \qquad (7.2.7)$$

Here, segment [i] [j] denotes the count of action j in segment i, and the 'min' function ensures that the action is considered in the metric computation only once.

$$Nr.\ of\ all\ possible\ Actions\ in\ all\ segments = M \times N_{Total} \qquad (7.2.8)$$

The total number of possible actions is calculated as the product of the number of actions M and the total number of segments N_{Total}.

$$Nr.\ of\ all\ possible\ Actions\ in\ visited\ segments = M \times N_{Visited} \qquad (7.2.9)$$

The total number of possible actions is calculated as the product of the number of actions M and the total number of visited segments $N_{Visited}$.

Segment i is considered visited if at least one of the actions in segment i has been taken and $n_{aX} > 0$.

7.2.1.1.6 Implementation of the Simulation Framework for Quantifying and Visualizing Proposed Exploration Metrics

The Software in the Loop (SiL) Simulation framework employs the Gym Cartpole environment from Chapter 5 to quantify and visualize exploration metrics, as proposed earlier in the previous section[20]. With four continuous factors for each state, this framework provides an ideal platform for experimentation.

Discretization of the State Space:

Following the discretizing approach proposed in the previous section of this dissertation, the 4-dimensional state space of the environment is transformed into a segmented grid comprising 10,000 segments (10 x 10 x 10 x 10 for each factor) in this implementation, enabling a detailed examination of the RL agent's exploration behaviour. The action space is confined to two actions—pushing left or pushing right.

M = 2, N_{Total} = 10k segments

The experiment involves running 20,000 episodes, each lasting 500 time steps, resulting in a total of 10 million iterations. Notably, the early termination of training based on the accumulated reward reaching a threshold is disabled in this configuration.

The primary focus of this simulation is to study the exploration behaviour of the RL agent, without considering its overall performance or design variations.

The chosen epsilon-greedy strategy is designed to promote exploration throughout all episodes, starting with 100% exploration and gradually decreasing to 1% with an exponential decay over 2000 episodes. Here is the corresponding code snippet:

```
# Epsilon greedy parameters
EPS_START = 1
EPS_END = 0.01
EPS_DECAY = 2000

self.epsilon = EPS_END + (EPS_START - EPS_END) * math.exp(-1. * DecayCount /
EPS_DECAY)
```

Software implementation:

The code snippet to implement the action storage data-structure, ‘ActionStorage_DataStructure’, is as follows:

```
class ActionStorage_DataStructure:
    def __init__(self, num_segments, action_space):
        # Initialize the data structure with zeros
        self.segments = [[0] * len(action_space) for _ in range(num_segments)]
        self.action_space = action_space
        self.visited_segments = set()  # Set to keep track of visited segments
    def record_action(self, segment_index, action):
        # Calculate the index of the action in the action_space array
        action_index = self.action_space.index(action)
```

[20] The entire source code is provided as a supplementary document.

```
        # Increment the count for the specified action in the given segment
        self.segments[segment_index][action_index] += 1
        # Check if the segment has been visited before
        if segment_index not in self.visited_segments:
            # Add the segment to the set of visited segments
            self.visited_segments.add(segment_index)
    def get_segment_data(self, segment_index):
        # Retrieve the counts for all actions in the specified segment
        return self.segments[segment_index]
    # Return the count of visited segments
    def get_visited_segments_count(self):
        # Return the count of visited segments
        return len(self.visited_segments)
    def get_taken_actions_count(self):
        # Compute the total number of taken actions over visited segments
        total_actions_count = 0
        for segment_index in self.visited_segments:
            # Increment the count for each action in the current segment
            total_actions_count += sum(1 for action in self.action_space if
self.segments[segment_index][self.action_space.index(action)] > 0)
        return total_actions_count
```

This structure is employed to record visited segments and unique taken actions in each segment, and retrieve all actions for a given segment.

The 'get_visited_segments_count' and 'get_taken_actions_count' methods provide counts of visited segments and taken actions, respectively.

```
SegmentNr= compute_segment_number(state.squeeze().tolist())
action_data_structure.record_action(SegmentNr, action)
```

The code snippet 'compute_segment_number' calculates the current segment using the four factors of the current state, based on the discretization approach as proposed in formulas 7.1.2 and 7.1.3.

```
def compute_segment_number(current_state):
    """
    Compute the current segment number from the current state.
    Parameters:
    - current_state: List[float], representing the current state with 4 factors.
    Returns:
    - current segment number [int].
    """
    segment_counts = [10, 10, 10, 10]  # Number of segments for each factor
    factor_ranges = [(-4.8, 4.8), (-5, 5), (-24, 24), (-5, 5)]
    relative_positions = [
        max(1, min(int((current_state[i] - min_value) / (max_value - min_value) *
segment_counts[i]) + 1, 10))
        for i, (min_value, max_value) in enumerate(factor_ranges)
    ]
    current_segment = (
        relative_positions[0] +
        (relative_positions[1] - 1) * segment_counts[0] +
        (relative_positions[2] - 1) * segment_counts[0] * segment_counts[1] +
        (relative_positions[3] - 1) * segment_counts[0] * segment_counts[1] *
segment_counts[2]
    )
    return current_segment
```

Finally, the code snippet that calculates exploration metric 1 and metric 2 based on formulas 7.1.5 to 7.1.9:

```
# Append data to lists for plotting
exploration_metric1_list.append(100 * unique_actions_count / (NSegmentTotal *
action_space_size))
exploration_metric2_list.append(100 * unique_actions_count /
(visited_segments_count * action_space_size))
```

7.2.1.1.7 Evaluation of the Experiment Results

Utilizing the computational power of a high-speed system, the training process efficiently concludes within a few hours of training. In the conducted training, no optimization is applied to force the agent to explore new segments and actions.

The results of quantifying and visualizing exploration metrics using the discretizing technique are presented in Figure 7.3 (a), (b), (c), (d) and Figure 7.4. This visualization offers a comprehensive depiction of the exploration behaviour, providing valuable insights into the agent's interactions with the environment.

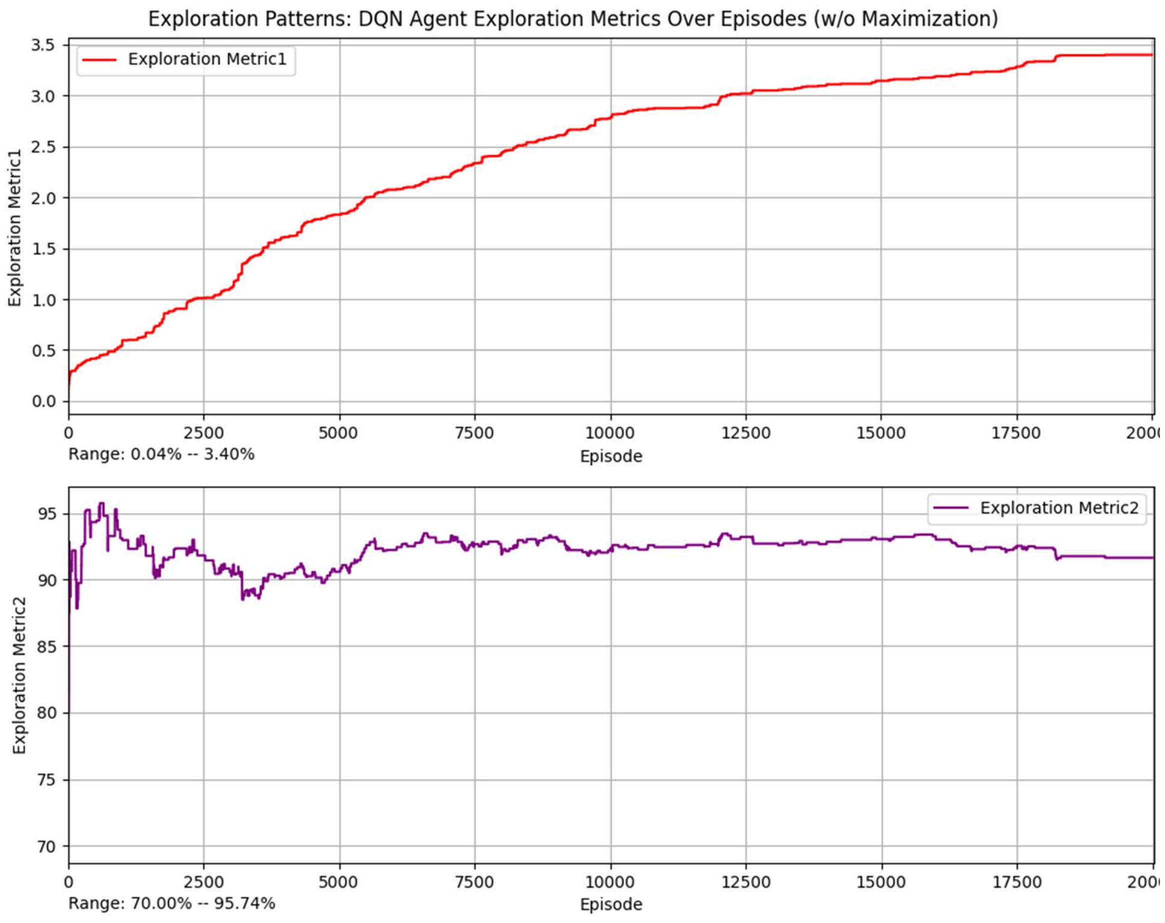


Fig. 7.3: Quantifying and visualizing of the Exploration behaviour of an RL Agent (a) and (b) w/o Maximization

Exploration Metric 1, as illustrated in Figure 7.3 (a), reveals a notable deficiency in the agent's exploration behaviour, evident by a maximum exploration ratio of only 3.4%. This suboptimal performance persists despite the agent being compelled to explore on most occasions. The limited exploration observed in the figure underscores the challenge the agent faces in adequately exploring the environment.

Exploration Metric 1 exhibits a continually increasing ratio, as its denominator remains constant, while the numerator is cumulative and consequently on the rise. In contrast to Exploration Metric 1, Metric 2 has the capacity to both increase and decrease. A decrease in Metric 2 (Figure 7.3 (b)) occurs when not all actions are executed in newly visited segments.

Due to the limited size of the action space, which consists of only two actions, the agent has a high chance of taking all available actions within the visited segments. This factor is the primary cause behind the consistent display of high ratios in Exploration Metric 2.

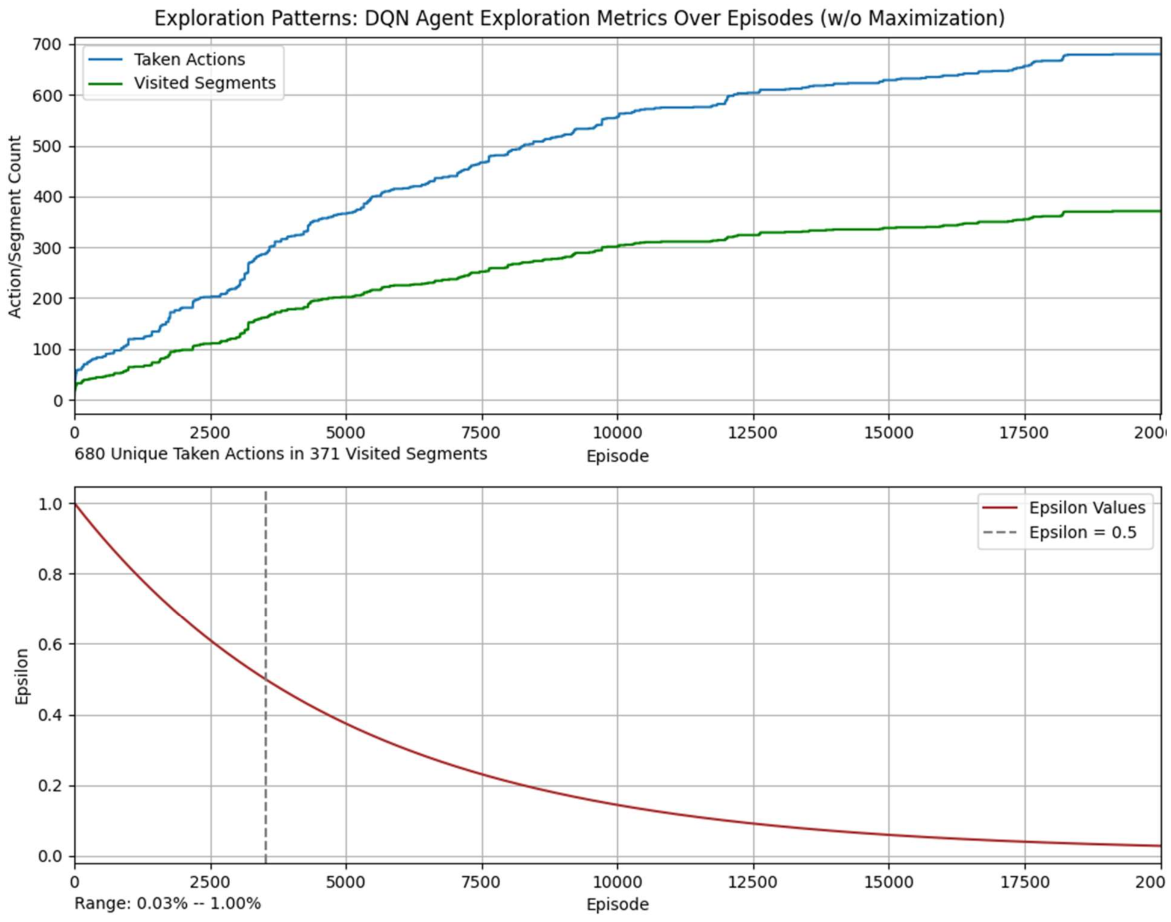


Fig. 7.3: Quantifying and visualizing of the Exploration behaviour of an RL Agent (c) and (d) w/o Maximization

Figure 7.3 (c) illustrates both the number of visited segments and the number of actions taken within those segments. A noteworthy observation in this quantification is the significant number of segments that the agent does not explore at all. This deficiency is further highlighted in Figure 7.4.

The exploration-exploitation trade-off configuration of the agent is shown in Figure 7.3 (d). In order to encourage the agent to exploit its policy less, the epsilon hyperparameters are configured to transition from 100% exploration (representing random actions) to 3% towards the end of the training.

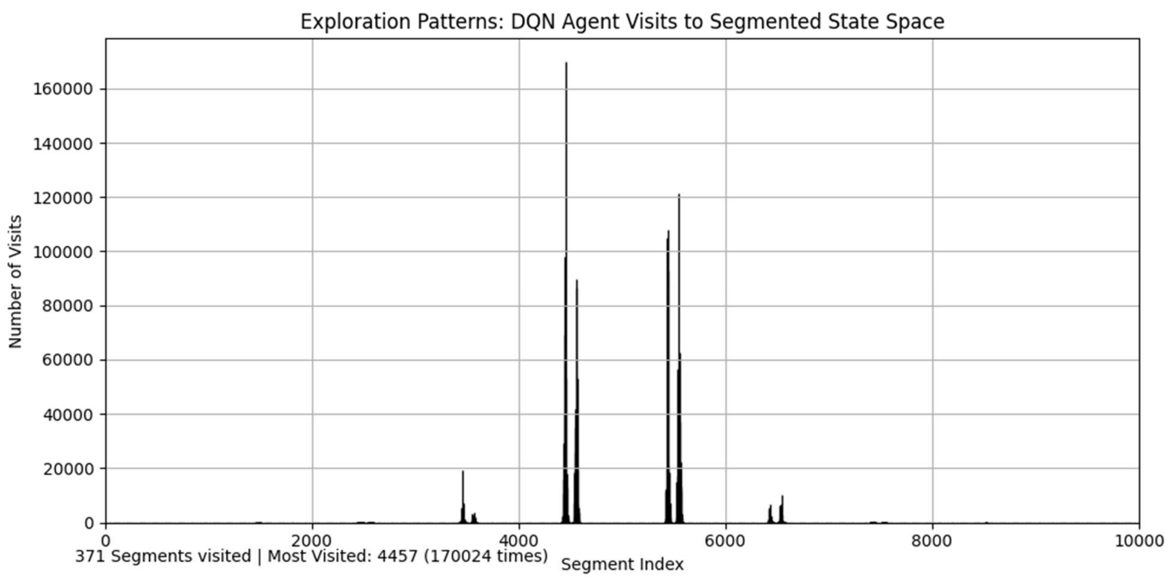


Fig. 7.4: Quantifying and visualizing of the visited segment of an RL Agent w/o Maximization

The distribution of visited segments across the segment spectrum is visualized in the bar chart plot in Figure 7.4. The RL DQN agent aims to maintain the cart and the pole at the centre of their trajectories. This objective is evident in the observation that segments around 5000 (out of 10000), where all four continuous factors fall within the middle of their respective ranges, are the most frequently visited segments. Quantifying and visualizing the exploration behaviour of the DQN agent highlights that a few segments are frequently visited, such as segment 4457, which was visited over 170k times, while many segments (10000 - 371) are never visited. This reveals a significant percentage of the environment remains unexplored during the training of the RL agent, $N_{Visited} \ll \ N_{Total}$. This may indicate that the deployed agent could lack robustness in handling a broad range of scenarios, potentially affecting its performance in real-world applications where comprehensive training is crucial.

7.2.1.1.8 Algorithm: Epsilon-Greedy-with-Exploration Maximization

The novel approach of state space segmentation proposed in this dissertation provides the agent with the ability to evaluate whether an action has been taken in the current segment, thus facilitating more informed decision-making during exploration.

The exploration maximization algorithm is integrated with the epsilon-greedy algorithm during the training phase, providing the agent with more intelligent action selection strategies. Its underlying principle prompts the agent to take actions that haven't been executed before, maximizing exploration. This algorithm does not disrupt the exploration-exploitation trade-off; rather, it becomes specifically active when exploration is needed, exclusively during phases when a random action is desired.

The incorporation of exploration maximization enhances the agent's adaptability and encourages a comprehensive exploration of the environment.

The pseudo code of the algorithm is presented below:

Algorithm 7.2.1.1 **Epsilon-greedy-with-Exploration Maximization**

```
1.  function select_action (current_State, current_segment):
2.  # Input: current_State, current_segment
3.  # Output: action, Replace_Action_Count
4.  If Epsilon > RandomNumber   # exploration desired
5.  action = Select a random action from action space
6.  If Action already taken in current_segment
7.          If any Action available in current_segment # not taken before
8.          # action is replaced!!
9.              action = Select a random action from available in current_segment
10.             # increment the action_replacement counter!!
11.             Replace_Action_Count ++
12. Else
13.         action = Policy_Net[current_state] # exploitation
14. return (action, Replace_Action_Count)
```

The code snippet to implement the exploration maximization algorithm is as follows:

```
def select_action(self, state, current_segment):
    global action_replacement_count
    randNr = random.random()

    if self.epsilon > randNr:
        # Exploration: Randomly select an action
        selected_action = env.action_space.sample()

        # Check if the selected action has been taken before
        if current_segment[selected_action] > 0:
            available_actions = [action for action in range(env.action_space.n) if
current_segment[action] == 0]
            if available_actions:
                selected_action = random.choice(available_actions)
                action_replacement_count += 1
    else:
        with torch.no_grad(): # Exploitation from the policy network
            selected_action = self.policy_net(state).max(1)[1].item()

    return torch.tensor([[selected_action]], device=device, dtype=torch.long)
```

7.2.1.1.9 Exploration Maximization Algorithm Experimentation

Continuing under the same experimental conditions, the simulation is reiterated with the incorporation of the exploration maximization. This innovative approach, added to the epsilon-greedy strategy, aims to further enhance the RL agent's exploration behaviour.

As in the previous setup, the experiment spans 20,000 episodes, each lasting 500 time steps, totalling 10 million iterations.

The experiment outcomes depicted in Figure 7.5 (a) and (b) reveal a subtle improvement in the exploration metrics. Specifically, there is an enhancement of 0.06% (3.46% - 3.4%) in the first metric and a slightly more significant improvement of 0.08% (96.82% - 95.74%) in the second metric.

In Figure 7.5 (c), it is evident that the algorithm effectively substituted 95 random actions to augment the agent's exploration during training and prioritize actions it had not previously undertaken. While it's conceivable that the agent might have eventually taken some of these actions in subsequent iterations even without this method, the exploration maximization algorithm ensures their immediate consideration, contributing to a more proactive exploration strategy.

Figure 7.5 (d) is presented to demonstrate that the exploration-exploitation strategy remains consistent with the experiment conducted without the exploration maximization algorithm.

Figure 7.6 illustrates comparable results to the previous findings, underscoring that a significant portion of the state space segments remains unexplored even after extensive training. This emphasizes the persistent challenge of achieving comprehensive exploration within the RL agent's environment.

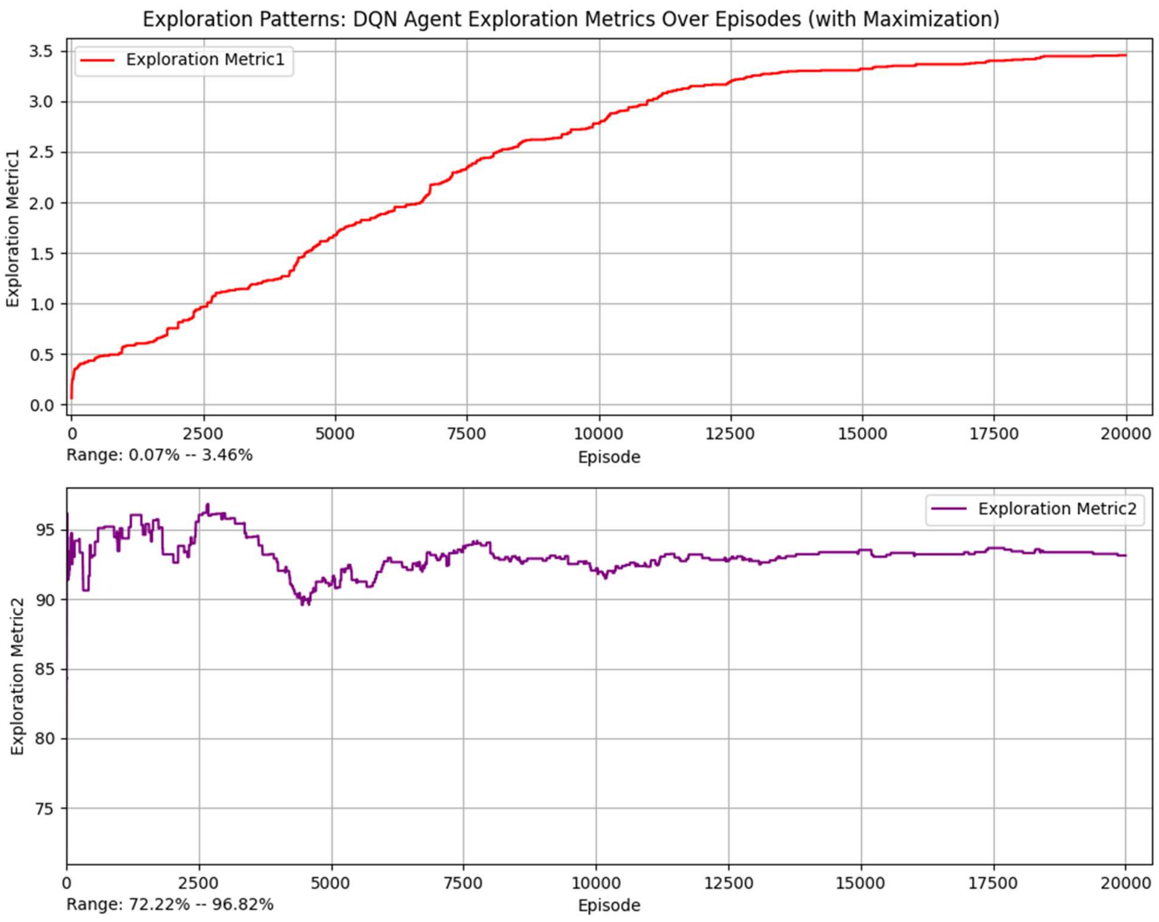


Fig. 7.5: Quantifying and visualizing of the Exploration behaviour of an RL Agent with Maximization (a) and (b)

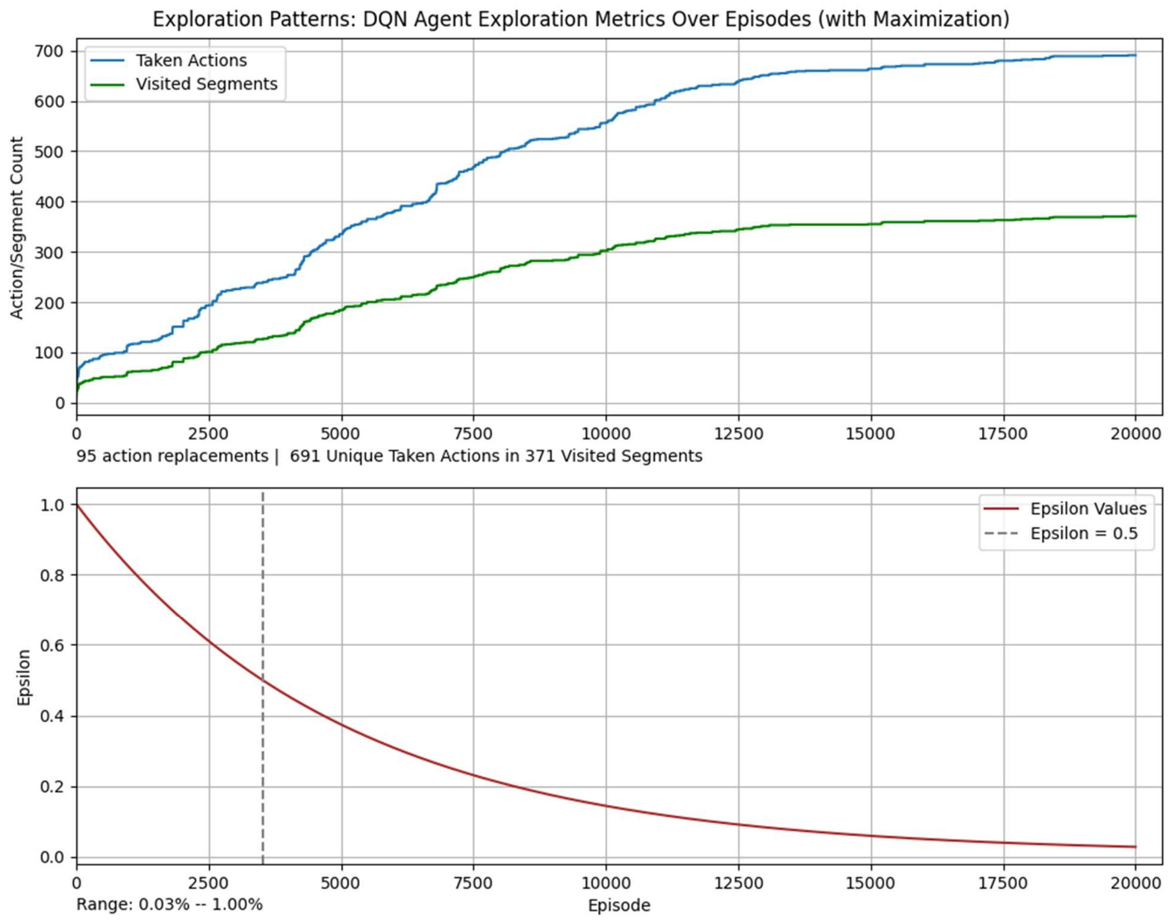


Fig. 7.5: Quantifying and visualizing of the Exploration behaviour of an RL Agent with Maximization (c) and (d)

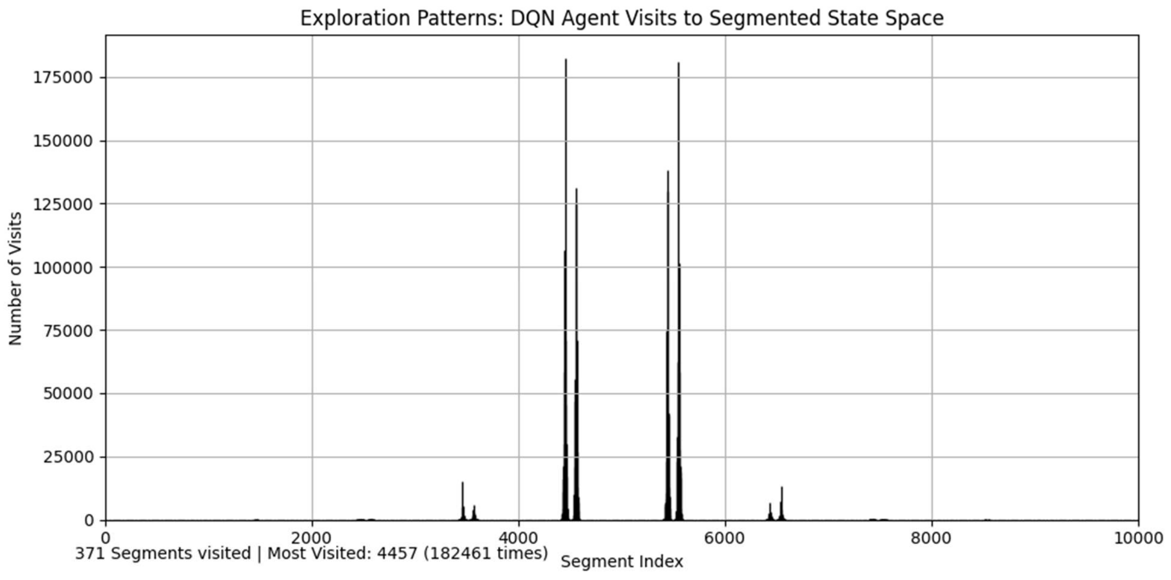


Fig. 7.6: Quantifying and visualizing of the visited segment of an RL Agent with Maximization

7.2.1.1.10 Conclusion

The employed proposed and implemented methodology of state space segmentation to quantify metrics and visualize exploration, as outlined in this dissertation, proves to be an effective approach, and seems to successfully provide detailed perspectives on the agent's learning dynamics.

The observed subtle improvements in exploration metrics underscore the ability of the algorithm to actively substitute random actions, enhancing the agent's proactive exploration during training.

It's noteworthy that this approach, while impactful in a smaller action space (limited to two actions in this experiment), holds promising potential for real-world applications with larger action space sizes. For instance, in scenarios like autonomous driving, where actions such as steering wheel or acceleration ratio are continuous and can be discretized into a larger number of segments, the exploration maximization algorithm is likely to offer even more substantial benefits. The adaptability of this algorithm to higher action space sizes could contribute significantly to the efficacy of training in complex, real-world environments.

The outcomes of this experiment, revealing the ongoing challenges in achieving thorough exploration, pave the way for future enhancements and diverse applications of the exploration maximization algorithm.

7.2.1.2 Integrating Human Guidance to Enhance RL Training[21]

Integrating human guidance in reinforcement learning involves leveraging the expertise of human experts to enhance the training process of RL algorithms. This collaborative approach introduces valuable insights and domain knowledge that may be challenging for algorithms to acquire independently. Additionally, integrating human guidance enhances safety by incorporating real-world expertise to mitigate risks and improve the robustness of the trained algorithms.

In this dissertation, three key strategies are proposed to harness the valuable insights from human experts, shaping the learning process of reinforcement learning agents.

- Targeted Policy or state-action Q-Values Initialization
- Forcing certain training patterns by Shaping Environment feedback (observation, state, reward)
- Forcing RL agent to always / never take certain actions in certain states

7.2.1.2.1 Targeted Policy or State-Action Q-Values Initialization:

Recalling RL training algorithms (tabular 4.3.3, DQN 4.5.2, or Policy Gradient 4.7.2.2), where policy networks are typically initialized with random weights and Q-values initialized with zero.

In the experimentation detailed in chapter 5.3.1.5, the weighting factors of the policy neural network was initialized using the HE initialization method, resulting in a rapid increase in the total reward of the agent.

Contrary to HE initialization or similar methods, which primarily address training challenges or generalization issues, the Human-shaped initialization method proposed here involves initializing the agent's policy or state-action Q-Values to prioritize specific decision-making by the agent.

It particularly involves initializing the policy network to output specific Q-values (in the case of DQN RL) or actions (in the case of policy gradient RL) during its forward propagation pass.

Forward Pass:

$$Q(s, a) = f_\theta\,(s) \qquad (7.2.10)$$

For a known state: s and desired known action: a, the weighting factors of the policy neural network can be determined.

As an example of such targeted policy initialization, consider a scenario where the autonomous vehicle exhibits a lateral deviation of 50 cm to the left. A proper initialization of the network would compel the agent to turn the steering wheel by 15 degrees to the right, promptly correcting the deviation. Without such a targeted

[21] Other resources such as [2] and [3] have addressed the topic from a distinct perspective. The methodology proposed in this dissertation, however, takes a fundamentally different approach.

initialization, it would take the agent significantly longer or potentially result in suboptimal manoeuvres to rectify the lateral deviation.

7.2.1.2.2 Forcing certain training patterns by Shaping Environment feedback (observation, state, reward)

To further enhance the training of RL agents, leveraging the expertise of human professionals with domain knowledge, this section introduces a proposed method for shaping environment feedback.

The strategy involves deliberate manipulation of observations, state representations, and rewards during training. This intentional shaping of the environment feedback, in terms of state, action, and observations, aims to guide the RL agent toward unexplored areas of the state space, expose the agent to unforeseen and unexpected situations, facilitate safe behaviours in individual scenarios, and ultimately improve its adaptability and generalization.

- $s_{t+1} \leftarrow s_{Human}$
- $r_t \leftarrow r_{Human}$
- $observation_{t+1} \leftarrow observation_{Human}$

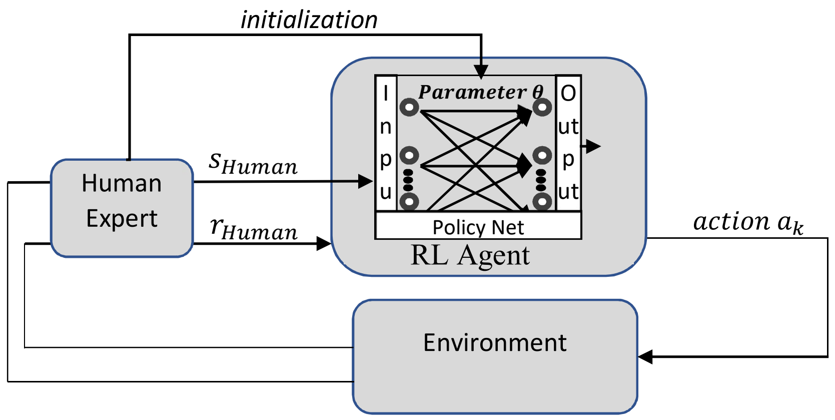


Fig. 7.7: Human Guidance in RL Training: Policy Initialization & Shaping Environment Feedback

<u>Examples of state shaping in the context of AD:</u>
For instance, simulating a slippery road by reducing the distance to an already seen obstacle after braking, longitudinal distance to target vehicle 50 [m], lateral distance to road boundaries [30cm, 70cm], vehicle speed at: 100 km/h, …

<u>Examples of reward shaping in the context of AD:</u>
A reward of +10 or a penalty of -20 for the last taken action to influence the q-learning or policy update of the agent.

<u>Examples of observation shaping in the context of AD:</u>
A curvy road ahead, battery charge level at 30%, detours, or roadwork in 5 km, heavy rain expected, …

7.2.1.2.3 Forcing RL agent to always or never take certain actions in certain states

A crucial aspect of enhancing RL training involves instructing the agent to consistently adopt or avoid specific actions in specified states. The human expert with domain knowledge can compel the agent during training to always take or never take an action in certain states (override). This strategy aims to enforce decision-making patterns that align with safety requirements. By compelling the RL agent to exhibit particular actions under specific conditions, the training process is steered toward a safer performance.

This section delves into the methodology of instructing the RL agent to always or never take certain actions in predefined states.

The pseudo code of the algorithm to force an action is presented below:

Algorithm 7.2.1.2 **Human Guidance Force Action Policy**

```
1.  function select_Action (current_state):
2.  # Input: current_state, Output: action
3.  # Set of states where action A should be taken
4.    setOfStatesToAlwaysTakeActionA = {state1, state2, state3, ...}
5.    if current_state belongs to setOfStatesToAlwaysTakeActionA:
6.      return action: A
7.    else:
8.        # Use another method or policy for action selection
9.        # For example, exploit the action from the policy network
10.       return Policy_Net[current_state] # exploitation
```

The state, in this context, comprises signals that indicate the safe margin (see section 7.3). Thus, the proposed method can compel specific actions in certain states or when any of the comprising signals indicate that the safety margin has been reduced. In such cases, the pseudo code in line 5 can tackles the condition:

```
# current_state[x]: comprising signal in current_state

if safetyMargin (current_state[x]) < threshold
```

similarly, the pseudo code for the algorithm to avoid an action is presented below:

Algorithm 7.2.1.3 **Human Guidance Avoid Action Policy**

```
1.  function select_Action (current_state):
2.  # Input: current_state, Output: action
3.  # Set of states where action A should be avoided
4.    setOfStatesToAlwaysAvoidActionA = {state1, state2, state3, ...}
5.    if current_state belongs to setOfStatesToAlwaysAvoidActionA:
6.      return chooseAlternativeAction (ActionA)
7.    else:
8.        # Use another method or policy for action selection
9.        # For example, exploit the action from the policy network
10.       return Policy Net[current_state] # exploitation
```

7.2.2 Safety Layer- Protection of prior Knowledge

In Reinforcement Learning, the agent's knowledge resides in its state-action values, often denoted as Q-values. These Q-values encapsulate the agent's learned expectations of the cumulative rewards associated with specific actions in particular states. The Q-update rule in DQN and tabular Q-learning, utilizing a bootstrapping technique, is expressed as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Refer to Formula 4.1.24 in Chapter 4 for details.

This technique involves estimating the value of the current state-action pair based on the expected rewards of the subsequent state-action pair. While effective for iterative learning, an overreliance on bootstrapping and the estimate of the Q-value of the next state, $Q(s_{t+1}, a)$ in Q-learning can inadvertently erase or distort the agent's previously acquired knowledge. Thus, safeguarding this acquired knowledge becomes necessary to ensure the safety of RL training.

The method proposed in this thesis for preserving prior knowledge encoded in Q-values involves selectively blocking updates to the agent within predefined states. In this context, the state comprises signals that indicate the safe margin (see section 7.3). Therefore, the proposed method can block Q-updates in certain states or when any of the comprising signals indicate that the safety margin has been reduced.

The pseudo code for the algorithm to block Q-updates in certain states is presented below:

Algorithm 7.2.2.1 **Protection of prior Knowledge**

```
1.  function QValueUpdate (current reward, current_state, q_value, next_q_value):
2.  # Input: current reward, current_state, q_value, next_q_value
3.  # Output: Q-values
4.  # Set of states where Q update must be blocked
5.  setOfStatesToBlockQUpdates = {state1, state2, state3, ...}
6.  if current_state belongs to setOfStatesToBlockQUpdates:
7.          # Block Q-update in current states and return current q_value
8.          return q_value
9.  else:
10.         # Continue with Q-update rule
11.         return q_value + alpha * (reward + gamma *
                        max_next_q_value(next_q_value) -  q_value)
```

The proposed method in this section effectively blocks updates to Q-values in specific states to safeguard prior knowledge. Similarly, it can be applied to policy gradient RL methods, blocking policy updates in the same manner.

7.2.3 Safety Layer- Prevent Unsafe Actions[22]

Unsafe actions that lead to catastrophic events must be prevented during training and after deployment. Given the paramount importance of safety in autonomous driving (AD), this thesis adopts a solution that promptly intervenes to block the actions of the reinforcement learning (RL) agent. This intervention directly overrides the RL agent's policy when safety Key Performance Indicators (KPIs), and the safety margin identify unsafe regions within the state space.

The state space is classified into safe and unsafe regions, differentiating between the safety context and the RL context, as outlined in **safety margin scheme 7.3**.

Following this safety margin scheme, the management of unsafe regions is exclusively conducted within the safety context, entirely bypassing the RL context. The proposed solution in this thesis includes a rule-based strategy within the safety context, integrating clear logic for signals contributing to safety KPIs and relevant threshold parameters. This approach ensures immediate corrective actions to enhance the safety margin in real-time, independent of reliance on the agent's policy, thereby prioritizing effective safety responses.

<u>Examples of rule-based logics to prevent unsafe actions within the safety context:</u>

- ***If*** *Ego-Target Distance: (d1)* **<=** *d1_threshold,* ***then*** *throttle pedal ratio = 0*
  unsafe action of accelerating the vehicle is prevented
- ***If*** *Ego-Lane Distance:(d3)* **<=** *d3_threshold,* ***then*** *steering angle = +20 degree*
  unsafe action of over/understeering to the right or steering to the left is prevented
- ***If*** *throttle pedal pressed* ***AND*** *brake pedal pressed,* ***then*** *throttle pedal ratio = 0*
  When the brake override system detects that both the brake and throttle pedals are engaged simultaneously, it gives priority to the brake command.

7.2.4 Safety Layer- Safety Dependent Policy Optimization

Enhancing the safety of RL agents often involves adopting a constrained policy optimization approach. Rather than allowing unrestricted updates to the policy network, the implementation of constraints on policy updates is a common practice. This technique is notably employed in RL methods such as Proximal Policy Optimization (PPO) (refer to Section 4.7.5) to ensure training stability and promote a more robust learning process.

The original PPO RL method is based on the divergence between the old and new policies. The methodology proposed in this dissertation enhances this constraint by introducing a safety aspect.[23] The safety margin-dependent constraint is designed to dynamically adjust the constraining clipping of the policy update based on the safety margins (see section 7.3) of the RL agent.

[22] Other resources such as [4], [5], and [6] have addressed the topic from a distinct perspective. The methodology proposed in this dissertation, however, takes a fundamentally different approach.
[23] Other resources such as [7], [8], and [9] have addressed the topic from a distinct perspective. The methodology proposed in this dissertation, however, takes a fundamentally different approach.

The main intuition is to encourage the agent to promptly return to a safer state as the safety margin diminishes, achieved by minimizing the impact of the update constraint.

The original PPO clipped surrogate objective function:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t, clip\ (\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}, 1-\in, 1+\in) \hat{A}_t]$$

Where $\hat{A}_t$ is the advantage function at time-step t for the taken action and the hyperparameter ∈ controlling the size of the clip for the policy update.
Refer to Formula 4.7.24 in chapter 4 for details.

Incorporating safety penalty to the clipped surrogate objective function:
Let M(s) be the penalty function that represents the safety margin of the agent in state: s

For example, if the distance of the ego-vehicle to the nearest obstacle is below a safe threshold, impose a penalty; otherwise, the penalty is zero. Safety KPIs and safety margins in the context of autonomous driving in this dissertation are elaborated in defined in section 7.2.1.

The safety-aware advantage function A (s, a) considering the penalty term: Penalty(M(s)) is defined as:

$$A(s,a) = R(s,a) - \lambda . Penalty\ M(s) \qquad (7.2.11)$$

$R(s,a)$ is the reward of action: a in state: s and λ is a hyperparameter controlling the strength of the safety penalty.

Adjusting the advantage based on the safety margin encourages actions that maintain or increase the safety margin.
Hence, the safety-margin-dependent surrogate objective function of an PPO RL agent in the context of autonomous driving can be defined as:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \left(R(s,a) - \lambda . Penalty\ M(s)\right),$$
$$clip\ (\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}, 1-\in, 1+\in)(R(s,a) - \lambda . Penalty\ M(s))] \qquad (7.2.12)$$

7.2.5 Safety Layer- Dynamic Reward Shaping

Reward shaping in Reinforcement Learning (RL) refers to the process of modifying and optimizing the rewards received from the environment to guide the agent, improving the agent's decision-making capabilities and overall performance across various environments.

Raw Reward from the environment is shaped by a reward function. Researchers have proposed numerous reward functions in this area of research, as exemplified in [3], [4], [10]. In their study [1] a linear reward function grounded in the error signal and the authors in [11] developed a reward function based on human preferences and subsequently optimized it to achieve optimal results.

The raw reward obtained from the environment, r_k, is generally a straightforward function, not necessarily complex. For instance, a reward of +1 might be assigned if, in the current time step, the autonomous driving (AD) vehicle is navigating without encountering any accidents.

The reward shaping approach in this dissertation adopts a distinctive methodology based on the safety Key Performance Indicators (KPIs) of autonomous driving, as proposed in Section 7.3. Refer to Figure 7.8 for visual representation.

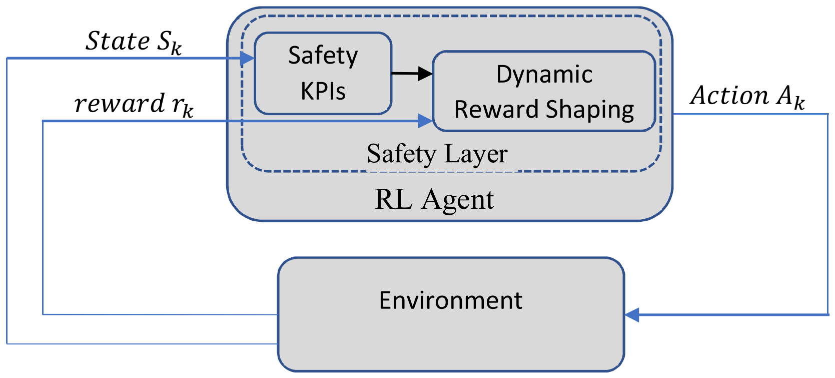


Fig. 7.8: Reward Shaping by RL Safety layer based on safety KPIs

Specifically, the safety layer of the Autonomous Driving (AD) agent imposes a negative reward (penalty) offset to the raw reward based on safety Key Performance Indicators (KPIs). This reward function aims to proactively prevent accidents and facilitate rapid transitions to safer driving states.

$$r_k \leftarrow r_k + RewardFunction\ (safety\ KPIs) \qquad (7.2.13)$$

The reward function for safety KPI: Ego-Lane Distance, defined in section 7.3, is proposed as follows:

<u>Reward shaping for lane keeping assist system (LKAS):</u>

Lateral Deviation: The lateral deviation(LD) is the horizontal distance between the centre of the vehicle and the centre of the lane. This deviation is measured perpendicular to the direction of the lane. If the vehicle is perfectly cantered, the lateral deviation is zero.

Mathematically, if P_{veh} is the position of the vehicle and P_{LC} is the centre of the lane (LC), the lateral deviation (LD) can be expressed as:
$d_{LD} = P_{veh} - P_{LC}$ (7.2.14)
A reduced form of the state in the context of a lane-keeping system can be represented by a combination of the lateral deviation (d_{LD})and the yaw rate (ψ).

This forms the input observation vector: state= [d_{LD} , ψ]

A reward function that encourages the agent to minimize lateral deviation while maintaining a reasonable yaw rate can be defined as:
$RewardFunction\ (Ego - Lane\ Distance) \doteq -\propto d_{LD}^{2} - \beta(\psi_{actual} - \psi_{target})^{2}$ (7.2.15)

Here, α and β are weighting factors, and ψ is the actual Yaw Rate. This reward function penalizes large lateral and yaw rate deviations.

7.2.6 Safety Layer- Redundant RL Agents

Multi-Agent Reinforcement Learning (MARL) is a subfield of reinforcement learning where multiple agents interact with an environment simultaneously. It is categorized into competitive versus cooperative and centralized versus decentralized, along with their hybrid variants [12]. A comprehensive overview of the applications, safety considerations, robustness, and generalization aspects of multi-agent reinforcement learning can be found in [13]. MARL methods have been successfully applied to a variety of safety-critical autonomous driving systems, including traffic light control [14], and lane change [15]. Authors in [16] introduce a multi-agent reinforcement learning model with hard constraints to ensure the functional safety of autonomous driving.

Building upon the foundation of MARL methods, this dissertation proposes a new reinforcement learning setting termed 'Redundant RL agents' to enhance the redundancy and safety of autonomous driving.

In the 'Redundant RL agents' setting, two RL agents with distinct RL algorithms, but identical tasks, state space, and action space function independently from each other and interact with the autonomous driving environment. This collaborative approach between the two agents can significantly enhance autonomous driving safety.

Safety Arbitration between two redundant RL agents:

In real-world driving scenarios, misalignments between two agents may arise due to various factors, including but not limited to differences in RL algorithms, policy networks, or training methodologies.

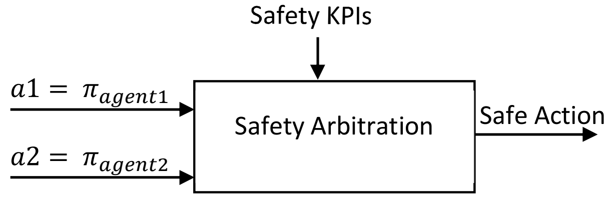


Fig. 7.9: Safety Arbitration between two RL agents considering safety KPIs for driving scenarios

Utilizing system redundancy, the safety arbitration method proposed here aims to resolve misalignments between the two redundant agents, ensuring the safety of decision-making. See Figure 7.9.

This is achieved by employing a **predefined arbitration strategy** that systematically evaluates Safety KPI metrics (refer to 7.3) and reconciles differences in the agents' outputs. The arbitration strategy is designed to prioritize safety-critical decisions, promoting a cohesive and reliable decision-making process in the autonomous driving environment.

Pre-established priority between the agents:

In a simplified model of autonomous driving with only actuators as: accelerator pedal brake pedal, steering wheel a pre-established safety arbitration mechanism can be defined as follows:

$a1 = \pi_{age}$ = acceleration ratio = A1%, $a2 = \pi_{agen}$ = acceleration ratio = A2%

Safety arbitration (A1%, A2%) = min (A1%, A2%)

$a1 = \pi_{agent1}$ = braking ratio = B1%, $a2 = \pi_{agen}$ = braking ratio = B2%

Safety arbitration (B1%, B2%) = max (A1%, A2%)

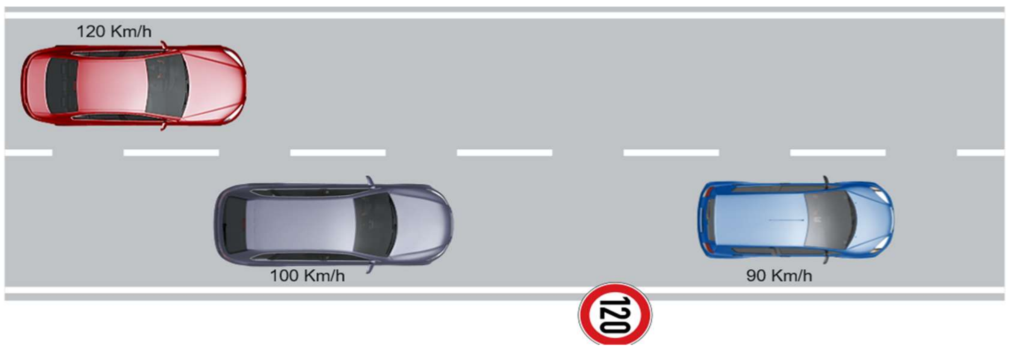


Fig. 7.10: Safety Arbitration between two RL agents for lane change or lane follow scenarios

$a1 = \pi_{agent1}$ = lane change, $a2 = \pi_{agen}$ = follow lane and slow down

Safety arbitration (lane change, follow lane and slow down) = follow lane and slow down

Algorithm 7.2.6.1 **Safety Arbitration between Redundant Agents**

```
function safety-arbitration (current_state, AgentPolicy1, AgentPolicy2):
    # Input: current_state, AgentPolicy1, AgentPolicy2
    # Output: Action selected by the safety arbitration
    # Check if safety KPIs exceed safety threshold 1
    if safetyKPIs(current_state) > safety_threshold_1:
        # Check if the policies of Agent 1 and Agent 2 differ for the current state
        if AgentPolicy1(current_state)! = AgentPolicy2(current_state):
            # Arbitrate actions for continuous actions (e.g., throttle and brake)
            arbitrated_throttle = min (AgentPolicy1.throttle(current_state), AgentPolicy2.throttle(current_state))
            arbitrated_brake = max (AgentPolicy1.brake(current_state), AgentPolicy2.brake(current_state))[24]
            # Arbitrate actions for discrete actions (e.g., steering)
            # Compute lateral deviation
            lateral_deviation = compute_lateral_deviation(current_state)
            # Arbitrate actions for steering based on lateral deviation
            if lateral_deviation >= 0:
                arbitrated_steering = min (AgentPolicy1.steering(current_state), AgentPolicy2.steering(current_state))
            elif lateral_deviation < 0:
                arbitrated_steering = max (AgentPolicy1.steering(current_state), AgentPolicy2.steering(current_state))
            # Output the arbitrated actions
            return (arbitrated_throttle, arbitrated_brake, arbitrated_steering)
    # If safety KPIs are at risk, no arbitration is performed
    return None
```

[24] In some driving scenarios unintended braking might pose a safety threat, and therefore, maximizing brake is not the safest action.

7.2.7 Safety Layer- Human Driver in Loop (Human Imitation)

There is[25] extensive research dedicated to this subject area; however, the testing and implementation of this specific safety feature is beyond the scope of this thesis. In the future, the author intends to extend the capabilities of the proposed solution, by practically deploying this feature as an important factor for practical solutions provided in this domain to industry.

In the context of autonomous driving, integrating human imitation into reinforcement learning (RL) presents a valuable avenue for improving the capabilities of RL agents. Post-deployment, the agent's policy must be adaptable and subject to continuous updates due to various factors such as the aging of the vehicle, variations in road conditions, changes in vehicle components like tires, braking systems, and suspension. In such cases, the agent should learn from human drivers, particularly when their decisions prove superior to the pre-existing policies stored by the agent.

The presented approach involves the RL agent entering a silent mode when a human driver assumes control. In this mode, the RL agent functions as a passive observer of the environment and human's actions.

To learn effectively from human drivers, it is essential to assess the merit of updating an agent's policy with human actions. If the reward received for the human's action is notably positive and the safety KPIs are improved by the action, the policy parameters are adjusted to align the agent's actions in the corresponding state.

The policy update occurs under the following conditions:

- The human's action differs from the agent's action.
- The reward received for the human's action surpasses a predefined threshold.
- Safety Key Performance Indicators (KPIs) show improvement.

<u>Policy update for a DQN agent based on the reward from the human's action:</u>

The Q-update, utilizing the Bellman Equation and Q-Learning principles (see formula (4.1.24) in chapter 4), ensures that the agent captures the impact of proper human actions in the long term:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R_{Human} + \gamma maxa'Q(s',a') - Q(s,a)) \tag{7.2.16}$$

Where, R_{Human} represents the reward to the human's action.

Similarly, in policy gradient-based RL, the policy network is updated to align the agent's action with the human's action for the same state.

[25] Other resources such as [1], [11], [17...19] have addressed the topic from a distinct perspective. The methodology proposed in this dissertation, however, takes a fundamentally different approach.

Policy update for a PPO agent based on the reward from the human's action:

In the PPO update equation, the adjustment of the policy parameters r_t(θ) is guided by the comparison of the current policy's probability to the old policy's probability (for the taken action. The update rule, as detailed in formula (4.7.26) in Chapter 4, is mathematically expressed as:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r(\theta)\,\hat{A}_t\,, clip\,(r(\theta)\,, 1-\in, 1+\in)\,\hat{A}_t]$$

The key factor influencing this adjustment is the advantage function (A_t), which reflects how much better or worse the current action is compared to the average action in a given state.

To include a signal from the human in the PPO update, the algorithm can be modified to incorporate the reward term specific to the human's action.

The approach that is proposed in this dissertation is to directly add the observed rewards as a signal to the advantage function as:

$$A_t = Q_t(s,a) - V_t(s) + \lambda \cdot R_{Human} \tag{7.2.17}$$

Here, R_{Human} is the reward obtained by the human's action and λ is a weighting factor.

This modification directly adds the reward for the human's action to the advantage function. In practical terms, this means that the advantage of the human's action will be adjusted by the additional reward (R_{Human}).

Consequently, during the PPO update, the policy will be adjusted to maximize the advantage, considering the specific reward associated with the human's actions.

7.2.8 Safety Layer- Fail- Safe Strategy

Ensuring safety in a self-driving car system, particularly in a fail-safe scenario where the Operational Design Domain (ODD) is not maintained, or there are hardware (HW) or communication failures, impaired sensor data, etc., poses a critical challenge that demands an effective fail-safe strategy.

The following is a non-exhaustive list of risks that an RL agent in the context of autonomous driving may encounter:

- HW failure in the system (ISO26262 safety standard)
- Communication failures (ISO26262 safety standard)
- Unavailable or impaired sensor data (performance deficiency in SOTIF ISO21448 safety standard)
- impaired actuator functionality (performance deficiency in SOTIF ISO21448 safety standard)
- Detection of unknown driving scenarios (SOTIF ISO21448 safety standard)
- Inability to verify the presence of a human driver (misuse in SOTIF ISO21448 safety standard)
- Operational Design Domain (ODD) conditions are not met
- Safety KPIs reaching a non-compensable level (RL agent failed to maintain a safe margin for the Ego vehicle)
- …

Considering the severity and potential catastrophic consequences of such system failures, this thesis advocates for a practical approach to handling fail scenarios within the safety context, entirely bypassing the RL context (see 7.3) to avoid an irrational reliance on the RL decision-making capability in the presence of a system failure.

This approach involves following clear logical operations and immediately transitioning the system to a predefined safe state within individual fault tolerance time intervals (FTTI).

Cyclic and real-time plausibility checks of input data, along with the continuous monitoring of ODD conditions and Safety Key Performance Indicators (KPIs), are prerequisite conditions for the success of the fail-safe strategy, enhancing the system's reliability and responsiveness.

Furthermore, the fail-safe strategy must strictly adhere to the relevant normative requirements of automotive safety standards[26]. It should include an appropriate strategy for promptly and effectively involving the human driver, ensuring a smooth and safe degradation of autonomous driving.

[26] Chapter 6 of this dissertation introduces a safety framework detailing how ML solutions can comply with automotive safety standards.

7.3 Safety Layer- Safety Margin

7.3.1 Safety Key Performance Indicators(KPIs)

The safety layer of the RL agent determines the safety margin by constantly assessing the environment based on five key performance indicators (KPIs)for safety. These metrics define the distance relationships and reaction capabilities critical for preventing accidents and ensuring the overall safety of the RL agent's operation.

<u>Ego-Target Distance: (d1)</u>

The space maintained between the autonomous vehicle and the vehicle in front (target vehicle). A smaller safety margin here could indicate a higher risk of collisions (unsafe).

Ego-Target Distance can be computed using sensor data such as radar or LIDAR, providing real-time distance information.

<u>Ego-Obstacle Distance: (d2)</u>

The minimum distance maintained from stationary obstacles, such as parked cars, barriers, or roadside objects. A smaller safety margin in this metric indicates an increased risk of collisions with nearby objects.

<u>Ego-Lane Distance:(d3)</u>

Evaluates the lateral space maintained within the lane. A narrower margin increases the likelihood of unintended lane departures, posing a safety risk.

Determining the Ego-Lane Distance is performed by analysing the vehicle's position in relation to the lane boundaries identified by camera-based Electronic Control Units (ECUs).

<u>Reaction Time Buffer: (t1)</u>

Reflects the time it takes for the RL agent to react to a perceived threat, unexpected events or changing situation. A reduced buffer may compromise the agent's ability to respond promptly to emerging risks.

<u>Severity of a Potential Accident: (s1)</u>

Quantifies the potential severity of an accident based on the relative speeds, angle of collision, and mass of the vehicles involved. This metric considers the consequences of various safety-related scenarios.

7.3.2 Safety goals

In alignment with the defined safety KPIs, the following safety goals are set for the autonomous driving (AD) agent.

<u>Collision Avoidance:</u>

Safety Goal: The autonomous driving (AD) agent must avoid collisions with other vehicles, pedestrians, and obstacles by maintaining a safe distance to other vehicles (d1) and stationary obstacles (d2).

Lane-Keeping and Steering Control:

Safety Goal: The AD agent should maintain accurate and safe control over the vehicle's steering, ensuring it stays within designated lanes and avoids unintended lane departures. The system must respond appropriately to sudden changes in road curvature and unexpected obstacles.

Speed Management:

Safety Goal: The AD agent must adhere to posted speed limits and adjust the vehicle's speed according to traffic conditions and road characteristics. It should prioritize smooth acceleration and deceleration to prevent abrupt manoeuvres that could lead to accidents (t1).

Adaptive Driving Behaviour:

Safety Goal: The AD agent should adapt its driving behaviour to the prevailing traffic conditions, weather, and road circumstances. It should avoid aggressive driving, tailgating, and other behaviours that may compromise safety (s1).

Emergency Braking:

Safety Goal: The AD agent should be capable of recognizing imminent collision scenarios and engage emergency braking when necessary. This ensures the vehicle can rapidly decelerate or stop to avoid or mitigate the severity of potential accidents (t1).

7.3.3 Safety margin scheme - Safety context versus RL context[27]

In the proposed safety margin scheme within the RL agent's safety layer, the autonomous driving agent continuously monitors safety key performance indicators (KPIs) and determines the safety criticality of the autonomous driving. The assessment of safety criticality is based on predefined individual thresholds for each of the five safety metrics, referred to as Threshold1 and Threshold2.

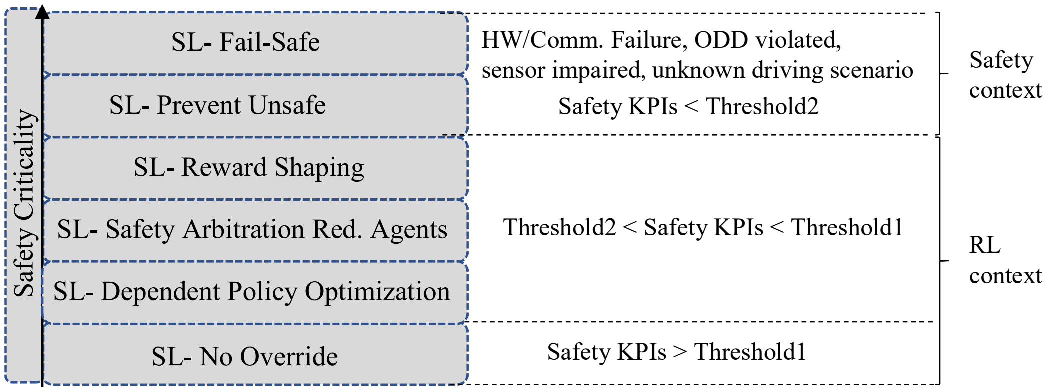

Safety Criticality
SL- Fail-Safe
SL- Prevent Unsafe
SL- Reward Shaping
SL- Safety Arbitration Red. Agents
SL- Dependent Policy Optimization
SL- No Override
HW/Comm. Failure, ODD violated, sensor impaired, unknown driving scenario
Safety KPIs < Threshold2
Threshold2 < Safety KPIs < Threshold1
Safety KPIs > Threshold1
Safety context
RL context

Fig. 7.11: Comprehensive Safety Margin Scheme and Types of Safety Layer Interventions

Safety Enhancement within the Reinforcement Learning (RL) Context:

[27] An example of safety and RL context implementation is presented in Chapter 8, Section 8.4.

In the context of reinforcement learning (RL), when safety KPIs are larger than their Threshold1, the reinforcement learning (RL) agent avoid executing any safety interventions. Alternatively, when any safety KPI falls between defined Threshold1 and Threshold2, the RL agent dynamically engages at least one of the proposed safety-layer measures. These measures include safety arbitration between two agents, application of reward shaping, or dependent policy optimization, as detailed in Section 7.2. The choice of safety-layer measure depends on the applied RL method and the current status of autonomous driving. See Figure 7.11.

Mathematically, this condition can be expressed as follows:

$$SM\_RLContext \ = \ argmin(Safety_{KPIi}) \geq SM\_Threshold2_i \qquad (7.3.1)$$

$i = 1, \dots, N$ where N is the number of relevant KPIs

Safety Enhancement within the Safety Context:

In the safety context of the proposed safety margin scheme, if any of the safety KPIs falls below a non-compensable Threshold2, signalling an increased safety risk, the safety layer takes immediate control to prevent unsafe actions.

Mathematically, this condition can be expressed as follows:

$$SM\_SafetyContext \ = \ argmin(Safety_{KPIi}) < SM\_Threshold2_i \qquad (7.3.2)$$

$i = 1, \dots, N$ where N is the number of relevant KPIs

Moreover, in cases of hardware or communication failures, violations of Operational Design Domain (ODD), sensor impairments, or encountering unknown driving scenarios, a fail-safe strategy is promptly implemented, leading to the degradation of the autonomous driving system to ensure heightened safety measures. See Figure 7.11.

The safety margin scheme presented in this thesis guarantees a transparent strategy for autonomous driving safety. It places a high priority on accident prevention while ensuring optimal operational safety.

8 Chapter 8 – Implementing, validating, and Verifying Safe Highway Driving RL agent with a safety layer

8.1 Introduction

Chapter 8 outlines the experimentation framework for integrating the proposed Safety Layer (SL) into state-of-the-art Reinforcement Learning (RL) agents (DQN, PPO, A2C, and DDPG) within the customized Highway-Env environment, focusing on highway driving scenarios. The process involves training RL agents initially without safety layer features. Following the training phase, a systematic quantitative comparison is conducted. In addition to simulating SL-Enhanced training in chapter 7, three further key features of the safety layer, namely 'SL-Redundant RL Agents', 'SL-Prevent Unsafe Actions' and 'SL-Safety Dependent Policy Optimization' are demonstrated in this chapter. These features are suggested and the algorithms created in Chapter 7. The results of this chapter, successfully demonstrate that the selected safety features enforced make huge impact on the enhancement of autonomous driving safety.

Two safety Key Performance Indicators (KPIs) are established, aligning with the safety margin scheme proposed in Chapter 7 (Section 7.3). These KPIs serve as crucial metrics for evaluating the safety performance of RL agents in different driving scenarios. The analysis explores the impact on safety metrics when specific Safety Layer features are either activated or deactivated. This chapter intends to validate and evaluate the proposed solution by experiment, analysing the impact on autonomous driving safety influenced by the activation or deactivation of Safety Layer features. Insights derived contribute to ongoing efforts aimed at enhancing autonomous driving safety.

This chapter wraps up by summarizing findings and outlining potential avenues for future research and development within the domain of autonomous driving safety.

8.2 'Highway-env' Simulation Environment

'Highway-Env'[28][1] is an open-source Python reinforcement learning environment designed for training and testing autonomous vehicles in various traffic scenarios. The environment provides a framework for creating diverse driving scenarios including Intersection, Lane Keeping, Roundabout, Parking, Highway, and more.

In this chapter of the dissertation, the 'highway-env' environment is employed to simulate highway driving scenarios. See Figure 8.1.

[28] Citation: @misc {highway-env, author = {Leurent, Edouard}, title = {An Environment for Autonomous Driving Decision-Making}, year = {2018}, publisher = {GitHub}, journal = {GitHub repository}, howpublished = {\url{https://github.com/eleurent/highway-env}},}

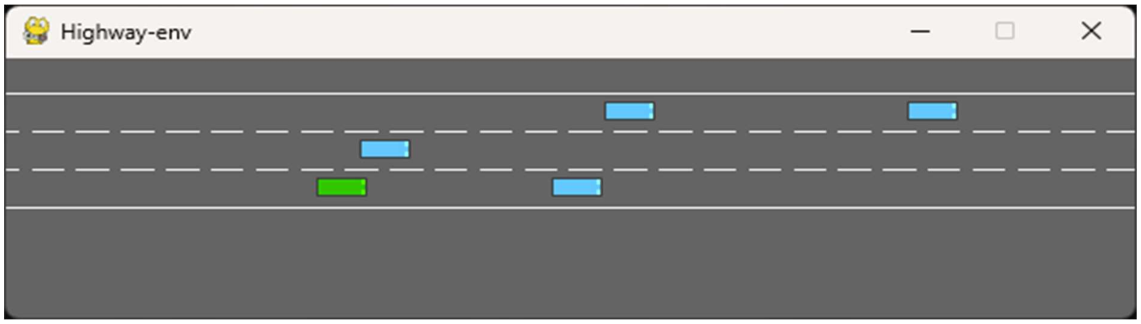


Fig. 8.1: Highway-Env Simulation - Configured for Highway Driving

Figure 8.1. presents a snapshot of the highway-env environment specifically configured for highway driving. The simulation environment consists of three lanes, with the ego vehicle highlighted in green, representing the autonomous vehicle under study. Additionally, four target vehicles, denoted by the blue markings, populate the highway.

Action Space:

The environment can be customized to have either continuous and discrete action spaces. In the context of this dissertation, the environment is configured with a discrete action space.

Actions: = ["lane left", " lane right", " faster'", " slower'", " idle"]

State (Observation) Space:

The observation provided by the highway-env environment is structured as an occupancy grid, incorporating specific features that denote the presence, position, and velocity of both the ego vehicle and other target vehicles.

The Observation is a VxF (vehicle, feature) array that describes a list of V nearby vehicles by a set of features of size F, listed in the "features" configuration field.

Features: = ["presence", "x [m]", "y[m]", "vx [m/s^2]", "vy[m/s^2]"]

The observation info with respect to the ego-vehicle is always described in the first row of the array. The coordinates of the Ego vehicle are always absolute.

The observation information for target vehicles can be configured either as absolute values or relative to the ego vehicle. When configured with normalized=True (default), the observation is normalized within a fixed range for each coordinate. Refer to Table 8.1 (a) and (b).

Vehicle	**x**	**y**	**vx**	**vy**
ego-vehicle	5.0	4.0	15.0	0
vehicle 1	-10.0	4.0	12.0	0
vehicle 2	13.0	8.0	13.5	0
….	...	...	...	...
Vehicle V	22.2	10.5	18.0	0.5

Or

Vehicle	**x**	**y**	**Vx**	**vy**
ego-vehicle	0.05	0.04	0.75	0
vehicle 1	-0.1	0.04	0.6	0
vehicle 2	0.13	0.08	0.68	0
….	...	...	...	...
Vehicle V	0.22	0.11	0.9	0.03

Table 8.1: An example of highway-env observations in (a) absolute or (b) relative configuration

In summary, the RL agent observes the state regarding the coordinates of Ego and target vehicles and takes actions as illustrated in Figure 8.2.

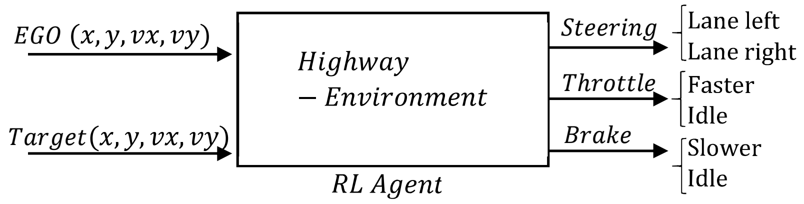


Fig. 8.2: RL agent in Highway-Env Simulation – States and Actions

Reward Function:

The reward function of the Highway-env environment is designed to encourage the agent to drive faster on the road and avoid collisions. The reward is the sum of the following reward values:

Rewards: = ["Collision Reward" (-1), " Right Lane Reward" (0.1), " High Speed Reward" (0.4), "Lane Change Reward" (0)]

Reward	Value	Explanation
Collision Reward	-1	The reward received when colliding with a vehicle.
Right Lane Reward	0.1	The reward received when driving on the right-most lanes, linearly mapped to zero for other lanes
High Speed Reward	0.4	The reward received when driving at full speed, linearly mapped to zero for lower speeds
Lane Change Reward	0	The reward received at each lane change action

Table 8.2: Rewards in Highway-Env Highway driving scenario

Environment Configuration:

Highway-env is highly customizable and offers a wide range of configuration parameters. Interested readers can refer to the highway-env documentation for more insights. In the context of this dissertation, default configurations are utilized. Some of the highlighted configurations include:

Action type: Discrete Action, road type: straight, episode duration: 40 [s], Vehicle length: 5.0 [m], Vehicle width= 2.0 [m], MAX_SPEED = 40.0 [m/ s^2], …

8.3 Training State-of-the-Art RL Agents on Highway-env

The training of various state-of-the-art Reinforcement Learning (RL) agents, namely DQN, A2C, PPO, and DDPG, using the highway-fast-v0 environment under identical conditions are discussed. The focus is on assessing their performance, saving the trained models, and generating Tensorboard reports for further analysis.
For the implementation of the RL agents Stable-baseline3 (SB3) library [2] is used. Stable-baseline3 (SB3) library is a set of reliable implementations of reinforcement learning algorithms in PyTorch.
GitHub repository: https://github.com/DLR-RM/stable-baselines3

Training Setup
The training process involves executing these RL agents on the highway-fast-v0 environment, capturing default settings for thorough comparison. Models are saved periodically, and Tensorboard reports are generated every 20,000 training time steps. These saved models and Tensorboard reports are revisited to monitor the agents learning progress and efficiency of the training process.

Configuration Parameters
To ensure a meaningful basis for comparison, all four RL agents (A2C, DQN, PPO, and DDPG) are configured with a Fully Connected MLP network as their policy model. Apart from the MLP network, other configuration parameters are predominantly set to their default values.
In Chapter 5 the impact of different design variations of RL agents and the influence of RL configuration parameters and neural network setup on the performance and stability of RL training were explored

Performance Evaluation
In the next step, the last trained model is reloaded and set to run a specific number of episodes. The performance of RL agents is analysed to identify the most effective models. The recorded video showcasing the RL agent's performance in MP4 format, submitted as a supplementary document, can be viewed and downloaded.
In the following sections, the trained RL models are selectively used and combined with the proposed safety-override layer for experimentation and evaluation of the impact of the safety layer features on autonomous driving safety. The goal is to objectively assess whether the proposed safety features can effectively enhance safety.

8.3.1 Highway-env: RL DQN Agent

The DQN reinforcement learning algorithm is comprehensively reviewed in Chapter 4, Section 4.6 and the major design variations to the algorithm were implemented and evaluated in Chapter 5.
Here, the major configuration parameters for DQN are set according to the code snippet below.

```
# Instantiate the agent
model = DQN('MlpPolicy', env,
            policy_kwargs=dict(net_arch=[256, 256]),
            learning_rate=5e-4,
            buffer_size=15000,
            learning_starts=200,
            batch_size=32,
            gamma=0.8,
            train_freq=1,
            gradient_steps=1,
            target_update_interval=50,
            verbose=1,
            tensorboard_log=tensorboard_log)
```

8.3.2 Highway-env: RL PPO Agent

The PPO reinforcement learning algorithm is comprehensively reviewed in Chapter 4, Section 4.7.5. The major configuration parameters for PPO are set according to the code snippet below.

```
# Instantiate the agent
model = PPO('MlpPolicy', env,
            policy_kwargs=dict(net_arch=[256, 256]),
            learning_rate=2e-3,
            batch_size=64,
            verbose=1,
            tensorboard_log=tensorboard_log)
```

Other parameters, including the major configuration parameters:
{'policy': MlpPolicy, 'learning_rate', 'batch_size', 'gamma', 'clip_range', 'n_epochs',' max_grad_norm', …}, are set to default values.

8.3.3 Highway-env: RL A2C Agent

The A2C reinforcement learning algorithm is comprehensively reviewed in Chapter 4, Section 4.7.6. A2C parameters, including the major configuration parameters: {'policy': MlpPolicy, 'learning_rate', 'n_steps', 'vf_coef', 'gamma',' max_grad_norm', 'lr_schedule', …}, are set to default values.

```
# Instantiate the agent
model = A2C('MlpPolicy', env,
            verbose=1,
            tensorboard_log=tensorboard_log)
 Highway-env: RL DDPG Agent
```

8.3.4 Highway-env: RL DDPG Agent

The DDPG reinforcement learning algorithm is comprehensively reviewed in Chapter 4, Section 4.7.7. As DDPG is specifically designed for environments with continuous action spaces, it is necessary to adjust the configuration of the environment from a discrete type of action space to a continuous one.

```
# Create environment
env = gym.make("highway-fast-v0", render_mode='rgb_array')
env.config["normalized"] = False
env.configure({"action": {"type": "ContinuousAction"}})
env.reset()
```

In the continuous action space configuration, the RL agent controls the throttle and steering angle within the following ranges:
ACCELERATION_RANGE = (-5.0, 5.0) m/s².
STEERING_RANGE = [-45.0, 45.0] degrees

DDPG parameters, including the major configuration parameters:
{'policy': MlpPolicy, 'learning_rate', 'batch_size', 'tau', 'gamma', 'train_freq', 'gradient_steps', 'action_noise', 'replay_buffer', …}, are set to default values.

```
# Instantiate the agent
model = DDPG('MlpPolicy', env, verbose=1,tensorboard_log=tensorboard_log)
```

8.3.5 Comparison of RL Agents' Performance on Highway-Env Environment

Training results of A2C, DQN, PPO, and DDPG Agents on Highway-env Driving

In the examination of RL agents on the Highway-Env environment, two key metrics were evaluated: average episode reward and average episode length over 40,000 time steps. As shown in Figures 8.3(a) and 8.3(b), PPO emerged with the most favourable outcomes, securing the top position. Following closely, DQN demonstrated solid performance, securing the second position. On the other hand, DDPG and A2C exhibited comparatively less satisfactory results in this evaluation.

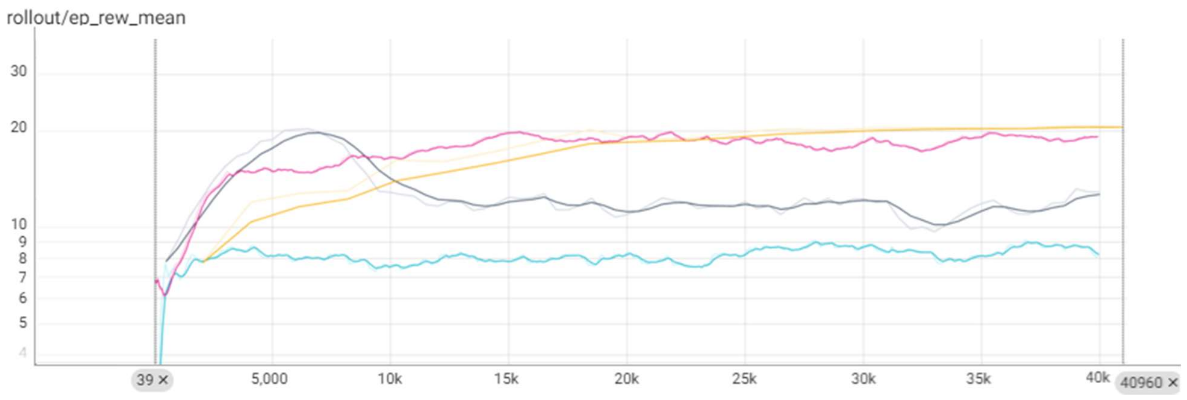


Fig. 8 8.3(a): Highway-Env - Agent Training(PPO, DQN, A2C, DDPG) - Average Episode Reward

Run	Min	Max	ΔValue
● Highway_PPO	7,7545	20,56	12,8054
● Highway_DQN	6,135	19,8481	12,324
● Highway_DDPG	0,9199	8,9891	7,3247
● Highway_A2C	7,8212	19,7452	4,8636

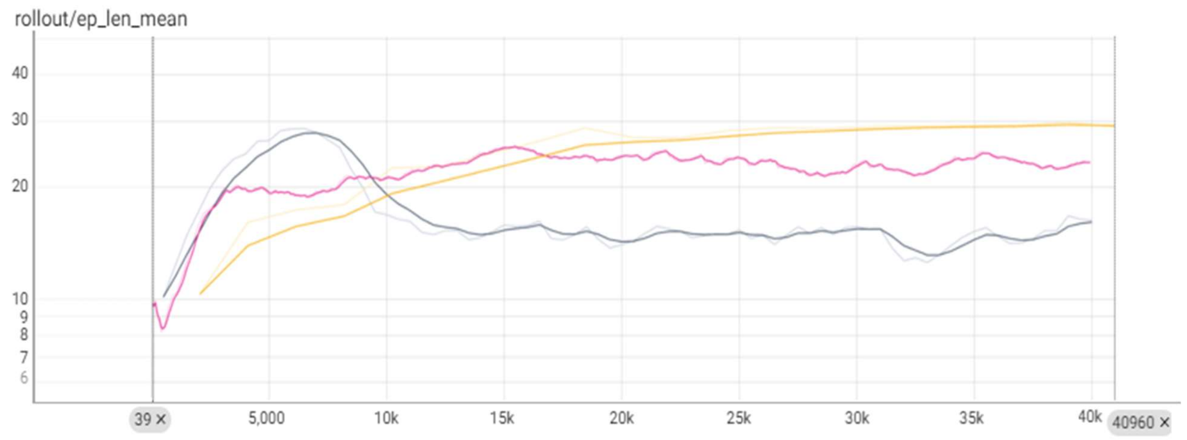


Fig. 8.3(b): Highway-Env - Agent Training(PPO, DQN, A2C, DDPG) - Average Episode length

Run	Min	Max	ΔValue
● Highway_PPO	10,32	29,39	18,926
● Highway_DQN	8,3567	25,6788	13,5703
● Highway_A2C	10,1429	27,947	5,9734

Performance evaluation of A2C, DQN, PPO, and DDPG Agents on Highway-env Driving

To evaluate the performance and safety aspects of driving, the final trained models—A2C, DQN, and PPO—were utilized in a comprehensive assessment comprising 100 episodes of highway driving within the highway-env environment. The maximum duration configured for each episode was set at 40 seconds by default, aiming to replicate realistic driving scenarios.

The observed behaviours of these agents exhibited striking differences, showcasing distinct driving strategies and safety outcomes. A2C, despite its functionality, displayed a tendency to drive unsafely, experiencing 95 crashes out of 100 episodes. Furthermore, it received the least rewards compared to its counterparts, as illustrated in Figure 8.4(a) and (b). Notably, A2C agent demonstrated a consistent reluctance to change lanes or slow down, contributing to its higher collision rate and decreased overall performance.

Conversely, the DQN agent showcased a distinctive driving style characterized by higher speeds and repeated overtaking manoeuvres compared to other agents.This resulted in the agent receiving maximum speed rewards from the environment. While this approach led to enhanced speed-related rewards, it also presented challenges in terms of safety, with the agent experiencing a notable number of 24 crashes.

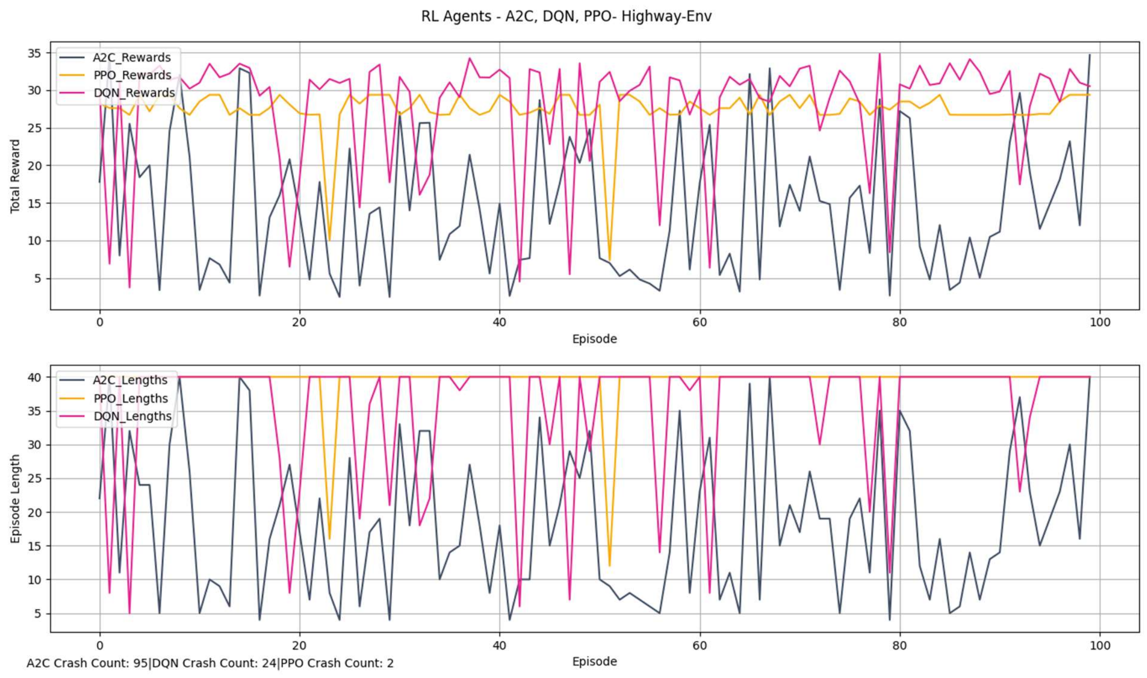


Fig. 8.4: Highway-Env - Agent Performance (PPO, DQN, A2C) - Reward (a) and Episode Lengths (b)

On the other hand, the PPO agent demonstrated a balanced approach between speed and safety. Despite driving at a slightly slower pace than DQN, the PPO agent received commendable rewards. Remarkably, PPO exhibited impressive driving safety by consistently maintaining a safe distance from the front vehicle, contributing to its exemplary safety outcomes. Throughout the 100 episodes, the PPO agent encountered only two crashes, specifically in episodes 23 and 51. The episode length of PPO, shown in Figure 8.4(b), further emphasizes its consistent performance and safety record, reaching the configured maximum duration of 40 seconds[29].

[29] The video showcasing the RL agent's performance in MP4 format, submitted as a supplementary document, can be viewed and downloaded

8.4 Safety Layer-Safety Margin

8.4.1 Highway Driving Safety KPIs from Agent Observations

Safety KPIs based on the longitudinal and lateral distances between EGO and Target vehicles:

In alignment with the safety margin scheme proposed in Chapter 7, specifically outlined in Section 7.3, two safety Key Performance Indicators (KPIs), applicable to the 'Highway-Env' environment, are defined as follows:

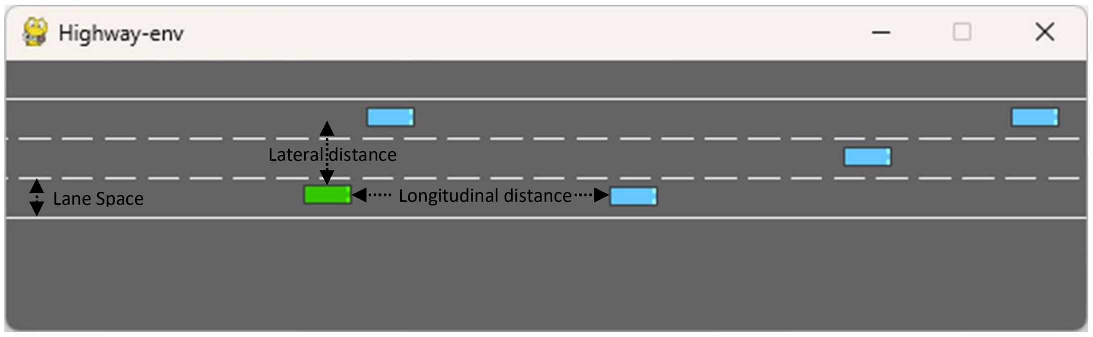


Fig. 8.5: Highway-env - Definition of Lateral and Longitudinal Distance Arrays

Safety_KPI1 and Safety_KPI2 are characterized by their dependency on both lateral (D_{lat}) and longitudinal (D_{long}) distance arrays, representing the lateral and longitudinal distances between the Ego and target vehicles, respectively.

Longitudinal Distance Array:

The Longitudinal distance array, denoted by D_{long}, is defined as the distance between the ego vehicle and the target vehicles , regardless of their y-coordinates. For each target vehicle i, the longitudinal distance is computed as the difference in the x-coordinates between the ego vehicle and that specific target vehicle. Mathematically:

$$D_{long}[i] = X_{Ego} - X_{Target_i} \tag{8.4.1}$$

Where X_{Ego} is the x-coordinate of the ego vehicle, and X_{Target_i} is the x-coordinate of the target vehicle i.

The resulting D_{long} array provides the longitudinal distances for each target vehicle relative to the ego vehicle.

Lateral Distance Array:

Similarly, the Lateral distance array, D_{lat} , refers to the sideways distance between the ego vehicle and the target vehicles, irrespective of their x-coordinates, and is defined as:

$$D_{lat}[i] = Y_{Ego} - Y_{Target_i} \tag{8.4.2}$$

Where Y_{Ego} is the y-coordinate of the ego vehicle, and Y_{Target_i} is the y-coordinate of the target vehicle i.

The resulting D_{lat} array provides the lateral distances for each target vehicle relative to the ego vehicle.

Safety_KPI1: EGO-Target-Longitudinal-Distance

$$Safety\_KPI1[i] = \begin{cases} D_{long}[i] & ; if D_{lat}[i] < lanespace \\ MaxValue & ; else \end{cases} \tag{8.4.3}$$

Equation 8.4.3 presents the mathematical expression for Ego-Target-Longitudinal-distance metric, denoted as Safety_KPI1[i].

When Ego and $Target_i$ vehicles traverse the roadway within the same line, indicated by their lateral distance is less than a predefined lane space, then Safety_KPI1[i] is defined as the longitudinal distance between the two vehicles. Otherwise, Safety_KPI1[i] is assigned a max-value. The rationale for assigning a max-value in this scenario is that when EGO and target vehicles are not in the same lane, the risk of collision or accident significantly decreases.

The nearest target vehicle corresponds to the one with the minimum absolute value in $Safety\_KPI1[i]$ and is identified by:$argmin(Safety\_KPI1[i])$.

Safety_KPI1 serves as a crucial metric for quantifying the longitudinal safety distance between the ego and target vehicles, aligning with the safety goal of maintaining a safe distance from the front vehicle.

Safety_KPI2: EGO-Target-Lateral-Distance

$$Safety\_KPI2[i] = \begin{cases} D_{lat}[i] & ; if D_{long}[i] < d1 \\ \alpha D_{lat}[i] & ; if D_{long}[i] > d1 \\ MaxValue & ; if D_{long}[i] \gg d1 \end{cases} \tag{8.4.4}$$

Equation 8.4.4 defines the safety KPI for Ego-Target-Lateral-distance, denoted as Safety_KPI2[i].

When Ego and $Target_i$ vehicles traverse the roadway close to each other in the x-coordinate and their longitudinal distance, D_{long}, is less than a predefined threshold (d1), Safety_KPI2[i] is set as the lateral distance, D_{lat}, between the two vehicles.

If the longitudinal distance is larger less than d1, Safety_KPI2[i] is scaled by the hyperparameter α $(\alpha > 1)$. In cases where the vehicles are not in close proximity

$(D_{long}[i] \gg d1)$, Safety_KPI2[i] is set to a Max-value, signifying that the Ego vehicle is presently not at risk of a side collision.

The nearest target vehicle, in terms of lateral distance, corresponds to the one with the minimum absolute value in $Safety\_KPI2[i]$ and is identified by: $argmin(abs(Safety\_KPI2[i]))$.

Safety_KPI2 plays a crucial role as a key metric in evaluating the lateral safety distance between the ego and target vehicles, thereby contributing to the broader safety goal of maintaining a secure spacing from neighbouring vehicles.

8.4.2 Implementation of Safety KPIs in Highway-Env

The code snippet to implement the safety margin scheme, as per Section 7.3, and compute Safety KPIs for the Highway-env environment is as follows:

```
class SafetyMargin:
    def __init__(self, SM_Threshold_RLContext, SM_Threshold_safetyContext,
lanespace=3, alpha=2, longitudinalspace=5):
        self.lanespace = lanespace
        self.longitudinalspace = longitudinalspace
        self.alpha = alpha
        self.SM_Threshold_RLContext = SM_Threshold_RLContext
self.SM_Threshold_safetyContext = SM_Threshold_safetyContext

    def compute_safety_kpi1(self, D_long, D_lat):
        # Compute Safety_KPI1 based on the absolute value of D_lat
        return np.where(np.abs(D_lat) < self.lanespace, D_long, XRange)
    def compute_safety_kpi2(self, D_lat, D_long):
        # Compute Safety_KPI2 based on the absolute value of D_long
        condition0 = np.abs(D_lat) < self.lanespace
        condition1 = np.abs(D_long) > 2 * self.longitudinalspace
        condition2 = np.abs(D_long) < self.longitudinalspace
        condition3 = np.abs(D_long) > self.longitudinalspace
        return np.where(condition0 | condition1, YRange, np.where(condition2,
D_lat, self.alpha * D_lat))
```

8.4.3 Analysing Driving Behaviour of RL Agents utilizing Safety KPIs

In the comparative experimentation section (Section 8.3.5) involving A2C, PPO, and DQN agents within a highway driving environment, it was highlighted that the A2C agent drives unsafely, experiencing the maximum number of crash incidents. To gain insights into the safety aspects of an RL agent's driving behaviour, the Safety KPIs defined in this dissertation are utilized. The study aims to uncover patterns, contributing factors, and the correlation between variations in these safety metrics and the crash incidents of the RL agents.

<u>A2C Agent Performance and crash incident Analysis</u>

The trained A2C, DQN, and PPO models were loaded and run for 10 episodes, generating visualizations of Safety_KPI1 and Safety_KPI2, as well as crash incidents, throughout their respective highway driving time steps. The results of this experiment are presented in the Figures 8.6 (a), (b), (c) and 8.7 (a), (b), (c).

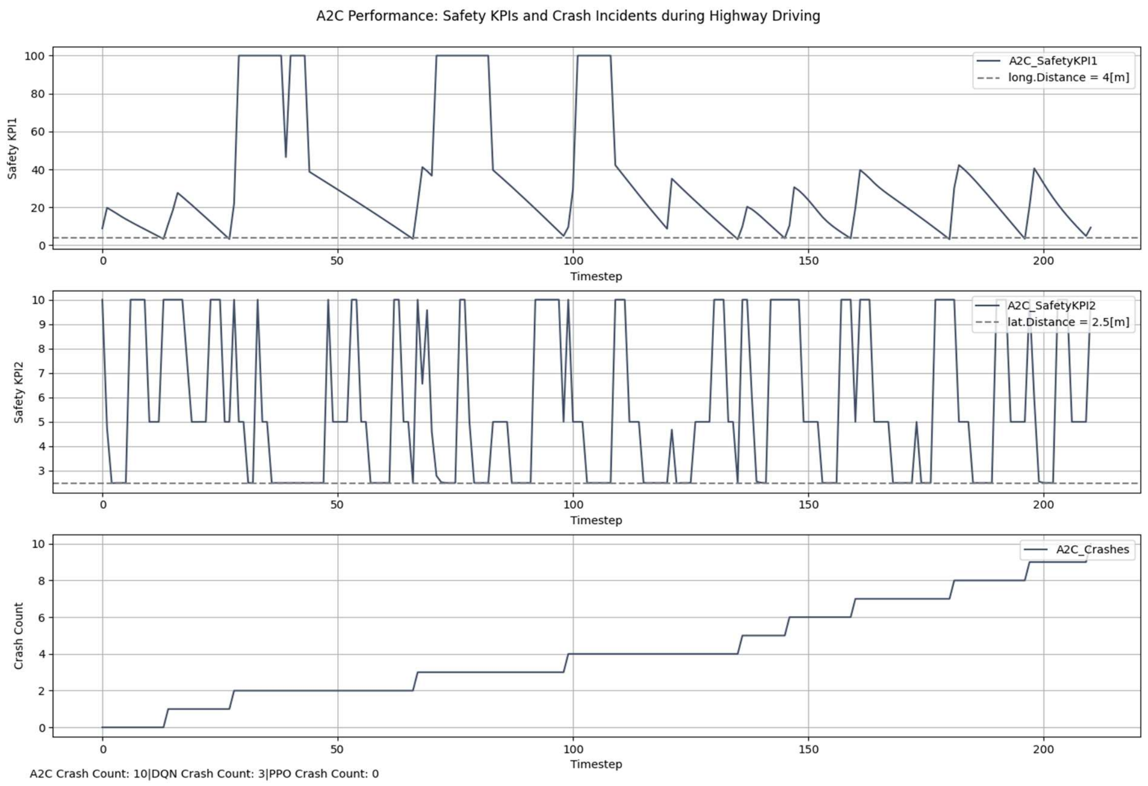


Fig. 8.6: A2C Performance: Safety KPIs (a), (b), crash incidents (c) in highway driving

<u>Correlating Safety KPIs with Crash Incidents</u>

A2C consistently encounters a crash incident in each episode, resulting in a total of 10 crashes, as illustrated in Figure 8.6 (c). In instances where the A2C agent fails to maintain a safe distance from the front vehicle, leading to Safety_KPI1 surpassing the critical longitudinal distance level (indicated by the dotted horizontal line) of 4 [m], the risk of colliding with the front target vehicle significantly increases, as depicted in Figure 8.6 (a).

Similarly, the potential for side collisions rises when Safety_KPI2 exceeds the critical lateral distance level (dotted horizontal line) of 2.5 [m], as shown in Figure 8.6 (b).

<u>Performance and crash incident Analysis of DQN and PPO Agents</u>

In contrast to A2C, both DQN and, notably, PPO consistently maintain a safe distance, as reflected in their higher levels of both lateral and longitudinal safety KPIs, as illustrated in Figure 8.7(a) and (b).

PPO, with a record of zero crashes, exhibits a commendable commitment to safe driving practices. Hence, the agent successfully navigates through the entirety of 10 episodes, each comprising 40 steps, resulting in a total of 400 time-steps.

In comparison, the driving behaviour of the DQN agent is characterized by a tendency to change lanes and overtake, as indicated by its smaller values of lateral safety KPI compared to PPO, as shown in Figure 8.7(b). The DQN agent, demonstrating a more assertive driving style, encounters 3 crashes over the course of 10 episodes, as depicted in Figure 8.7(c). The crossing of safety KPIs beyond critical thresholds is evident in these crash incidents (Figure 8.7(a)).

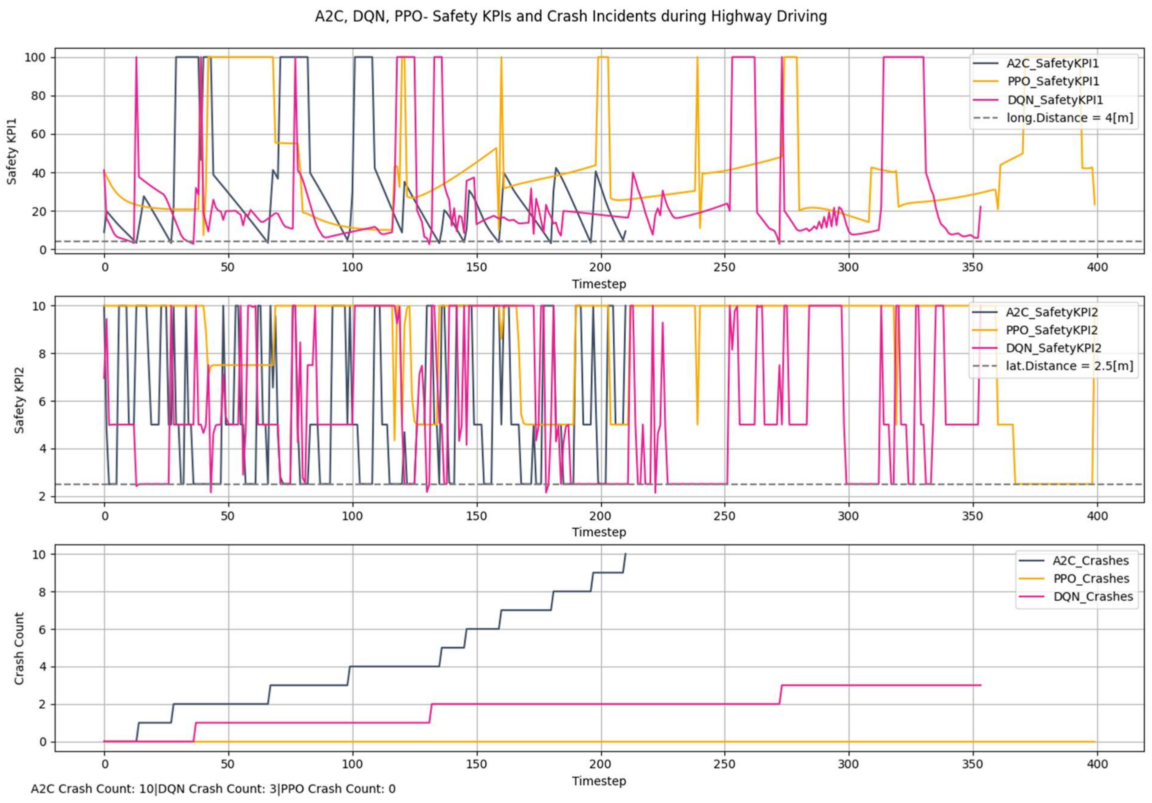


Fig. 8.7: Highway Driving - DQN and PPO Performance: Safety KPIs (a), (b), Crash Incidents(c)

The utilization of Safety KPIs in analysing the dynamic behaviour of RL agents, as presented in this dissertation, yields valuable insights into their driving performance and establishes a correlation between safety metrics and crash incidents in RL agents' highway driving dynamics. In the subsequent section, safety KPIs are further employed to delineate the distinction between 'safety context' and 'RL context,' following the safety margin scheme proposed in chapter 7. The type of safety layer intervention depends on the currently active context.

8.4.4 Safety Layer-RL vs Safety Context

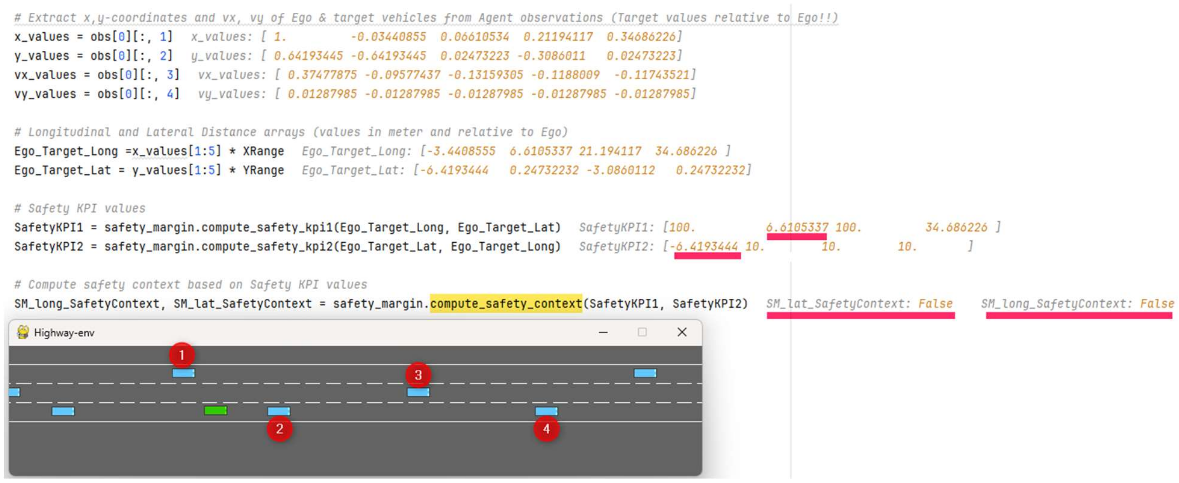
In accordance with the safety margin scheme proposed in Chapter 7, Section 7.3, the assurance of safety within autonomous driving scenarios relies on the fulfilment of predetermined safety KPIs. When the calculated value of a safety KPI falls below an established non-compensable threshold, denoted as Threshold2, the safety of the autonomous driving system can be ensured within the defined 'safety context.'

The validation of this safety context is formally represented by Formula 7.3.2:

$$SM\_SafetyContext\ =\ argmin(Safety_{KPIi}) < SM\_Threshold2$$

```
# Compute safety context based on the minimum Safety_KPI values and SM_Threshold1
def compute_safety_context(self, Safety_KPI1, Safety_KPI2):
    # Find the index of the minimum value in each Safety_KPI array
    min_index1 = np.argmin(np.abs(Safety_KPI1))
    min_index2 = np.argmin(np.abs(Safety_KPI2))
    # Get the absolute values of the minimum values in each array
    min_value1 = np.abs(Safety_KPI1[min_index1])
    min_value2 = np.abs(Safety_KPI2[min_index2])
    # Compare the absolute values of the argmin values with the thresholds
    sm_safetycontext1 = min_value1 <  self.SM_Threshold_safetyContext[0]
    sm_safetycontext2 = min_value2 <  self.SM_Threshold_safetyContext[1]
    # Return the comparison results as arrays of boolean values
    return sm_safetycontext1, sm_safetycontext2
```

The results of the implementation, as depicted in Figure 8.8 (a) and (b), provide insights into the debugger information. In (a), Target Vehicle 1 is positioned -3.44 [m] behind the ego vehicle, with a lateral distance of 6.41 [m]. Target Vehicle 2 is traversing in the same lane and is located 6.610 [m] ahead. Consequently, Longitudinal Safety-KPI1 is set to 6.610 [m], and Lateral Safety-KPI2 is set to 6.41 [m]. Since both these KPIs are still above SM_Threshold2 [4 m, 4 m], the RL agent can handle the driving scenario within the RL context and the safety context is disabled (SM_long_SafetyContext = False, SM_lat_SafetyContext = False).

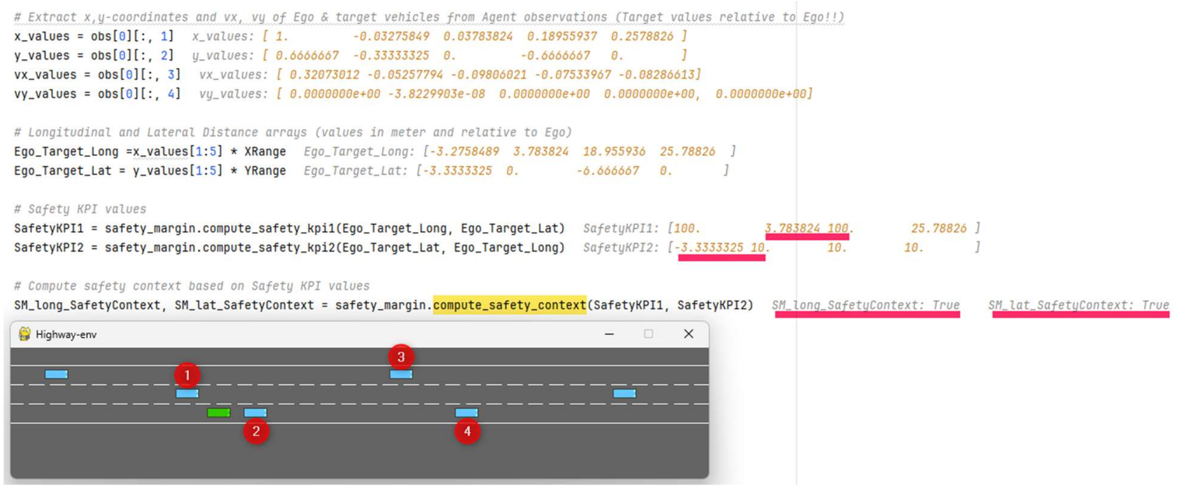


```
# Extract x,y-coordinates and vx, vy of Ego & target vehicles from Agent observations (Target values relative to Ego!!)
x_values = obs[0][:, 1]   x_values: [ 1.          -0.03440855  0.06610534  0.21194117  0.34686226]
y_values = obs[0][:, 2]   y_values: [ 0.64193445 -0.64193445  0.02473223 -0.3086011   0.02473223]
vx_values = obs[0][:, 3]   vx_values: [ 0.37477875 -0.09577437 -0.13159305 -0.1188009  -0.11743521]
vy_values = obs[0][:, 4]   vy_values: [ 0.01287985 -0.01287985 -0.01287985 -0.01287985 -0.01287985]

# Longitudinal and Lateral Distance arrays (values in meter and relative to Ego)
Ego_Target_Long =x_values[1:5] * XRange   Ego_Target_Long: [-3.4408555  6.6105337 21.194117  34.686226 ]
Ego_Target_Lat = y_values[1:5] * YRange   Ego_Target_Lat: [-6.4193444   0.24732232 -3.0860112   0.24732232]

# Safety KPI values
SafetyKPI1 = safety_margin.compute_safety_kpi1(Ego_Target_Long, Ego_Target_Lat)   SafetyKPI1: [100.         6.6105337 100.         34.686226 ]
SafetyKPI2 = safety_margin.compute_safety_kpi2(Ego_Target_Lat, Ego_Target_Long)   SafetyKPI2: [-6.4193444 10.         10.         10.        ]

# Compute safety context based on Safety KPI values
SM_long_SafetyContext, SM_lat_SafetyContext = safety_margin.compute_safety_context(SafetyKPI1, SafetyKPI2)   SM_lat_SafetyContext: False   SM_long_SafetyContext: False
```

Fig. 8.8 (a): In-Depth Debugging: Safety KPIs and SM_safetyContext Calculations of Highway-Env

In the driving scenario depicted in Figure 8.8(b), the ego vehicle is approaching Target Vehicle 2 with a short longitudinal distance of 3.78 [m]. Meanwhile, Target Vehicle 1 has shifted to the middle lane, traveling closely to the ego vehicle with a lateral distance of 3.33 [m]. Clearly, in this driving scenario, the safety margin has decreased, and both KPIs fall below the safe threshold SM_Threshold2. Consequently, the safety context is activated, as indicated by SM_long_SafetyContext = True and

SM_lat_SafetyContext = True.



```
# Extract x,y-coordinates and vx, vy of Ego & target vehicles from Agent observations (Target values relative to Ego!!)
x_values = obs[0][:, 1]  x_values: [ 1.         -0.03275849  0.03783824  0.18955937  0.2578826 ]
y_values = obs[0][:, 2]  y_values: [ 0.6666667  -0.33333325  0.         -0.6666667   0.        ]
vx_values = obs[0][:, 3]  vx_values: [ 0.32073012 -0.05257794 -0.09806021 -0.07533967 -0.08286613]
vy_values = obs[0][:, 4]  vy_values: [ 0.0000000e+00 -3.8229903e-08  0.0000000e+00  0.0000000e+00,  0.0000000e+00]

# Longitudinal and Lateral Distance arrays (values in meter and relative to Ego)
Ego_Target_Long =x_values[1:5] * XRange  Ego_Target_Long: [-3.2758489  3.783824  18.955936  25.78826  ]
Ego_Target_Lat = y_values[1:5] * YRange  Ego_Target_Lat: [-3.3333325  0.        -6.666667   0.       ]

# Safety KPI values
SafetyKPI1 = safety_margin.compute_safety_kpi1(Ego_Target_Long, Ego_Target_Lat)  SafetyKPI1: [100.        3.783824 100.        25.78826 ]
SafetyKPI2 = safety_margin.compute_safety_kpi2(Ego_Target_Lat, Ego_Target_Long)  SafetyKPI2: [-3.3333325 10.        10.        10.       ]

# Compute safety context based on Safety KPI values
SM_long_SafetyContext, SM_lat_SafetyContext = safety_margin.compute_safety_context(SafetyKPI1, SafetyKPI2)  SM_long_SafetyContext: True    SM_lat_SafetyContext: True
```

Fig. 8.8 (b): In-Depth Debugging: Safety KPIs and SM_safetyContext Calculations of Highway-Env

8.5 RL- Safety Layer for Highway-env Environment

A summary of what has been covered in this chapter includes:

- Training state-of-the-art RL agents and evaluating them using TensorBoard logs.
- Implementing the safety margin scheme proposed in Chapter 7, Section 7.3, incorporating two safety KPIs along with RL and safety context information.
- Analysing crash incidents based on safety KPIs.

In Chapter 7, the safety layer with eight key features to enhance autonomous driving safety was proposed. In this chapter the following key features of the safety layer are implemented for Highway-env Environment and the performance and safety metrics of each of these features are evaluated in the subsequent sections:

- Safety Layer- Prevent Unsafe Actions (refer to 7.2.2 and its implementation in section 8.6)
- Safety Layer- Redundant RL Agents (refer to 7.2.6 and its implementation in section 8.7)

The synopsis of the code snippet (without details) for implementing the safety layer and the aforementioned features in the Highway-env environment is as follows:

```
class RLSafety_layer:
    def __init__(self, latDistThreshold, longDistThreshold):
        self.latDistThreshold = latDistThreshold
        self.longDistThreshold = longDistThreshold

    def prevent_unsafe_actions(self,action, safety_KPI1, safety_KPI2, ego_lane):
    def safety_arbitration(self, safety_KPI1, safety_KPI2, Agent1_Action,
Agent2_Action):
```

8.6 Safety Layer-Prevent Unsafe Actions

To validate and implement the innovative 'Safety Layer – Prevention of Unsafe Actions' feature proposed in Algorithm 7.2.3 in Chapter 7, this dissertation embarks on a dedicated experimental investigation. The primary decision-making entity in this exploration is the DQN Agent, previously trained in Section 8.3.5, operating in conjunction with the safety layer for preventing unsafe actions during highway driving.

In the presence of an active safety context, signifying a minimized safety margin, the algorithm watchfully monitors and intervenes in the agent's actions. Unsafe actions are detected, blocked, and overridden using a rule-based prevention strategy implemented by the safety layer. The algorithm remains neutral outside of the safety-context window and does not interfere with the DQN agent's decision-making process. These three steps can be summarized as:

- Verify Safety context window is active
- Detect whether the RL agent action is unsafe
- Refer to the rule-based strategy and find a safe replacement

The computation flow is as follows:

Environment state → Safety KPIs → Safety Context active?→ Agent action unsafe? → Override Action

The rule-based strategy for overriding unsafe actions by the safety layer is summarized in the table below for highway-env driving.[30]

Unsafe Action	**Prevention unsafe Action Replacement Action**	

lane left	'idle'
	'lane right'
	'slower'
		if Safety_KPI1 [nearest car]) >= longDistThresh
		if Safety_KPI2 [i] < 0 or Safety_KPI2 [i] > latDistThresh
		otherwise

idle	'lane left'
	'lane right'
	'slower'
		if Safety_KPI2 [i] > 0 or abs (Safety_KPI2 [i]) > latDistThresh
		if Safety_KPI2 [i] < 0 or Safety_KPI2 [i] > latDistThresh
		otherwise

lane right	'idle'
	'lane left'
	'slower'
		if Safety_KPI1 [nearest car]) >= longDistThresh
		if Safety_KPI2 [i] > 0 or abs (Safety_KPI2 [i]) > latDistThresh
		otherwise

faster	'idle' 'slower'	if Safety_KPI1 [nearest car]) >= longDistThresh otherwise

slower	'lane left'
	'lane right'
	'slower'
		if Safety_KPI2 [i] > 0 or abs (Safety_KPI2 [i]) > latDistThresh
		if Safety_KPI2 [i] < 0 or Safety_KPI2 [i] > latDistThresh
		otherwise

Table 8.3: Rule-based strategy for prevention of unsafe action in Highway-env driving

According to the rule-based strategy, the 'idle' action holds the highest priority as a replacement action, while 'slower' is assigned the lowest priority. In instances where no alternative action is deemed safe, and the vehicle is unable to change lanes, the safety layer for preventing unsafe agent actions intervenes by slowing down the vehicle to lower the severity of a potential accident.

[30] The rule-based strategy presented here for overriding unsafe actions with safe alternatives is intended for demonstration purposes. In real-world driving scenarios, a more realistic and sophisticated approach, considering a broader range of factors, should be implemented to ensure safety.

8.6.1 Algorithm Implementation and Experimental Setup

The trained DQN model is loaded and executed for 30 episodes (up to a maximum of 1200 time-steps). This process generates visualizations of the agent's reward, episode length, Safety KPIs, crash incidents, as well as the primary actions taken by the agent and actions overridden by the safety layer.

The code snippet to implement the algorithm is as below:

```
# Compute safety context based on Safety KPI values
SM_long_SafetyContext, SM_lat_SafetyContext =
safety_margin.compute_safety_context(SafetyKPI1, SafetyKPI2)
#----------
# ACTIONS
#----------
# DQN Agent observes the Environment and predicts Actions
DQNAction, _states = DQNAgent.predict(obs, deterministic=True)
DQNActions.append(DQNAction)
# Check if Safety Context is activated
if SM_long_SafetyContext or SM_lat_SafetyContext:
    # Call safety layer - prev_unsafe_action to prevent unsafe actions
    DQNActionUnsafe= action_unsafe(DQNAction, SafetyKPI1, SafetyKPI2,
latdistThresh, longdistThresh, Ego_lane_number)
    if DQNActionUnsafe:
        unsafe_action_detected_count += 1
        ReplacedAction = rl_safety_layer.prevent_unsafe_actions(DQNAction,
SafetyKPI1, SafetyKPI2, Ego_lane_number)
        if ReplacedAction != DQNAction [0]:
            SL_Action_Prevent_count += 1
    else:
        ReplacedAction = DQNAction[0]
else: # no intervention required!
    ReplacedAction = DQNAction[0]
ReplacedActions.append(ReplacedAction)
```

8.6.2 Analysing Experimentation Results: Crash Incidents and Action Overrides

The characteristics of the DQN agent were examined in Section 8.3.5. In this experiment, the DQN represents a comparable driving style, characterized by the failure to maintain a safe distance from the front vehicle and engaging in repeated lane-changing manoeuvres. This behaviour is evident in Safety_KPI1 and Safety_KPI2 metrics, surpassing the critical longitudinal distance level of 4 [m] and the critical lateral distance level (indicated by the dotted horizontal line) of 2.5 [m].

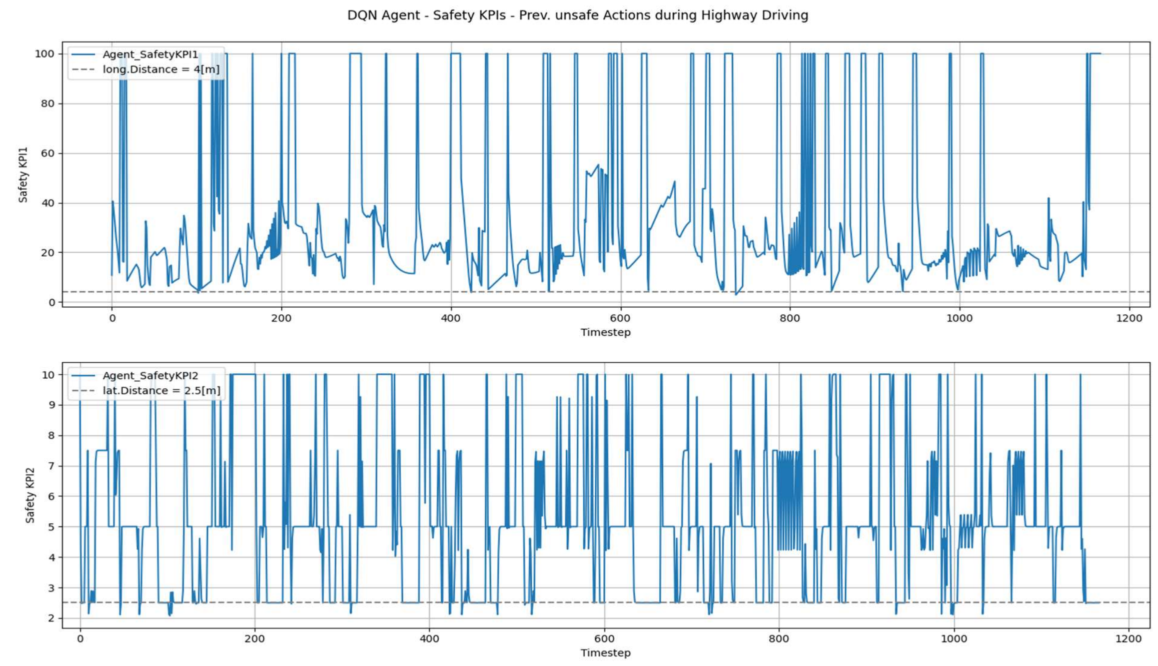


Fig. 8.9: Highway Driving with Prevention of Unsafe Actions - Safety KPIs of DQN Agent in

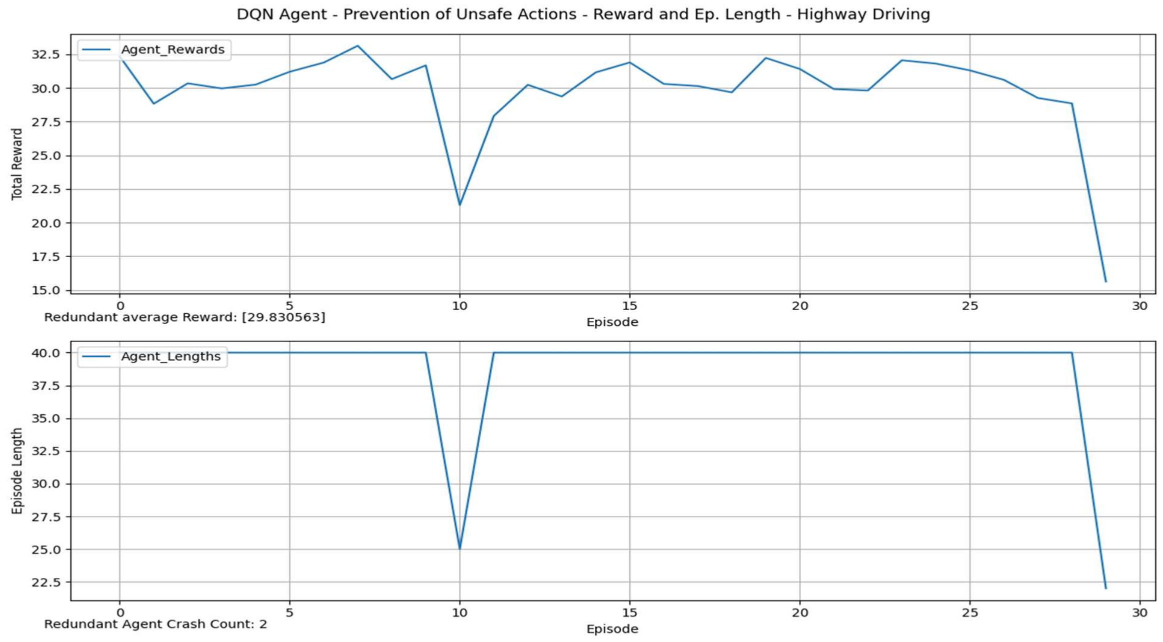


Fig. 8.10: Highway Driving with Prevention of Unsafe Actions - Total Reward and Episode Length of DQN Agent in

The agent achieves high rewards and episode length and experiences two crash incidents in episode 11 and 30. (refer to Figure 8.10). The episode numbers range from 0 to 29.

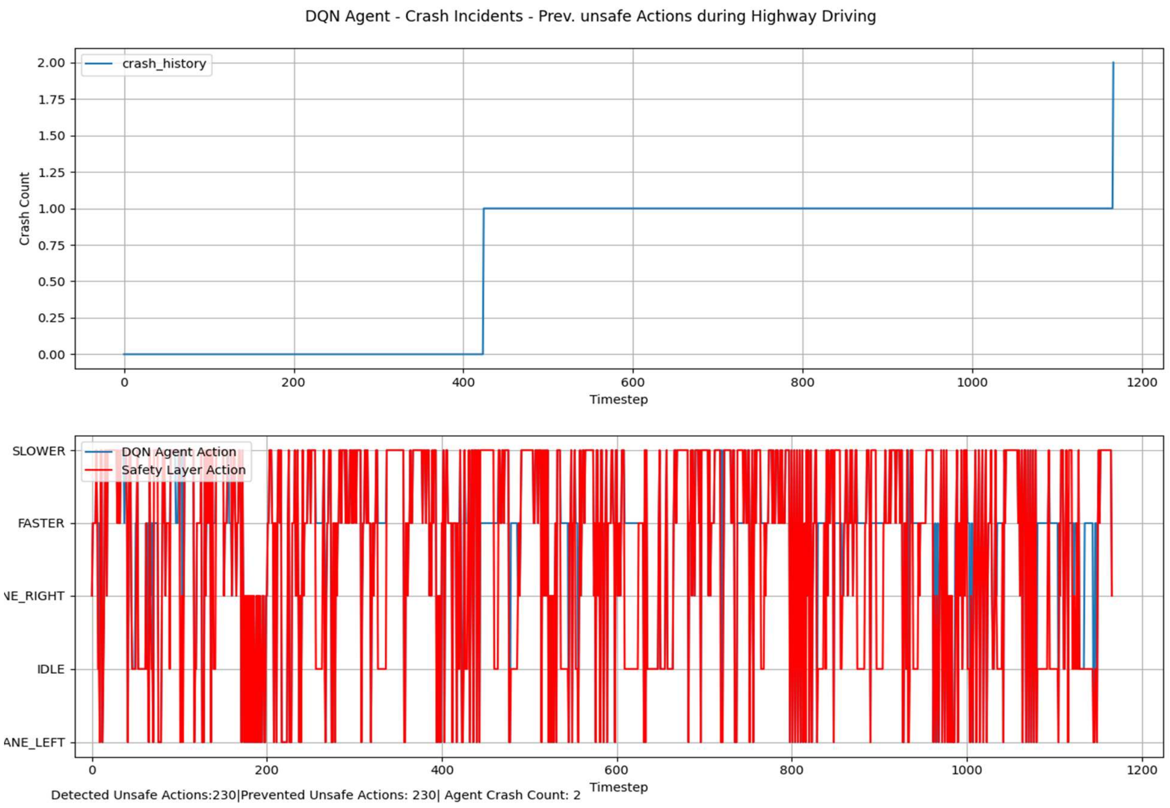


Fig. 8.11: Highway Driving with Prevention of Unsafe Actions - Crash Incidents and Action Override of DQN Agent in

In driving scenarios with the safety context active, specifically when the longitudinal distance is below 10 [m] or the lateral distance is below 6 [m], the safety layer effectively identifies and blocks 230 instances of unsafe actions performed by the agent. Following the rule-based guidelines outlined in Table 8.4, the safety layer replaces these risky actions with safer alternatives. This proactive intervention results in a noteworthy reduction in crash incidents, with only 2 crashes occurring over 30 episodes (see Figure 8.11 a). For comparison, the DQN agent, as observed in Section 8.3.5, encountered 24 crash incidents within 100 episodes, indicating a 24% accident rate.

Notable examples of action replacements include:
Transitioning from ‘Faster’ to ‘Idle’ at time-steps: [49-51], [1135-1143], [1080-1092]
Changing from ‘Faster’ to ‘Slower’ at time-steps: [71-75], [96-99], [1117-1120]
(see Figure 8.11b)

These instances illustrate the safety layer's ability to dynamically modify the agent's behaviour, mitigating potential risks and substantially enhancing the overall safety performance during highway driving simulations.

<u>Analysis of Crash Incidents Despite Safety-Layer Prevention of Unsafe Actions</u>

The software implementation of the algorithm records a log file each time a crash incident occurs, storing all necessary information for analysing the incident.

To highlight some of the limitations of the algorithm, the first crash incident in the experiment is thoroughly analysed here:

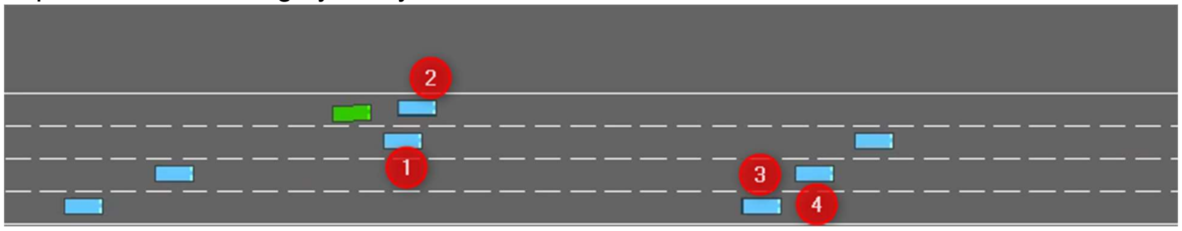


Fig. 8.12: Highway Driving with Prevention of Unsafe Actions - Ego vs Target Vehicles at Crash Incident 1

Figure 8.12 presents a snapshot one frame before crash incident 1 occurs. The crash information extracted from the log file is as follows:

```
Crash 1 at Time-Step: 424:
Coordinates:
X: [2.99891       3.869151     24.930276  28.20149 ],
Y: [ 2.1584259    -0.3415741   7.158426   4.658426 ]:
EGO Lane Nr.: 0,  Agent action [0], SL detected unsafe: False
Episode Information:
{'speed': 20.330862055536215, 'crashed': True, 'action': 0, … }
```

With only 3.869 m distance to the target vehicle 2 and a travel speed of 20.33 m/s, the ego vehicle has little chance to avoid an accident. The DQN agent detects target vehicle 1 at its right side and commands a lane change to the left side, which is considered an unauthorized action and replaced with ‘Idle’ by the environment.

The safety layer, with limited sophistication and a less comprehensive implementation, fails to identify the inadequacy of the agent's action, and consequently, it does not override the action, thus unable to prevent the accident.

One can infer that the 'Safety Layer - Prevention of Unsafe Actions,' characterized by its minimal yet effective intervention, exhibits the capability to prevent numerous crash incidents or mitigate the severity of accidents in various scenarios.

While the safety layer provides valuable enhancements, it is essential to underscore that it serves as a complementary measure and not a replacement for a more robust and secure machine learning-based autonomous driving system.

The algorithm's sensitivity to the rule-based strategy and its hyperparameters. A more comprehensive rule-set and fine-tuning of relevant parameters, such as safe longitudinal or lateral distances, along with the threshold parameter determining the safety context, can significantly enhance the performance and usability of the algorithm.

8.7 Safety Layer-Redundant Agents

To validate and implement the innovative 'Safety Layer - Redundant RL Agents' feature proposed in Algorithm 7.2.3 in Chapter 7, this dissertation proceeds with a dedicated experimental investigation.

Building upon previous experiments in section 8.3.5 with PPO and DQN agents, both are now employed as redundant agents within the 'Highway-env' environment.

The DQN agent demonstrated fast driving and frequent lane-changing behaviour, while the PPO agent, although relatively slower than the DQN agent, exhibited very high safety scoring.

The rule-based strategy for safety arbitration between two RL agents is summarized in the table below.[31]

Nr.	Action1-Action2	Safety-Arbitration		Explanation
2	faster - idle	'idle'		
1	faster - slower	'slower'		
3	slower - idle	'slower'		
6	lane left - slower	'slower'		
9	lane right - slower	'slower'		
4	lane left - lane right	'lane left' 'lane right'	*if **Safety_KPI2 [nearest car]** >= 0* *otherwise*	'lane left' is safer If the nearest car is at the right side. 'lane right' is safer If the nearest car is at the left side.

5	lane left - faster	'faster'
		'lane left'
			*if **Safety_KPI1 [nearest car]) >= longDistThresh***
			*if **Safety_KPI2 [i] > 0** or*
			abs(Safety_KPI2 [i]) > latDistThresh
				'faster' is safer *if the front vehicles are far than a threshold1.*
				'lane left' is safer if no target vehicle is at left side or lateral distance is larger than threshold2.

7	lane left - idle	'Idle'
		'lane left'
			*if **Safety_KPI1 [nearest car]) >= longDistThresh***
			*if **Safety_KPI2 [i] > 0** or*
			abs(Safety_KPI2 [i]) > latDistThresh
				'Idle' is safer *if the front vehicles are far than a threshold1*
				'lane left' is safer if no target vehicle is at left side or lateral distance is larger than threshold2.

8	lane right - faster	'faster'
		'lane right'
			*if **Safety_KPI1 [nearest car]) >= longDistThresh***
			*if **Safety_KPI2 [i] < 0** or*
			Safety_KPI2 [i] > latDistThresh
				'faster' is safer *if the front vehicles are far than a threshold1.*
				'lane right' is safer if no target vehicle is at right side or lateral distance is larger than threshold2.

10	lane right - idle	'idle'
		'lane right'
			*if **Safety_KPI1 [nearest car]) >= longDistThresh***
			*if **Safety_KPI2 [i] < 0** or*
			Safety_KPI2 [i] > latDistThresh
				'idle' is safer *if the front vehicles are far than a threshold.*
				'lane right' is safer if no target vehicle is at right side or lateral distance is larger than threshold2.

Table 8.4. Rule-Based Safety Arbitration Between Two Redundant Agents in the Highway-env Environment

[31] The rule-based strategy for safety arbitration presented here is intended for demonstration purposes. In real-world driving scenarios, a more realistic and sophisticated approach, considering a broader range of factors, should be implemented for ensuring safety.

8.7.1 Algorithm Implementation and Experimental Setup

To harness the strengths of each RL agent and enhance the overall safety of autonomous driving, a specific agent constellation is devised. The configuration involves:

The DQN agent is designated as the dominant agent within the constellation, responsible for primary decision-making in autonomous driving.

The PPO agent acts as a redundant agent, operating independently but concurrently with DQN, executing the same tasks, sharing the same state space, and having identical action spaces.

The trained DQN and PPO models in a redundant combination, as proposed above, are loaded and run for 10 episodes (maximum 400 time-steps), generating visualizations of Safety_KPI1 and Safety_KPI2, as well as action overrides by the safety layer-redundant agent.

The code snippet for implementing the Redundant Agent is as follows:

```
# Both DQN and PPO Agents observe the Environment and predict Actions
DQNAction, _states = DQNAgent.predict(obs, deterministic=True)
PPOAction, _states = PPOAgent.predict(obs, deterministic=True)
DQNActions.append(DQNAction)
PPOActions.append(PPOAction)

# Check if DQNAction and PPOAction are different
if DQNAction != PPOAction:
    # Call safety arbitration to compute Red. Agent action
    RedAgentAction = rl_safety_layer.safety_arbitration(SafetyKPI1, SafetyKPI2,
DQNAction, PPOAction)
    if RedAgentAction == DQNAction:
        # PPO Action overriden
        SL_DQNArbit_count += 1
    elif RedAgentAction == PPOAction:
        #DQN Action overriden
        SL_PPOArbit_count += 1
else: # no arbitration required!
    RedAgentAction = DQNAction[0]
#default DQN action if safety arbitration fails
if RedAgentAction != DQNAction[0] and RedAgentAction != PPOAction[0]:
    RedAgentAction  = DQNAction[0]
    SL_failedArbit_count += 1
```

8.7.2 Analysing Experimentation Results: Redundant Agents and Safety Arbitration

The experiment aimed to visualize Safety_KPI1 and Safety_KPI2 and record action overrides overruled by the safety layer-redundant agent, guided by the rule-based arbitration strategy outlined in Table 8.4. The results of this experiment are presented in the Figures 8.13 (a), (b), (c) and 8.14 (a), (b), (c).

The redundant agent exhibited remarkable effectiveness in preventing any crash incidents and enhancing the safety KPIs. It intervened by overriding 21 actions of the PPO agent in favour of the DQN agent and 366 actions of the DQN agent in favour of the PPO agent. These interventions were meticulously aligned with the safety layer's rule-based arbitration strategy, as outlined in Table 8.4.

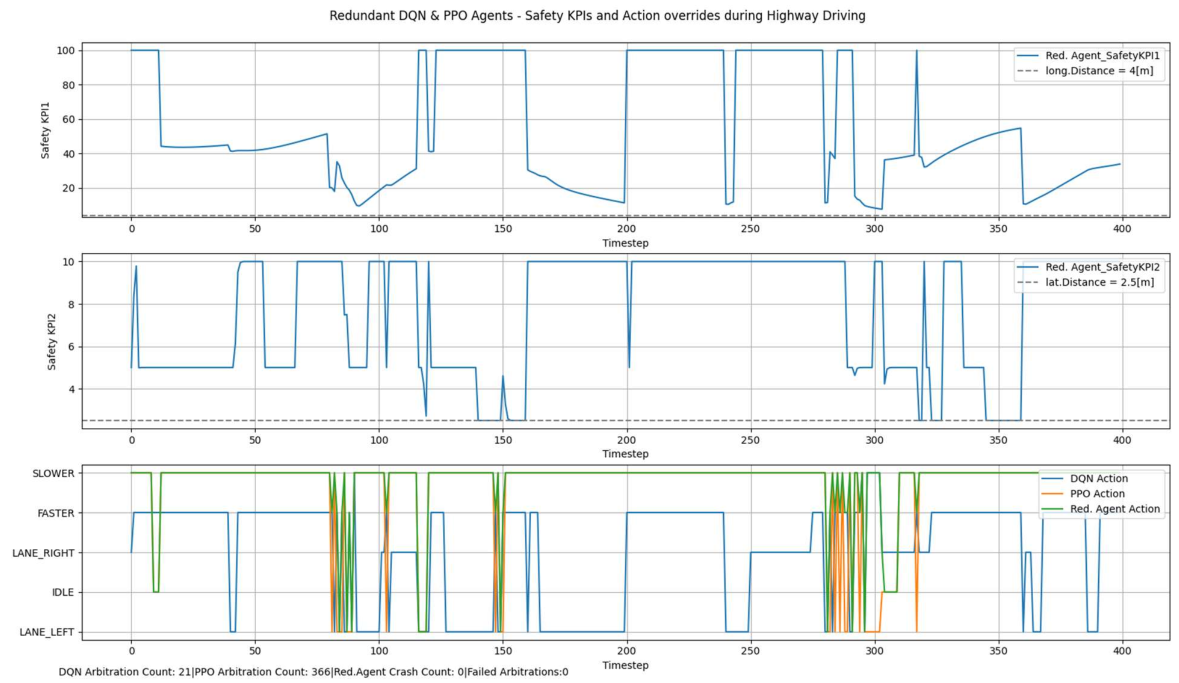


Fig. 8.13: Highway Driving with Redundant Agents - Safety KPIs (a), (b), Action Overrides (c)

At intervals for example [0-79], [151-277], and [318-400] the redundant agent (highlighted in green in Figure 8.13(c)) replaces DQN 'Faster' or 'Lane left' or 'Lane right' actions with 'Slower' actions from the PPO agent. This strategic adjustment ensures that the vehicle maintains a safe distance from the one in front. Consequently, Safety KPI1 remains above the critical longitudinal distance of 4 [m], as indicated by the dotted line in Figure 8.13(a).

Furthermore, at time-steps for example [80], [83-86], [103], [148-150], and [293-295], there were instances where the PPO actions 'Lane left' were identified as potentially unsafe by the safety layer-redundant agent. As a result, the redundant agent prioritized DQN actions, steering the vehicle to remain in the same lane and even accelerate. The benefits of overriding the 'Lane left' action can be clearly seen as an improvement in lateral distance in Figure 8.13(b).

Over the course of an extended duration spanning 100 episodes, providing a basis for comparison similar to the experimental results presented in Section 8.3.5, Figure 8.4. The results of the long run are depicted in Figure 8.14 (a), (b), and Figure 8.15 (a), (b), and (c).

The safety-layer redundant agent achieves high average rewards and episode length, as depicted in Figures 8.14(a) and (b).

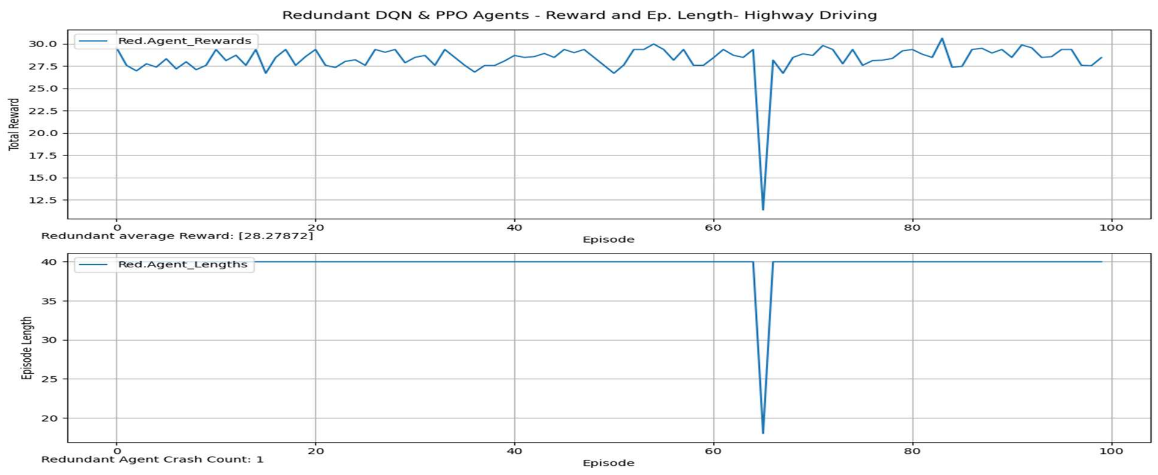


Fig. 8.14: Highway Driving with Redundant Agents - Average Reward (a) and Episode Length (b)

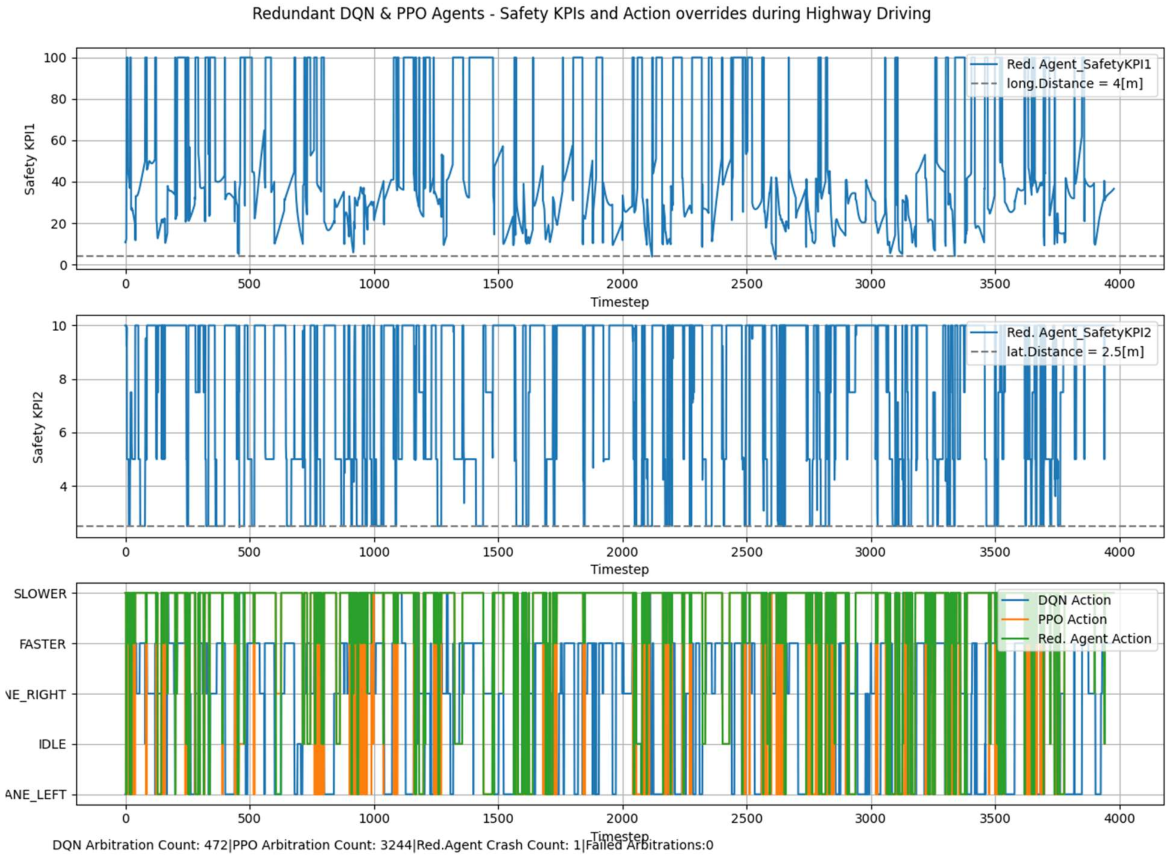


Fig. 8.15: Highway Driving Redundant Agents -100 Episode- Safety KPIs (a), (b), Action Overrides (c)

Thanks to its significant number of interventions (overriding 472 actions of PPO and 3244 actions of DQN), the redundant agent experienced a remarkable decrease in crash incidents from 24 crashes in 100 episodes, as observed in the DQN-alone agent in Section 8.3.5, to only 1 crash in 100 episodes in the new experimentation. These results demonstrate a substantial improvement over the DQN-alone agent.

In conclusion, the implementation of the safety-layer redundant agent, incorporating rule-based safety arbitration, exhibits promising outcomes. The redundant agent achieves high rewards and episodes and can effectively maintain a safe distance, hence mitigating crash incidents through numerous interventions.

An interesting aspect of the algorithm lies in its capacity to harmoniously combine two RL agents with distinct characteristics—one emphasizing performance and the other prioritizing safety—thereby complementing each other. Furthermore, there are opportunities to dynamically configure redundant agents more adaptively based on the specific demands of various applications.

Its implementation complexity is rated as medium. However, the success of the redundant agent is significantly tied to its rule-based arbitration strategy.

One minor point to highlight here is that, if the environment receives an unauthorized action from the agent, it substitutes it with the 'Idle' action, which may have potentially catastrophic consequences. Developing a rule-based arbitration strategy that takes such aspects into account can undoubtedly lead to safer results.

In scenarios where both agents output unsafe actions, the existing safety arbitration algorithm is incapable of preventing crash incidents.

8.8 Safety Layer-Safety Dependent Policy Optimization

To validate and implement the proposed 'Safety Layer - Safety Dependent Policy Optimization' feature outlined in Algorithm 7.2.4 from Chapter 7, this dissertation conducts dedicated experiments by integrating it into a PPO RL Agent.

The safety margin-dependent constraint is designed to dynamically adjust the clipping of the policy update, relying on an advantage function and a safety penalty function, as mathematically expressed by Formula 7.2.11: $A\ (s,\ a) = R\ (s,\ a) - \lambda.\ M(s)$

Where, λ is a hyperparameter controlling the strength of the safety penalty.

The safety penalty advantage function is defined based on Safety_KPI1, which represents the longitudinal distance between the autonomous vehicle (Ego) and the nearest target vehicle ahead.

$$\text{Penalty M}(s) = \begin{cases} Threshold\ -\ safetyKPI1 & ; if\ safetyKPI1 < Threshold \\ 0 & ; otherwise \end{cases} \qquad (8.8.1)$$

Here, the Threshold (by default set to 10[m]) serves as the boundary condition parameter, determining whether the penalty should be imposed.

8.8.1 Algorithm Implementation and Experimental Setup

To incorporate the 'Safety Layer - Safety Dependent Policy Optimization' feature into a Simulation in the Loop (SiL) framework, it is essential to extend the PPO class from the Stable Baselines 3 (SB3) library [2] and override its train function. ‘PPO.py’ from SB3 library function is MIT licensed. Hence, both in the code and in the References, this library is referenced.

The code snippet for inserting a safety penalty into the PPO train function:

```
# Access the current observation from the rollout buffer
current_observations = rollout_data.observations

#  safety penalty calculation
safety_penalty = self.calculate_safety_penalty(current_observations, 10)

# Clipped surrogate loss with safety penalty
policy_loss_1 = advantages * ratio - safety_penalty
policy_loss_2 = advantages * th.clamp(ratio, 1 - clip_range, 1 + clip_range) -
safety_penalty
policy_loss = -th.min(policy_loss_1, policy_loss_2).mean()
```

The code snippet for calculating the safety penalty function from the agent observations (state):

```
def calculate_safety_penalty(self, observations, threshold=10.0):
    x_values = observations[:, :, 1]
    Ego_Target_Long = x_values[:, 1:5] * XRange

    # Calculate Safety KPI1 , ignoring negative values
    safety_kpi1_values, _ = th.min(th.where(Ego_Target_Long < 0, th.inf,
Ego_Target_Long), dim=-1)
    # Calculate Safety Penalty based on Safety KPI1
    safety_penalty = th.where(safety_kpi1_values > threshold, th.tensor(0.0),
                              (threshold - safety_kpi1_values) * self.safety_coeff)

    return safety_penalty
```

Rationale behind the implementation:

The original Policy Loss (Clipped Surrogate Loss) is computed using the clipped surrogate objective function, which comprises two terms: *‘policy_loss_1’* and ‘*policy_loss_2’*, as shown in the code snippet above. ‘*policy_loss_1’* and ‘*policy_loss_2’* have been modified to incorporate the safety penalty function.

The safety penalty function operates within the range [0, 10] and is multiplied by the safety coefficient λ (default value: 0.1). This safety penalty function introduces a regularization effect on policy optimization. Higher penalty values penalize actions leading to lower safety, thereby encouraging a more conservative policy.

At its maximum value of 10 (SafetyKPI = 0), the safety penalty signifies a scenario where the longitudinal distance is zero, indicating the worst safety condition. Consequently, policy_loss_1 will be higher, promoting a more substantial shift in the policy towards safer actions.

When the safety penalty function is closer to 0 (SafetyKPI = 10 meter or higher), it indicates a better safety condition. This results in a smaller policy_loss_1, suggesting a less aggressive policy update in the direction of safety.

8.8.2 Results of the experimentation

The PPO RL agent is once again trained using the new policy optimization based on the safety penalty function. Both the previously trained (PPO-Baseline) and the newly trained PPO (PPO with Safety Penalty (SP)) models are loaded and executed for 30 episodes (with a maximum of 1200 time-steps). This process generates visualizations of Safety_KPI1 and Safety_KPI2, along with the reward and episode length of each agent.

The experiment's objective is to visualize Safety_KPI1 for both PPO models and compare which model maintains the highest safety distance to the nearest target vehicle ahead. The outcomes of this experiment are presented in Figures 8.16 (a) and (b). Both PTO baseline and PPO_with_SP exhibit comparable results in terms of total rewards and episode lengths (see Figure 8.16 (a) and (b)). The PPO agent with Safety Penalty demonstrates a slightly better average safety margin of 59.17 meters compared to the PPO baseline, which achieves an average of 51.87 meters over 30 episodes (as shown in Figure 8.16 (c). These results suggest that the inclusion of the safety layer and the safety-dependent policy optimization feature remarkably contributes to maintaining a safer distance from the nearest target vehicle. However, it's important to note that these results were not achieved under identical driving conditions for both models. For a more accurate evaluation, additional experiments and fine-tuning of the safety-dependent policy optimization function, are recommended.

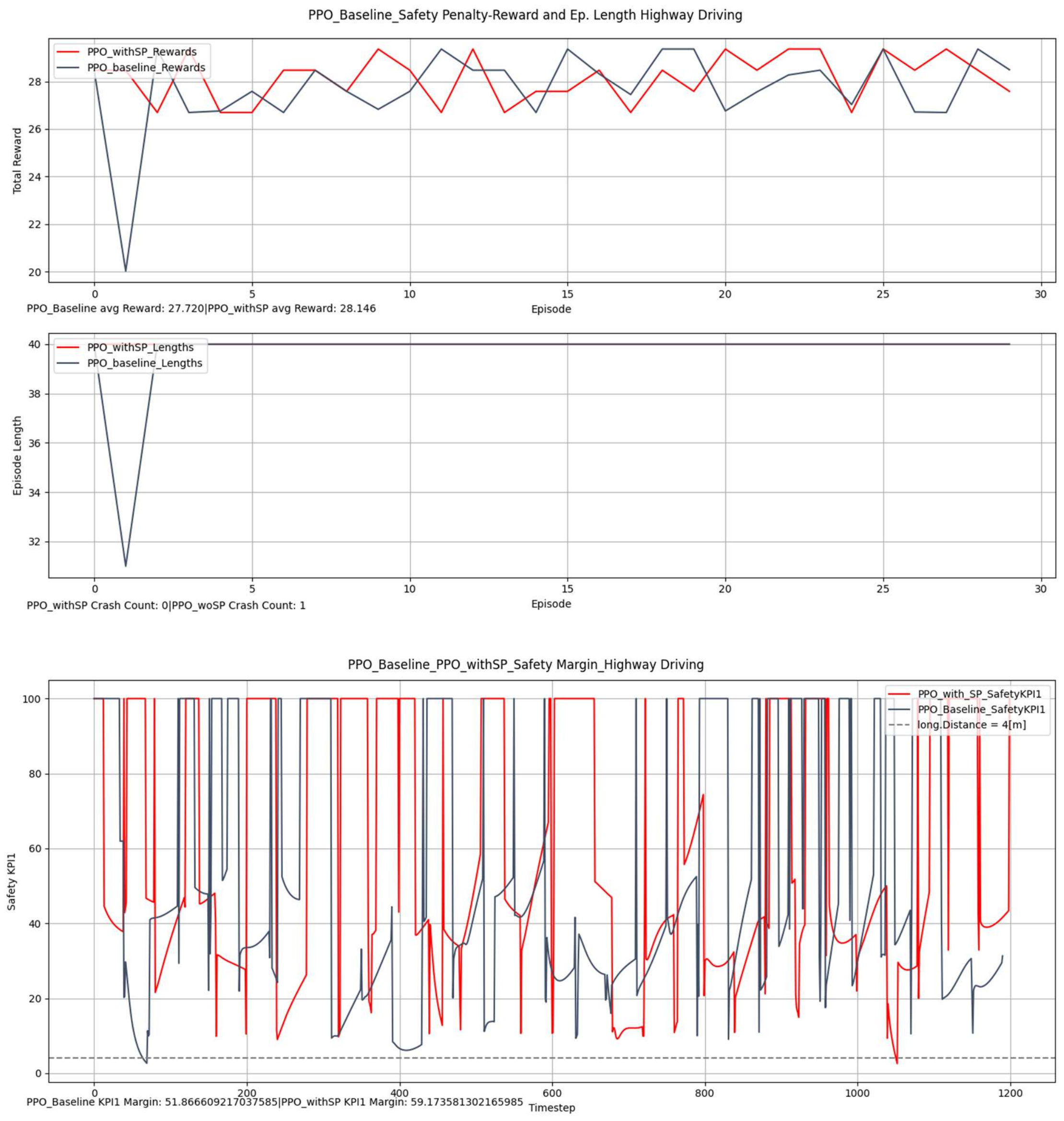


Fig. 8.16: Highway Driving PPO Safety Dependent Policy Update -30 Episode- Reward (a) and Ep. Length (b) Safety KPIs (c)

8.9 Conclusion

By providing essential algorithmic implementation and a well-structured experimental setup (Software-in-the-Loop (SiL), this chapter examined and reported on the effectiveness and usefulness of the proposed safety layer and safety margin scheme in enhancing the safety of autonomous driving.

The introduction of safety versus RL contexts, along with two safety Key Performance Indicators (KPIs) based on the longitudinal and lateral distances between the Ego and target vehicles, provided a proof-of-concept for the merits of RL agent decision-making in various driving scenarios and prevention of crash incidents.

By changing the safety KPIs in a simulation environment, the correlation between crash incidents and the degradation of safety metrics, particularly in the moments leading up to accidents was demonstrated. This quantified correlation sheds light on RL agent behaviour and establishes the analytical capabilities for better understanding of the potential occurrence and avoidance of accidents in autonomous driving.

To demonstrate the merit of the safety layer features, two features from its RL context (SL-Redundant Agents and SL-Safety Dependent Policy Optimization) and one feature from its Safety Context (SL-Prevent Unsafe Actions) were adapted and implemented in the SiL simulation framework.

The results show that Safety Layer features are critical factor in enhancing safety, based on safety constraints and driving behaviour. This is evident in the significant number of its action overrides, notable improvements in safety KPIs, and a remarkable reduction in crash incidents.

In addition, this chapter objectively assessed and reported on the limitations and potential areas for improvement of the Safety Layer features.

It could be concluded that a robust and adaptable environment generated by the proposed solutions can significantly improve the quality of autonomous control (controllability) of the vehicle within the diverse safety-enhancing functionalities. It proves the original hypothesis of this research work.

The results also lay the foundations and proof for the necessity of implementing features like SL-Reward Shaping and SL-Human Imitations for further improvements of safety, explained in Chapter 7.

9 Chapter 9: Conclusion, Contributions, and Future Work

<u>Summary of Research Objectives</u>

In conclusion, this dissertation has made steps in advancing the understanding the challenges of safety in road based autonomous driving. It provides and attempts to provide a tool/solution by improving and customizing reinforcement learning (RL) as a suitable alternative to the state-of-the-art.

In this thesis, the author set a number of objectives for finding a way to increase road worthiness of autonomous vehicles, which are gaining momentum in the society and leading automotive companies. These objectives were:

- Objective 1: Acquiring Subject Area Knowledge
- Objective 2: Gaps in state-of-the-art solutions (comparative analysis)
- Objective 3: Creating Experimental Framework
- Objective 4: Conducting Tests and Evaluations

Objective 1 was achieved through an in-depth literature review, particularly in Chapters 2, 3, and 4, where essential knowledge was acquired on relevant machine learning methods, neural networks, and reinforcement learning. Objective 2, outlined in Chapter 6, focused on identifying risks associated with state-of-the-art deep neural network (DNN) classifiers in Advanced Driver Assistance Systems (ADAS). The work included a comparative analysis, leading to the proposal of a safety framework designed to mitigate these risks and ensure compliance with industry standards.

Chapter 5 and Chapter 8 achieved Objective 3 by designing experimental frameworks through the customization of existing Gym environments and the establishment of necessary metrics for performance evaluation. These frameworks were designed and the necessary simulation platform was implemented to visualize and asses the influence design variations on RL DQN performance and safe autonomous highway driving, respectively.

Finally, Objective 4, detailed in Chapter 8, involved conducting tests and evaluations. Thanks to the proposed safety metrics, a quantitative analysis of RL agent decision-making and safety Key Performance Indicators (KPIs) was achieved, establishing a valuable correlation with crash incidents.

Overall, an attempt was made to fulfill these objectives in this dissertation. It demonstrated a holistic and structured approach to advancing the body of knowledge in the field of autonomous driving safety through reinforcement learning methodologies.

<u>Achievements and Contributions</u>

This dissertation has four interrelated but distinct features, resulting into a number of contributions.

First, Introduction of various major design variations of DQN reinforcement learning, including factors such as the size of the replay buffer, various architectures of hidden layers, backpropagation optimizers, convergence criteria, target network update, etc.

It systematically assesses their impact on agent performance and training stability (presented in Chapter 5).

Second, identification of potential risks, encompassing both immediate and design variation risks, in the development of deep neural network (DNN) classifiers. It introduces a qualitative safety framework designed to mitigate these risks throughout the design, training, implementation, and validation phases. The framework ensures compliance with automotive safety standards, including ISO 26262, ISO 21448, and ISO PAS 8800 (presented in Chapter 6).

Third, introduction of a safety override layer with eight key features, empowering reinforcement learning agents to enhance safety margins across diverse driving scenarios (presented in Chapter 7). The contributions include:

- Introduction of a new exploration metric, quantifying the exploration profile of RL agents in continuous state spaces.
- Introduction and Implementation of the Exploration Maximization Algorithm to maximize RL agent exploration.
- Integration of human expert guidance in RL training for autonomous driving.
- Protecting prior knowledge through policy update constraints.
- Preventing unsafe actions through the detection and override of unsafe RL decision-making.
- Integration of safety penalty Key Performance Indicators (KPIs) in reward shaping to enhance the safety awareness of RL agents.
- Safety-dependent policy optimization for enabling a quicker return to a safer state.
- Safety arbitration in the redundant multi-agent RL setting in AD.
- Introduction of a policy update mechanism based on human imitation
- Introduction of a fail-safe strategy for RL agents in the event of system degradation in autonomous driving.
- Introduction of a safety margin scheme and safety KPIs to enhance autonomous driving safety.

Fourth, quantification of safety metrics within the context of the AI-driven solution proposed by evaluating RL agents in different highway driving scenarios, establishing a correlation between crash incidents and safety metrics in autonomous driving, showcasing the effectiveness of the proposed safety layer in detecting and preventing unsafe actions, and demonstrating its success in conducting safety arbitration between two distinct RL agents (presented in Chapter 8).

<u>Empirical Results and Key Findings</u>

In essence, the presented results underscore the pivotal role of safety considerations in RL-based autonomous driving systems. The proposed safety metrics and safety layer features not only deepen our understanding of RL agent behaviour but also offer practical solutions to positively influence decision-making and proactively prevent accidents in dynamic driving situations.

Building upon these findings, the empirical results highlight notable enhancements achieved by the implemented safety measures:

- The Exploration Maximization Algorithm exhibited an increase in exploration metrics by 0.06% and 0.08%, as illustrated in in Section 7.2.1, Figure 7.5 (a).
- The Safety Layer-Prevent Unsafe Actions module resulted in a notable reduction of the accident rate from 24% to 6.6%, as illustrated in Section 8.6, Figure 8.11.
- The Safety Layer-Redundant Agents solution demonstrated remarkable success, reducing the accident rate from 24% to 1%, as illustrated in Section 8.7, Figure 8.15.

Integration with Real-world Applications

The findings of this dissertation can be further utilized in other safety-critical domains, such as robotics and the aerospace industry. The Safety Layer, either in its entirety or through selective features, demonstrates considerable potential for integration into real-world products to enhance safety in RL-based applications. Continued collaboration with the broader autonomous driving research community is essential to share insights and collectively advance the field.

Future work

The simulation framework developed throughout this dissertation provides a solid foundation for the continued development and accurate evaluation of these safety measures, aiming to further advance the safety of autonomous driving systems.

The following future work can profit from the existing framework and findings of this dissertation:

- Implementation and evaluation of additional safety layer features, such as SL-Reward Shaping and SL-Human Imitation.
- Integration of various safety-enhancing functionalities into RL Agents for comparative analysis with the safety layer features.
- Further exploration to refine and optimize Safety Layer algorithms for diverse driving scenarios beyond highways, including stop-and-go driving, city driving, parking lots, etc.
- Extending the simulation environment to include a broader range of genuine driving scenarios and the capability to import realistic field data.
- Extending training cycles leveraging powerful GPUs to boost the learning quality of RL agents

Closing Remarks

To summarize, the continuous advancements in sensor and processing unit technologies, along with the progress in control algorithms based on deep neural networks and reinforcement learning techniques hold promise to enhance safety in autonomous driving. Nevertheless, ensuring safety in this domain remains a persistent challenge for both researchers and practitioners.

References

Chapter 2

[1] Kantardzic, M. (2019). Data Mining: Concepts, Models, Methods, and Algorithms (3rd ed.). Wiley

[2] Brunton, S. L., & Kutz, J. N. (2021). Data Driven Science & Engineering_2ndEdition. Cambridge University Press.

[3] Gorunescu, F. (2011). Data Mining: Concepts, Models and Techniques. Springer Berlin, Heidelberg.

[4] Russell, R. (2018). Machine Learning: Step-By-Step Guide to Implement Machine Learning Algorithms with Python. CreateSpace Independent Publishing Platform. ISBN: 1719528403

[5] https://en.wikipedia.org/wiki/Loss_function

[6] https://en.wikipedia.org/wiki/Mathematical_optimization

[7] Haykin, S. (2009). Neural networks and learning machines (3rd ed.). Pearson Education

[8] Beysolow II, T. (2017). Introduction to Deep Learning Using R: A Step-by-Step Guide to Learning and Implementing Deep Learning Models Using R. DOI: 10.1007/978-1-4842-2734-3_4

[9] Breiman, L. Random Forests. Machine Learning 45, 5–32 (2001). https://doi.org/10.1023/A:1010933404324

[10] B Jagadeesh & D. V. Vidhya Sree, 2022. "Detection and Recognition of Traffic Sign Boards using Random Forest Classifier," Review of Computer Engineering Research, Conscientia Beam, vol. 9(3), pages 135-149. https://ideas.repec.org/a/pkp/rocere/v9y2022i3p135-149id3109.html

[11] Dr. K. Velmurugan , B. Mathumitha, B. Merylen Jenow, R. Thamizh Oviyam, 2019, “Automated Vehicle: Autonomous Driving using SVM Algorithm in Supervised Learning”, International Journal Of Engineering Research & Technology (IJERT) RTICCT – 2019 (Volume 7 – Issue 01), DOI : 10.17577/IJERTCONV7IS01006

Chapter 3

[1] Buduma, N. and Locascio, N., 2017. Fundamentals of deep learning: designing next-generation machine intelligence algorithms. Sebastopol, CA: O’Reilly Media.

[2] Zeng Y, Xu X, Fang Y, Zhao K (2015) Traffic sign recognition using deep convolutional networks and extreme learning machine. In: Intelligence science and big data engineering. Image and video data engineering (IScIDE). Springer, Berlin, pp 272–280

[3] Brunton, S. L., & Kutz, J. N. (2021). *Data Driven Science & Engineering_2ndEdition*. Cambridge University Press.

[4] Beysolow II, T. (2017). Introduction to Deep Learning Using R: A Step-by-Step Guide to Learning and Implementing Deep Learning Models Using R. DOI: 10.1007/978-1-4842-2734-3_4

[5] Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention Is All You Need. In Proceedings of the Advances in Neural Information Processing Systems, 5998-6008.

[6] Yao, S., Guan, R., Huang, X., Li, Z., Sha, X., Yue, Y., Lim, E. G., Seo, H., Man, K. L., Zhu, X., & Yue, Y. (2023). Radar-Camera Fusion for Object Detection and Semantic Segmentation in Autonomous Driving: A Comprehensive Review. IEEE Transactions on Intelligent Vehicles. DOI: 10.1007/978-1-4842-2734-3_4.

[7] Ioffe, S.; Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Proceedings of the 32nd International Conference on Machine Learning (ICML-15), 2015.

[8] Nair, Y.; Hinton, G. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th International Conference on Machine Learning, 2010, pp. 807–814.

[9] Kantardzic, M. (2011). Data Mining: Concepts, Models, Methods, and Algorithms. John Wiley & Sons.

[10] LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-Based Learning Applied to Document Recognition. In Proceedings of the IEEE, 1998, p. 2278–2324.

[11] Habibi Aghdam, H., & Jahani Heravi, E. (2017). Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification. Springer International Publishing

[12] Bishop, C.M. F.R.Eng. 2006, Pattern Recognition and Machine Learning, Springer, ISBN-10: 0-387-31073-8.

[13] Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference for Learning Representations (ICLR). DOI: 10.48550/arXiv.1412.6980

[14] Heaton, Jeff. (2015). Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks. ISBN 13: 9781505714340.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. (2015). "ImageNet Classification with Deep Convolutional Neural Networks" https://pdfs.semanticscholar.org/09b8/120cbc52e7df46122e8e608146289fddbdfa.pdf

Chapter 4 and5

[1] Sutton, R. S., & Barto, A. G. Reinforcement Learning: An Introduction. 2nd ed., MIT Press, 2018.

[2] Lapan, M. (2020). Deep Reinforcement Learning Hands-On: Second Edition. Packt Publishing Ltd

[3] Liu, Y. (2019). PyTorch 1.x Reinforcement Learning Cookbook. Packt Publishing Ltd

[4] CS Cheatsheet. ([2023]). Markov Decision Processes (MDPs). https://cs-cheatsheet.readthedocs.io/en/latest/subjects/ai/mdp.html#rl-course-mdp-deepmind

[5] Rahul Sarkar, and Emma Brunskill. “CS234 Notes - Lecture 2 Making Good Decisions Given a Model of the World.” Stanford University, 20 March 2018 https://web.stanford.edu/~rsarkar/materials/lecture2-CS234.pdf

Rahul Sarkar, and Emma Brunskill. “CS234 Notes - Lecture 3 Model-Free Policy Evaluation: Policy Evaluation Without Knowing How the World Works.” Stanford University, Winter 2019. https://web.stanford.edu/class/cs234/CS234Win2019/slides/lnotes3.pdf

[6] OpenAI. "An Introduction to Reinforcement Learning." Spinning Up in Deep Reinforcement Learning. OpenAI, December 17, 2018. https://spinningup.openai.com/en/latest/

[7] Wikipedia. ”Dynamic Programming” https://en.wikipedia.org/wiki/Dynamic_programming

[8] K. Kim, "Enhancing Reinforcement Learning Performance in Delayed Reward System Using DQN and Heuristics," in *IEEE Access*, vol. 10, pp. 50641-50650, 2022, doi: 10.1109/ACCESS.2022.3174361.

[9] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.

[10] Paszke, A., & Towers, M. (n.d.). Reinforcement Learning (DQN) Tutorial. PyTorch. Retrieved July 31, 2023, from https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[11] Roy, B. (2019, January 16). optim.Adam vs optim.SGD. Let’s dive in. Medium. https://medium.com/@Biboswan98/optim-adam-vs-optim-sgd-lets-dive-in-8dbf1890fbdc

[12] Wang, Z. T., & Ueda, M. (2021). Convergent and Efficient Deep Q Network Algorithm. arXiv preprint arXiv:2106.15419. Retrieved from https://arxiv.org/abs/2106.15419

[13] Wang, Z. T., & Ueda, M. (2021). Convergent and efficient deep Q network algorithm. arXiv preprint arXiv:2106.15419. https://arxiv.org/abs/2106.15419

[14] van Hasselt, H., Guez, A., & Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning. Proceedings of the AAAI Conference on Artificial Intelligence

[15] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized Experience Replay. arXiv preprint arXiv:1511.05952. https://arxiv.org/abs/1511.05952

[16] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining

Improvements in Deep Reinforcement Learning. Proceedings of the AAAI Conference on Artificial Intelligence.

[17] Fortunato, M.; Azar, M. G.; Piot, B.; Menick, J.; Osband,I.; Graves, A.; Mnih, V.; Munos, R.; Hassabis, D.; Pietquin, O.; Blundell, C.; and Legg, S. 2017. Noisy networks for exploration. arXiv:1706.10295.

[18] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2016). Dueling Network Architectures for Deep Reinforcement Learning. In International Conference on Machine Learning (pp. 1995-2003). arXiv:1511.06581

[19] Bellemare, M. G.; Dabney, W.; and Munos, R. 2017. A distributional perspective on reinforcement learning. In ICML. arXiv:1707.06887

[20] Williams, Ronald J. "Simple statistical gradient-following algorithms for connectionist reinforcement learning." Machine learning 8.3-4 (1992): 229-256.

[21] Sutton, R. S., McAllester, D. A., Singh, S. P., & Mansour, Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In Advances in Neural Information Processing Systems (NIPS), 12.

[22] Jan Peters (2010) “Policy gradient methods” Scholarpedia, 5(11):3698 http://www.scholarpedia.org/article/Policy_gradient_methods

[23] J. Peters and S. Schaal, “Reinforcement learning of motor skills with policy gradients,” Neural Networks, vol. 21, no. 4, pp. 682-697, 2008.

[24] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust region policy optimization,” in Proceedings of the 32nd International Conference on Machine Learning, pp. 1889-1897, 2015. https://arxiv.org/pdf/1502.05477.pdf This version of the paper was last revised on 20 Apr 2017 .

[25] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” arXiv preprint arXiv:1707.06347, 2017.

[26] Poupart, P. from University of Waterloo. "CS885 - Module 1: Introduction to Reinforcement Learning." 2020. Web. URL: https://cs.uwaterloo.ca/~ppoupart/teaching/cs885-spring20/slides/cs885-module1.pdf

[27] Abhishek Gupta, Joshua Achiam. from University of Berkeley. "CS 294: Deep Reinforcement Learning" Fall 2017. Web. URL: http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_13_advanced_pg.pdf

[28] Levine, S. (2022). CS 182/282A: Deep Neural Networks [Lecture notes]. University of California, Berkeley

[29] MathWorks. (2021). Actor-Critic Agents. Retrieved from https://www.mathworks.com/help/reinforcement-learning/ug/actor-critic-agents.html

[30] Pignatelli, E.; Ferret,J.; Geist,M.; Mesnard,T.; van Hasselt,H.; and Toni, L. 2023, A Survey of Temporal Credit Assignment in Deep Reinforcement Learning.

arXiv:2312.01072 https://arxiv.org/pdf/2312.01072.pdf

[31] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In International conference on machine learning (pp. 1928-1937). arXiv: 1602.01783 https://arxiv.org/pdf/1602.01783.pdf

[32] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. In International Conference on Learning Representations (ICLR). arXiv:1509.02971 https://arxiv.org/pdf/1509.02971.pdf

[33] Gym Environments https://www.gymlibrary.dev/index.html

[34] Pytorch Library https://pytorch.org/

[35] Gym Github https://github.com/openai/gym

[36] Pytorch Reinforcement Learning (DQN) Tutorial https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[37] Kaviani, S.; Sohn, I. Application of complex systems topologies in artificial neural networks optimization: An overview. Expert Systems with Applications 2022, 180. https://doi.org/10.1016/j.eswa.2021.115073

[38] Tian, Y.; Su, D.; Lauria, S.; Liu, X. Recent advances on loss functions in deep learning for computer vision. Neurocomputing 2022, 497, 129–158. https://doi.org/10.1016/j.neucom.2022.04.127

[39] RMSProp Cornell Jason Huang (SysEn 6800 Fall 2020) University Computational Optimization Open Textbook https://optimization.cbe.cornell.edu/index.php?title=RMSProp

[40] Kingma, D. P. and Ba, J. (2015) ‘Adam: A Method for Stochastic Optimization’, in Bengio, Y. and LeCun, Y. (eds.) ICLR (Poster). https://arxiv.org/pdf/1412.6980.pdf

[41] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. In Proceedings of the International Conference on Learning Representations (ICLR)

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2015. https://arxiv.org/pdf/1502.01852.pdf

[43] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In International Conference on Machine Learning (ICML).
https://arxiv.org/pdf/1502.03167.pdf

[44] S. Karsoliya, "Approximating Number of Hidden layer Neurons in Multiple hidden Layer BPNN Architecture", International Journal of Engineering Trends and Technology, vol. 3, no. 6, 2012.

[45] M. Uzair and N. Jamil, "Effects of Hidden Layers on the Efficiency of Neural networks," 2020 IEEE 23rd International Multitopic Conference (INMIC), Bahawalpur, Pakistan, 2020, pp. 1-6, doi: 10.1109/INMIC50486.2020.9318195.

Chapter 6

1. Shaout, A.; Colella, D.; Awad, S. Advanced Driver Assistance Systems - Past, present and future. In Proceedings of the Seventh International Computer Engineering Conference (ICENCO'2011), Cairo, Egypt, 2011, pp. 72–82.

https://doi.org/10.1109/ICENCO.2011.6153935

2. Belmonte, F.J.; Martín, S.; Sancristobal, E.; Ruipérez-Valiente, J.A.; Castro, M. Overview of Embedded Systems to Build Reliable and Safe ADAS and AD Systems. IEEE Intelligent Transportation Systems Magazine 2021, 13, 239–250. https://doi.org/10.1109/MITS.2019.2953543

3. Lee, C.W.; Nayeer, N.; Garcia, D.E.; Agrawal, A.; Liu, B. Identifying the Operational Design Domain for an Automated Driving System through Assessed Risk. In Proceedings of the IEEE Intelligent Vehicles Symposium (IV), Las Vegas, NV, USA, 2020, pp. 1317–1322. https://doi.org/10.1109/IV47402.2020.9304552

4. Society of Automotive Engineers. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. SAE International 2018.

5. Research Insights Automotive 2030.IBM. https://www.ibm.com/downloads/cas/NWDQPK5B

6. International Organization for Standardization (ISO). ISO 26262 Road vehicles — Functional safety 2018.

7. ISO - ISO/PAS 21448:2019 - Road vehicles — Safety of the intended functionality (SOTIF). https://www.iso.org/standard/70939.html

8. SOTIF - A New Challenge for Functional Testing | SpringerLink https://link.springer.com/article/10.1007/s38314-020-0257-4

9. Xu, S.; et al. A Review of SOTIF Research for Human-machine Driving Mode Switch of Intelligent Vehicles. In Proceedings of the 6th CAA International Conference on Vehicular Control and Intelligence (CVCI), Nanjing, China, 2022, pp. 1–6.

https://doi.org/10.1109/CVCI56766.2022.9964885

10. Putting Safety of Intended Functionality SOTIF into Practice. https://www.sae.org/publications/technical-papers/content/2021-01-0196

11. Borrego-Carazo, J.; Castells-Rufas, D.; Biempica, E.; Carrabina, J. Resource-Constrained Machine Learning for ADAS: A Systematic Review. IEEE Access 2020, 8, 40573–40598. https://doi.org/10.1109/ACCESS.2020.2976513

12. Koopman, P.; Wagner, M. Challenges in Autonomous Vehicle Testing and Validation. SAE Int. J. Trans. Safety 2016, 4. https://doi.org/10.4271/2016-01-0128

13. Henriksson, J.; Borg, M.; Englund, C. Automotive Safety and Machine Learning: Initial Results from a Study on How to Adapt the ISO 26262 Safety Standard. In Proceedings of the IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS), Gothenburg, Sweden, 2018, pp. 47–49

14. ISO/AWI PAS 8800 - Road Vehicles — Safety and artificial intelligence. https://www.iso.org/standard/83303.html

15. (ISO) ISO PAS 8800 Road Vehicles - Safety and Artificial Intelligence. https://unece.org/transport/documents/2021/09/informal-documents/iso-iso-pas-8800-road-vehicles-safety-and-artificial

16. Autonomes Fahren – Auf der sicheren Seite - DE / Safe Intelligence. https://safe-intelligence.fraunhofer.de/artikel/autonomes-fahren-auf-der-sicheren-seite
17. Xu, P.; et al. Towards the Quantification of Safety Risks in Deep Neural Networks. arXiv, 2020.

18. Zhang, R.; et al. DDE process: A requirements engineering approach for machine learning in automated driving. In Proceedings of the IEEE 29th International Requirements Engineering Conference (RE), Notre Dame, IN, USA, 2021, pp. 269–279 https://doi.org/10.1109/RE51729.2021.00031

19. Schwalbe, G.; Schels, M. A Survey on Methods for the Safety Assurance of Machine Learning Based Systems. In Proceedings of the 1st European Congress on Embedded Real Time Software and Systems (ERTS 22), Toulouse, France, 2022.

20. Santana, M.A.; Calinescu, R.; Paterson, C. Mitigating Risk in Neural Network Classifiers. In Proceedings of the 48th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA), 2022.

21. González-Saavedra, J.F.; Figueroa, M.; Céspedes, S.; Montejo-Sánchez, S. Survey of Cooperative Advanced Driver Assistance Systems: From a Holistic and Systemic Vision. Sensors 2022, 22, 3040.

22. Nair, Y.; Hinton, G. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th International Conference on Machine Learning, 2010, pp. 807–814.

23. Glorot, X.; Bordes, A.; Bengio, Y. Deep sparse rectifier reading below neural networks. In Proceedings of the the 14th International Conference on Artificial Intelligence and Statistics, 2011, pp. 315–323.

24. Hinton, G.E. A practical guide to training restricted boltzmann machines. Neural Networks: Tricks of the Trade 2012, pp. 599–619.

25. Dong, X.and Zhuang, B.; Mao, Y.; Liu, L. Radar Camera Fusion via Representation Learning in Autonomous Driving. In Proceedings of the IEEE Intelligent Vehicles Symposium (IV), 2021, pp. 1–8.

26. Chen, Z.; Li, Z.; Sun, Y. Radar-Camera Fusion for Object Detection and Semantic Segmentation in Autonomous Driving: A Comprehensive Review. arXiv preprint arXiv:2304.10410., 2021.

27. Xue, M.; Li, J.; Luo, Q. Toward Optimal Learning Rate Schedule in Scene Classification Network. IEEE Geoscience and Remote Sensing Letters 2022, 19, 1–5. https://doi.org/10.1109/LGRS.2020.3040359

28. Iiduka, H. Appropriate Learning Rates of Adaptive Learning Rate Optimization Algorithms for Training Deep Neural Networks. IEEE Transactions on Cybernetics 2022, 52, 13250–13261. https://doi.org/10.1109/TCYB.2021.3107415.

29. Kaviani, S.; Sohn, I. Application of complex systems topologies in artificial neural networks optimization: An overview. Expert Systems with Applications 2022, 180. https://doi.org/10.1016/j.eswa.2021.115073.

30. Li, L.; Doroslovaˇcki, M.; Loew, M.H. Approximating the Gradient of Cross-Entropy Loss Function. IEEE Access 2020, 8, 111626– 111635. https://doi.org/10.1109/ACCESS.2020.3001531

31. Tian, Y.; Su, D.; Lauria, S.; Liu, X. Recent advances on loss functions in deep learning for computer vision. Neurocomputing 2022, 497, 129–158. https://doi.org/10.1016/j.neucom.2022.04.127

32. Mokhov, S.B.; Paquet, J.; Debbabi, M. Assessing the Adherence of an Industrial Autonomous Driving Framework to ISO 26262 Software Guidelines. In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2019, pp. 1–10.

33. White paper. Safety First for Automated Driving (SaFAD). Mercedes-Benz, Aptiv, Audi, Baidu, BMW, Continental, Fiat Chrysler Automobiles, HERE, Infineon, Intel and Volkswagen, 2019.

Chapter 7

[1] Barros, A., de Carvalho, J. P., & Rocha, L. (2020). Reinforcement and imitation learning applied to autonomous aerial robot control. In Journal of Intelligent & Robotic Systems (JIRS), 97(3), pp. 1-27. https://sol.sbc.org.br/index.php/sbrlars_estendido/article/download/14956/14802

[2] C.Burns, P.Izmailov, J.Hendrik Kirchner, B.Baker, L.Gao, L.Aschenbrenner, Y.Chen, A.Ecoffet, M.Joglekar, J.Leike, I.Sutskever, J.Wu, "Weak to Strong Generalization: Eliciting Strong Capabilities with Weak Supervision", 2023,

arXiv:2312.09390 https://doi.org/10.48550/arXiv.2312.09390

[3] Saunders, W., Sastry, O., Levine, S., Ekenel, H. K., & Isola, T. (2017). Trial without error: Towards safe reinforcement learning via human intervention. In Proceedings of the 31st International Conference on Machine Learning (ICML 2017), pp. 3286-3294. https://arxiv.org/abs/1707.05173

[4] Dalal, N. M., Finn, C., & Levine, S. (2018). Safe exploration in continuous action spaces. In Proceedings of the 32nd International Conference on Machine Learning (ICML 2018), pp. 1706-1717. https://arxiv.org/abs/1801.07780: https://arxiv.org/abs/1801.07780

[5] Amodei, D., Olah, C., Steinhardt, J., Tenenbaum, J., & Schulman, J. (2016). Concrete problems in ai safety. arXiv preprint arXiv:1606.06565. https://arxiv.org/abs/1606.06565: https://arxiv.org/abs/1606.06565

[6] Eysenbach, B., Levine, S., & Clune, D. (2017). Leave no trace: Learning to reset for safe and autonomous reinforcement learning. In Proceedings of the 34th International Conference on Machine Learning (ICML 2017), pp. 1580-1589. https://arxiv.org/abs/1711.06782

[7] L. Zhang, L. Shen, L. Yang, S. Chen, X. Wang, B. Yuan, D. Tao, "Penalized Proximal Policy Optimization for Safe Reinforcement Learning", 2022, arXiv:2205.11814 https://arxiv.org/abs/2205.11814

[8] Sumanta Dey, Pallab Dasgupta, Soumyajit Dey, "Safe Reinforcement Learning through Phasic Safety Oriented Policy Optimization", 2023, CEUR Workshop Proceedings (CEUR-WS.org) https://ceur-ws.org/Vol-3381/22.pdf

[9] H.Hsu, Q.Huang, S.Ha "Improving Safety in Deep Reinforcement Learning using Unsupervised Action Planning" 2022 IEEE International Conference on Robotics and Automation (ICRA) https://ieeexplore.ieee.org/document/9812181

[10] B. Ravi Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep Reinforcement Learning for Autonomous Driving: A Survey," IEEE Transactions on Intelligent Transportation Systems, vol. 22, no. 9, pp. 5786-5805, 2021. https://arxiv.org/abs/2002.00444

[11] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, Dario Amodei (2023). Deep reinforcement learning from human preferences. arXiv preprint arXiv:2204.06085. https://arxiv.org/abs/1706.03741

[12] L. M. Schmidt, J. Brosig, A. Plinge1, B.M. Eskofier, C. Mutschler
An Introduction to Multi-Agent Reinforcement Learning and Review of its Application to Autonomous Mobility, arXiv:2203.07676v2 [cs.AI] 2 Aug 2022

[13] Zhou, Z., Liu, G., & Tang, Y. (2023). Multi-Agent Reinforcement Learning: Methods, Applications, Visionary Prospects, and Challenges. arXiv preprint arXiv:2305.10091

[14] Tong Wu, Pan Zhou, Kai Liu, Yali Yuan, Xiumin Wang, Huawei Huang, and Dapeng Oliver Wu. 2020. Multi-Agent Deep Reinforcement Learning for Urban

Traffic Light Control in Vehicular Networks. IEEE Transactions on Vehicular Technology (2020) https://doi.org/10.1109/TVT.2020.2997896

[15] Wu, X., Chen, M., & Zhu, F. (2021). Multi-agent reinforcement learning for cooperative lane changing of connected and autonomous vehicles in mixed traffic. In Transportation Research Part C: Emerging Technologies, 129, 103401 https://link.springer.com/article/10.1007/s43684-022-00023-5

[16] S. Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. "Safe, multi-agent, reinforcement learning for autonomous driving" (2016). CoRR. arXiv:1610.03295

[17] Peng, X., Bertschinger, M., van de Ven, G., & Abbeel, P. (2018). Deep mimic: Example guided deep reinforcement learning of physics-based character skills. In Proceedings of the 35th International Conference on Machine Learning (ICML 2018), pp. 4248-4257. https://arxiv.org/abs/1804.02717

[18] Seo, S., Ishibashi, T., & Kobayashi, N. (2022). Semi-supervised imitation learning of team policies. In Machine Learning Proceedings (PMLR), 162(1), pp. 1-18. https://arxiv.org/abs/2205.02959

[19] Ciosek, K., Choromanski, K., & Isenberg, T. (2022). Imitation learning by reinforcement learning. In Journal of Machine Learning Research (JMLR), 23(71), pp. 1-57. https://pi-starlab.github.io/JIRL/

Chapter 8

[1] Highway-Env: An Environment for Autonomous Driving Decision-Making @misc {highway-env, author = {Leurent, Edouard}, title = {An Environment for Autonomous Driving Decision-Making}, year = {2018}, publisher = {GitHub}, journal = {GitHub repository}, howpublished = {\url{https://github.com/eleurent/highway-env}},}

[2] stable-baselines3 Library: @article{stable-baselines3, author = {Antonin Raffin and Ashley Hill and Adam Gleave and Anssi Kanervisto and Maximilian Ernestus and Noah Dormann}, title = {Stable-Baselines3: Reliable Reinforcement Learning Implementations}, journal = {Journal of Machine Learning Research}, year = {2021}, volume = {22}, number = {268}, pages = {1-8}, url = {http://jmlr.org/papers/v22/20-1364.html}}