Technical University of Crete

School of Electrical and Computer Engineering

Deep Learning-Guided Monte Carlo Tree Search for Lane-Free Autonomous Driving

Diploma Thesis

Ioannis Peridis

Thesis Committee

Supervisor: Georgios Chalkiadakis, Professor (School of ECE)

Committee Member: Michail G. Lagoudakis, Professor (School of ECE)

Committee Member: Ioannis Papamichail, Professor (School of PEM)

Chania, June 2024

Πολυτεχνειο Κρητης

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Δενδρική Αναζήτηση Μόντε Κάρλο Καθοδηγούμενη από Βαθιά Μάθηση για Αυτόνομη Οδήγηση Χωρίς Λωρίδες Κυκλοφορίας

Διπλωματική Εργασία

Ιωάννης Περίδης

Επιτροπή Διπλωματικής

Επιβλέπων: Γεώργιος Χαλκιαδάκης, Καθηγητής (Σχολή ΗΜΜΥ)

Μέλος Επιτροπής: Μιχαήλ Γ. Λαγουδάκης, Καθηγητής (Σχολή ΗΜΜΥ)

Μέλος Επιτροπής: Ιωάννης Παπαμιχαήλ, Καθηγητής (Σχολή ΜΠΔ)

Χανιά, Ιούνιος 2024

Abstract

Vehicular traffic management has become increasingly complex due to the rise in autonomous driving technologies. Traditional lane-based traffic systems, while structured and familiar, often struggle with optimization and dynamic flow control, leading to congestion and inefficiencies. By contrast, lane-free traffic environments offer a promising alternative by allowing vehicles to maneuver laterally across the entire roadway without the constraints of lanes, which could significantly enhance road capacity and traffic fluidity.

This thesis explores the use of Monte Carlo Tree Search (MCTS) and supervised deep learning techniques to advance autonomous driving technologies. MCTS is well-suited for dynamic and unpredictable environments, such as autonomous driving, due to its robust decision-making capabilities in complex scenarios. Combined with Deep Neural Networks (DNNs), which excel in pattern recognition and predictive modeling from large datasets, these technologies could potentially revolutionize traffic management systems.

A representation of MCTS in a Markov Decision Process (MDP) framework specifically tailored for lane-free traffic scenarios is developed, enhancing traditional MCTS approaches to better handle the demands of this environment. This thesis builds upon the existing MCTS model developed in a 2023 diploma thesis by Pantelis Giankoulidis, focusing initially on refining how the algorithm processes state information. Our enhancements significantly improved the operational efficiency and effectiveness of the MCTS framework, enabling it to handle high-density traffic situations more adeptly.

Further advancements were achieved by integrating a neural network into the MCTS framework to guide the selection phase. This integration utilized the predictive capabilities of DNNs, allowing for more informed decision-making and faster exploration during the tree search process. Additionally, we investigated a standalone neural network approach, designed to function without the exploratory benefits of MCTS, to evaluate its comparative effectiveness in decision-making.

Our thorough experimental evaluation demonstrates that the enhanced MCTS framework, supported by neural network guidance, markedly improves upon the lane-free vehicles' behaviour. Key metrics such as safety, through reduced collision rates, and efficiency, by optimizing speed, highlighted the substantial benefits of this integrated approach. These results underscore the potential of combining MCTS with neural network technologies to aid the decision-making in autonomous vehicular driving environments.

Περίληψη

Η διαχείριση οδικής κυκλοφορίας έχει γίνει ιδιαίτερα περίπλοκη λόγω της αύξησης των τεχνολογιών αυτόνομης οδήγησης. Τα παραδοσιακά συστήματα κυκλοφορίας με λωρίδες, ενώ είναι δομημένα και οικεία, συχνά αντιμετωπίζουν προβλήματα με την βελτιστοποίηση και τον δυναμικό έλεγχο ροής, οδηγώντας σε συμφόρηση και αναποτελεσματικότητες. Αντίθετα, τα περιβάλλοντα κυκλοφορίας χωρίς λωρίδες προσφέρουν μια ελπιδοφόρα εναλλακτική λύση, επιτρέποντας στα οχήματα να εκτελούν ελιγμούς πλάγια σε ολόκληρο το οδόστρωμα, χωρίς τους περιορισμούς των λωρίδων, πράγμα που θα μπορούσε να ενισχύσει σημαντικά την χωρητικότητα των δρόμων και την ροή της κυκλοφορίας.

Η διπλωματική εργασία αυτή διερευνά τη χρήση της Δενδρικής Αναζήτησης Μόντε Κάρλο (Monte Carlo Tree Search - MCTS) και των τεχνικών βαθιάς επιβλεπόμενης μάθησης για την προώθηση των τεχνολογιών αυτόνομης οδήγησης. Η MCTS είναι ιδιαίτερα κατάλληλη για δυναμικά και απρόβλεπτα περιβάλλοντα, όπως η αυτόνομη οδήγηση, λόγω των ισχυρών δυνατοτήτων λήψης αποφάσεων σε περίπλοκα σενάρια. Σε συνδυασμό με τα Βαθιά Νευρωνικά Δίκτυα (Deep Neural Networks - DNNs), τα οποία διακρίνονται στην αναγνώριση μοτίβων και την προγνωστική μοντελοποίηση από μεγάλα σύνολα δεδομένων, αυτές οι τεχνολογίες θα μπορούσαν ενδεχομένως να φέρουν επανάσταση στα συστήματα διαχείρισης κυκλοφορίας.

Μια αναπαράσταση του MCTS σε πλαίσιο Μαρκοβιανής Διαδικασίας Λήψης Αποφάσεων (Markov Decision Process) αναπτύχθηκε ειδικά για σενάρια οδήγησης χωρίς λωρίδες, βελτιώνοντας τις παραδοσιακές προσεγγίσεις MCTS για να χειρίζονται καλύτερα τις λεπτομερείς απαιτήσεις αυτού του περιβάλλοντος. Η διπλωματική εργασία αυτή βασίζεται στο υπάρχον μοντέλο MCTS που ανέπτυξε ο Παντελής Γιανκουλίδης στην διπλωματική του εργασία το 2023, εστιάζοντας αρχικά στην βελτίωση του τρόπου επεξεργασίας των πληροφοριών μιας κατάστασης. Αυτές οι βελτιώσεις αύξησαν σημαντικά την λειτουργική αποδοτικότητα και αποτελεσματικότητα του πλαισίου MCTS, επιτρέποντάς του να διαχειρίζεται πιο επιδέξια καταστάσεις υψηλής πυκνότητας κυκλοφορίας.

Περαιτέρω πρόοδος επιτεύχθηκε με την ένταξη ενός νευρωνικού δικτύου στο πλαίσιο MCTS για να καθοδηγεί τη φάση επιλογής. Αυτή η ένταξη αξιοποίησε τις προγνωστικές ικανότητες των DNNs, επιτρέποντας πιο ενημερωμένη λήψη αποφάσεων και ταχύτερη εξερεύνηση κατά τη διαδικασία αναζήτησης δέντρου. Επιπλέον, διερευνήθηκε μια αυτόνομη προσέγγιση νευρωνικού δικτύου, σχεδιασμένη να λειτουργεί χωρίς τα εξερευνητικά οφέλη της MCTS, για να αξιολογηθεί η συγκριτική της αποτελεσματικότητα στη λήψη αποφάσεων.

Η ενδελεχής πειραματική αξιολόγηση της προσέγγισής μας, δεικνύει ότι το βελτιωμένο πλαίσιο MCTS, υποστηριζόμενο από την καθοδήγηση του νευρωνικού δικτύου, βελτίωσε σημαντικά τα αποτελέσματα διαχείρισης της κυκλοφορίας. Καίριες μετρήσεις, όπως η ασφάλεια, μέσω της μείωσης των ποσοστών σύγκρουσης, και η αποδοτικότητα, με τη βελτιστοποίηση της ταχύτητας και της ροής της κυκλοφορίας, επισημαίνουν τα ουσιαστικά οφέλη αυτής της ενσωματωμένης προσέγγισης. Αυτά τα αποτελέσματα υπογραμμίζουν το δυναμικό του συνδυασμού της MCTS με τις τεχνολογίες νευρωνικών δικτύων για να αλλάζουν ριζικά τις διαδικασίες λήψης αποφάσεων σε αυτόνομα οχηματικά περιβάλλοντα.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Georgios Chalkiadakis, for accepting me into his team and guiding me throughout this thesis journey. I am also thankful to the thesis committee members, Prof. Ioannis Papamichail and Prof. Michail G. Lagoudakis, for dedicating their valuable time to my presentation.

I would also like to specially thank Iason Chrysomallis, a PhD student at the Technical University of Crete, whose great ideas and useful comments significantly enriched my work. I extend my gratitude and particular thanks and appreciation to Dimitrios Troullinos, also a PhD student at the Technical University of Crete, for his constant guidance and insightful contributions throughout the whole development of my thesis.

Last, but definitely not least, I want to deeply thank my family from the bottom of my heart for the consistent support they provided me with during my academic journey. Additionally, I am grateful to my friends who have inspired me and stood by me, making this academic journey a shared and memorable experience.

List of Figures

List of Tables

List of Abbreviations

- **CAV**: Connected and Automated Vehicles
- **AI**: Artificial Intelligence
- **ML**: Machine Learning
- **RL**: Reinforcement Learning
- **ANN**: Artificial Neural Network
- **NN**: Neural Network
- **DNN**: Deep Neural Network
- **RNN**: Recurrent Neural Network
- **CNN**: Convolutional Neural Network
- **ReLU**: Rectified Linear Unit
- **tanH**: Hyperbolic Tangent
- **MSE**: Mean Squared Error
- **SGD**: Stochastic Gradient Descent
- **Adam**: Adaptive Moment Estimation
- **MDP**: Markov Decision Process
- **POMDP**: Partially Observable Markov Decision Process
- **MCTS**: Monte Carlo Tree Search
- **UCB**: Upper Confidence Bound
- **UCT**: Upper Confidence Trees
- **PUCT**: Predictive Upper Confidence Trees
- **SUMO**: Simulation of Urban MObility

Contents

Chapter 1

Introduction

Vehicular traffic has long been a cornerstone of modern transportation, essential for the movement of both people and goods, enhancing life's pace and convenience. Yet, the increasing traffic congestion brings with it a host of challenges, like higher emissions, increased travel times, and significant economic losses. To mitigate these issues, the automobile industry and researchers have focused on developing autonomous driving technologies and innovative traffic models. Autonomous vehicles, powered by advanced machine learning algorithms and equipped with state-of-the-art sensors and actuators, promise to revolutionize traffic systems by enhancing safety and efficiency. These vehicles are designed to reduce human error, which is the cause of most traffic accidents [9], and offer independence to those unable to drive due to disabilities. While current automated cars have achieved a respectable level of autonomy, they mostly operate within traditional, lane-based traffic environments. These environments, structured around well-defined lanes, have dominated the approach to vehicular navigation and control. However, they often fail to optimize traffic flow dynamically and are prone to congestion. The common strategies deployed in these settings include sophisticated path-planning, motion control, and environmental perception to navigate the complexities of lane-based traffic and aim for higher automation levels [10].

Now, the potential of connected and automated vehicles (CAVs) surpasses traditional traffic paradigms to allow for lane-free traffic. The conventional multi-lane structure is redesigned, specifically from the TrafficFluid [11] project, which recommends a dynamic vehicular flow, where the dominant lane-based structure is lifted and vehicles have the liberty to maneuver laterally across the entire road. This approach not only enhances road capacity by utilizing lateral space more efficiently, but also addresses the challenges posed by anisotropy [12], a phenomenon where vehicles are largely influenced by the traffic ahead, leading to wave-like congestion patterns. By enabling vehicles to respond to both forward and backward traffic through nudging [11], lane-free environments introduce a novel paradigm for CAVS that significantly reduces congestion and improves substantially the capacity of the road.

Such environments empower vehicles to perform overtaking maneuvers without the constraints of lane changing, providing a smooth driving experience that enhances both safety and comfort. This flexibility is particularly advantageous in highly automated traffic systems, where vehicles communicate or have information for each other and coordinate their movements, ensuring optimal spacing and speed adjustments in real-time. The shift to-

wards lane-free traffic not only promises a reduction in typical traffic bottlenecks, but also allows for a more creative approach to traffic design and management, potentially revolutionizing how future urban infrastructure is planned and developed. By adopting this cutting-edge traffic model, the research aims to demonstrate a marked improvement in traffic efficiency and safety, setting a new standard for autonomous vehicular technology in complex driving environments.

Researchers inspired by the TrafficFluid concept have proposed various vehicle navigation and path planning strategies. Approaches range from Control Theory [11, 13] to Multi-Agent Decision Making [14, 15] and heuristic-based strategies for nudging [11]. Additionally, [14] utilized the max-plus algorithm for collaborative communication and [16] used implicit imitation learning with Deep Reinforcement Learning [17].

Indeed, Artificial Intelligence (AI) [18] has revolutionized our ability to process information and make automated decisions, with Machine Learning (ML) [19] as a crucial sub-field. ML algorithms learn to identify patterns and make predictions from data, allowing systems to improve autonomously. This approach shifts away from traditional rule-based programming, offering a more adaptable solution to complex challenges. Supervised Learning [20], a significant class of methods in ML, trains models on labeled data to accurately predict outcomes on new, unseen data. Artificial Neural Networks (ANNs) [21], inspired by the human brain, enable learning at multiple abstraction levels, making them suitable for complex tasks. Deep learning [22], a branch of ML, utilizes multi-layered Neural Networks (NNs) to capture complex data patterns. By increasing the depth of these networks, deep learning models can learn more intricate features, transforming inputs into increasingly sophisticated outputs. This capability has led to major advances in various fields like natural language processing [23], computer vision [24] and decision-making, broadening the impact of ML technologies.

MCTS [1, 25] is a versatile decision-making algorithm widely used across various domains, such as gaming, robotics, and autonomous systems. It combines the robustness of tree search with the randomness of Monte Carlo simulations to efficiently navigate large decision spaces. By progressively building a search tree and utilizing random trials to evaluate potential moves, MCTS balances exploration of new strategies with the exploitation of known successful paths. This method has been particularly effective in complex games, like Go [26, 27], where it significantly outperformed traditional AI techniques. Its ability to adapt to changing scenarios makes it invaluable for applications that require dynamic decision-making under uncertainty [28].

MCTS has proven particularly effective in autonomous driving for its handling of sequential decision-making and motion planning [29, 30, 31, 32]. This algorithm excels in environments that demand rapid, dynamic decisions, as it simulates multiple potential future scenarios to navigate vehicles through complex traffic situations safely and efficiently. By employing a blend of probabilistic modeling and optimization techniques, MCTS enables vehicles to predict and adapt to quick changes in traffic conditions consistently. Integrating MCTS with advanced learning methods further enhances decision-making capabilities, ensuring autonomous systems are well-equipped to maintain high performance across diverse and challenging traffic environments.

1.1 Contributions

Serving as a source of inspiration, Karalakou's study [8] on the use of Deep Reinforcement Learning (Deep-RL) for policy optimization in a lane-free, ring-road environment provided an essential foundation for our work. Utilizing a Markov Decision Process (MDP) framework, this approach enabled the effective representation of complex driving behaviors within a controlled setting. We adopted Karalakou's MDP environment as the basis for developing our MCTS-MDP framework [33], tailored to address the challenges of autonomous lane-free driving. Building upon this groundwork, our research significantly leverages the innovative work of Pantelis Giankoulidis [34], who applied MCTS to navigate autonomous vehicles in a lane-free environment. Giankoulidis's work, serves as a cornerstone for our thesis. Our approach aims to refine and expand upon his initial model by integrating NN knowledge and guidance into the MCTS framework, enhancing decision-making processes. We benchmark our advancements against Giankoulidis's original MCTS algorithm, highlighting our improvements and delineating the progression in a comprehensive and accessible manner.

More specifically, we started, by refining the existing MCTS of [34]. Our enhancements involved fine-tuning parameters and redefining how the algorithm interprets state information, which significantly boosted the overall efficiency and effectiveness of the MCTS. These improvements enabled the algorithm to perform robustly in complex scenarios featuring dense traffic, where the original version initially exhibited inferior performance.

Further advancements, and the primary goal of this thesis, involve the integration of a NN into the selection phase of the MCTS. Our inspiration to apply NN guidance in autonomous driving, came from the success of Deep Neural Network (DNN) and MCTS integration in games like Computer Go [28] and Hex [35], that aimed on enhancing move prediction and evaluation via supervised learning methods. This integration leveraged a NN trained through offline self-play simulations, utilizing data produced by our improved MCTS model [36]. The incorporation of NN's prior knowledge, enabled a predictive approach within the MCTS, enhancing the algorithm's ability to quickly and effectively navigate through the search space to identify optimal solutions. This new methodology we employ in our thesis, reduces the necessity for extensive computational MCTS iterations and achieved the same quality results exceptionally faster, significantly accelerating the decision-making process. The refined algorithm not only demonstrates superior performance in key metrics, such as safety with fewer vehicle collisions and operational efficiency with higher vehicle speeds, but also confirmed the potential utility of NN guidance in MCTS for complex decision-making environments, such as lane-free traffic.

To thoroughly evaluate our methods, we also examined the performance of a standalone NN, independent of search strategy integration. This approach is considered greedy, relying on a deterministic algorithm devoid of exploratory mechanisms. Predictably, it fell short compared to other algorithms, highlighting its tendency to settle on sub-optimal solutions, due to the lack of exploration.

1.2 Thesis Outline

In what follows, Chapter 2 lays the theoretical groundwork necessary for understanding MCTS, deep learning, NN architectures, and their probabilistic learning integration. It further explores how NNs are integrated with MCTS to enhance decision-making processes. Then, Chapter 3 compares traditional lane-based traffic environments with the more dynamic lane-free settings and discusses the role of MCTS in motion planning for autonomous vehicles. In Chapter 4, the discussion transitions to our specific approach, detailing the MCTS-MDP framework, the adaptations in plain MCTS, the integration of NN-Guided MCTS, and the utilization of NNs without search mechanisms. Chapter 5 dives into the experimental evaluation with a thorough description on the simulation setup, the examined methods, along with a systematic analysis of the results. Finally, Chapter 6 wraps up the thesis with a summary of findings and potential avenues for future work.

Chapter 2

Theoretical Background

This chapter lays the theoretical groundwork for the methodologies used in the thesis. It begins with an overview of Monte Carlo methods, particularly focusing on the Monte Carlo Tree Search (MCTS) and its application within decision-making frameworks. The discussion then shifts to Markov Decision Processes (MDP), detailing how they support the functioning of MCTS. The chapter also extensively covers machine learning basics, neural network (NN) structures, deep learning architectures, and the necessary aspects of training and optimization. Finally, the integration of MCTS with NNs is explored, highlighting the enhancements and challenges involved.

2.1 Monte Carlo Methods

Monte Carlo methods [37] are a broad class of computational algorithms that rely on repeated random sampling and statistical analysis to obtain numerical results. Their fundamental principle is to use randomness to solve problems that might be deterministic by nature. They achieve that, by generating random variables, that simulate the behavior of a complicated system or process. The outcomes of these simulations are then analyzed statistically to estimate parameters of interest, such as averages, variances, or probabilities.

These methods are particularly useful for solving complex problems with a high degree of uncertainty, and in simulations where direct analytical solutions are challenging or impossible to find. Monte Carlo methods are widely applied in various fields, especially in areas requiring the modeling of complex phenomena or the evaluation of probabilistic outcomes [38].

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [1, 25] is an innovative algorithm that has seen widespread application across various domains, demonstrating its versatility and efficiency in tackling complex decision-making challenges. MCTS combines the standards of Monte Carlo

methods, which utilize random sampling and statistical analysis, with the structured approach of tree-based search strategies. Unlike traditional search algorithms that rely upon exhaustive exploration on the entire search space, MCTS focuses on selectively sampling and investigating the most promising regions.

The core concept behind MCTS involves progressively constructing a search tree through the simulation of multiple random trials, often referred to as rollouts or playouts, starting from the existing state. These simulations proceed until reaching either a terminal state or a specified depth. Subsequently, the outcomes of these simulations are fed back through the tree, enhancing the statistics for each traversed node, including metrics such as the frequency of visits and win ratios (e.g., in games) or a scalar evaluation value.

As the search advances, MCTS manages the balance between exploration and exploitation, choosing actions by weighing the potential exploitation of highly successful moves against the need to investigate less visited or entirely new paths. This equilibrium is achieved by strategically determining which moves or nodes to explore next, ensuring a thorough yet focused traversal of the search landscape. This balance is critical in environments where the decision space is too large for exhaustive search methods to be feasible to apply. The algorithm's ability to learn from simulated outcomes of decisions, updating its strategy with each iteration, makes it exceptionally adaptable to changing conditions and previously unseen scenarios.

MCTS stands out from other searching and decision making algorithms for many reasons [1, 25], namely:

- **Flexibility in Complex Environments and Asymmetry:** MCTS excels in navigating through intricate and strategic scenarios, effectively managing large search spaces where conventional algorithms might falter due to the sheer number of possible outcomes. That happens, because the algorithm prioritizes more promising or interesting nodes, focusing its resources on parts of the tree that are more likely to lead to successful outcomes.
- **Adaptability and Learning :** The algorithm adjusts its strategy based on the outcomes of simulations, refining its approach with each iteration to better navigate the decision space and improve its decision making capabilities.
- **Handling of Uncertainty and Imperfect Information:** MCTS is particularly suited for environments where complete information is not available, relying on statistical sampling to make informed decisions. Also If there is an already existing reward system, MCTS does not rely on domain-specific knowledge, making it a versatile tool that can be applied across a wide range of problem domains without the need for custom-tailored heuristics.
- **Anytime:** The algorithm can be interrupted at any stage to provide the best solution found up to that point, making it flexible and practical for real-world applications where time constraints are a factor.

A visual illustration of an MCTS search space decision-tree is provided in the Figure 2.1 below. This showcases the asymmetry of different exploration depths between the various paths that have been expanded. The figure was taken from [1].

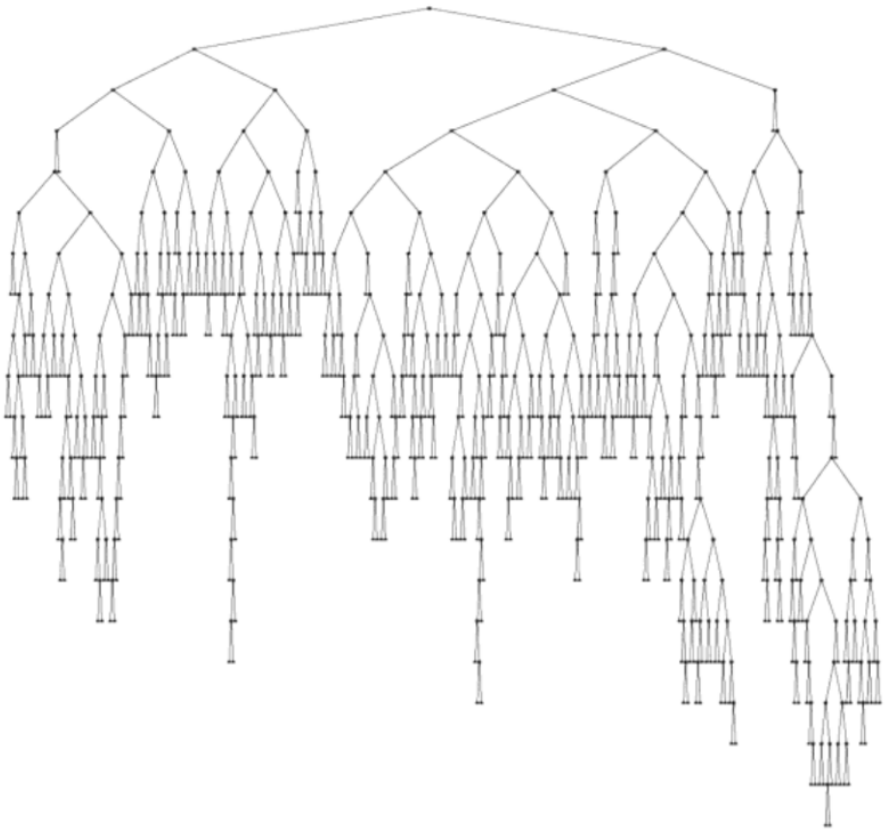


Figure 2.1: Visualization of an MCTS Search Space, showcasing the Assymetry of Exploration Depth between different Paths [1].

Despite the versatility and broad applicability of MCTS across various domains, it is not always the ideal solution for every problem. The algorithm's probabilistic approach, while powerful in vast and uncertain search spaces, does not guarantee the identification of the optimal solution, as it prioritizes statistical significance over exhaustive exploration.

MCTS has emerged as a cornerstone algorithm in the field of Artificial Intelligence (AI) [18], showcasing its versatility and effectiveness , starting from strategic games [26] and reaching across a wide range of domains. One of the most notable successes of MCTS was its application in AlphaGo [27], the computer program developed by DeepMind that defeated the world champion of Go, a board game known for its profound strategic complexity. This victory underscored MCTS's capability to handle decision-making processes in games that were previously considered beyond the reach of computer algorithms due to their intricate dynamics and the sheer number of possible moves.

Beyond the realm of games, MCTS finds applications in real-world problems [28] that require strategic planning under uncertainty. A great usage example that will concern us in this thesis, is motion planning. MCTS can be used to navigate through complex environments where multiple paths and obstacles exist, optimizing for efficiency and safety. In the domain of autonomous-driving to the best of our knowledge, there has been only one application of the MCTS algorithm specifically in lane free based traffic [34], that we examine in detail , along with the uses of MCTS in motion planning in Section 3.2.2.

2.3 Markov Decision Process

MDP Formal Definition

A Markov Decision Process (MDP) [39] provides a mathematical framework for modeling decision-making in environments with stochastic outcomes, formally defined as a 5-tuple (S, A, P, R, γ), where:

- S is the state space, a finite set of all possible states. In more complex scenarios, S can be continuous.
- A is the action space, encompassing all possible actions. This space can also be discrete or continuous, depending on the application.
- P represents the transition probability function,

$$P(s', s, a) = \Pr(St + 1 = s' \mid St = s, At = a) \tag{2.1}$$

  defining the probability of moving from state s to state s' under action a.
- R is the reward function, $R(s, a)$, providing immediate feedback by assigning rewards for actions taken in specific states.
- γ is the discount factor, $0 \leq \gamma < 1$, which quantifies the importance of future rewards. The expected reward at time t is:

$$R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots$$

Markov Property: The core principle of MDPs is the Markov property which implies memorylessness, asserting that the probability of transitioning to the next state depends only on the current state and action, independent of the preceding sequence of events. This property simplifies analysis and computation by ensuring that the state transition dynamics are fully captured by the current state and action, without needing to account for the entire history of states, making MDPs a powerful framework for modeling and solving sequential decision-making problems.

Observable Environments: In a fully observable environment, the agent has complete information about the current state. Conversely, in a partially observable environment, the agent has limited information, which necessitates the use of different approaches such as Partially Observable Markov Decision Processes (POMDPs) [40].

Action Space: The action space can be either discrete, consisting of a finite set of actions, or continuous, comprising an infinite set of actions represented by real-valued vectors.

A visual illustration of a random MDP example with various states, actions and rewards is provided in the Figure 2.2 below.

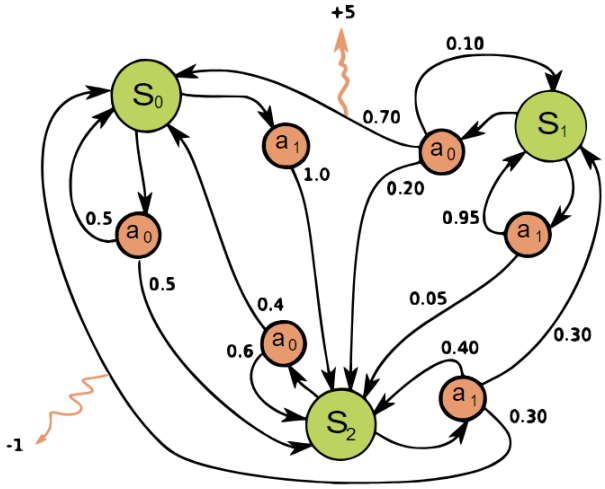


Figure 2.2: Representation of an MDP.

Policy

A policy π guides an agent's actions within its environment, manifesting in two distinct forms: deterministic, where $\pi(s)$ specifies a singular action a for each state s, and stochastic, where $\pi(a \mid s)$ provides a probability distribution over possible actions given a state, allowing for varied responses based on the same state. In other words, a policy determines the agent's behaviour at every possible state, therefore it fully describes the agent's decisions.
Methods that generate a policy can be either online or offline [41].

- **Offline Methods:** These involve a precomputed policy that has found the best action for every possible state. During execution, actions are chosen based on this established policy, allowing for decision-making without real-time computation.
- **Online Methods:** By contrast, these methods perform execution and planning in parallel. They adaptively calculate the next action directly from the current state, considering only states that are currently accessible.

The general goal within an MDP framework is to discover an optimal policy π^*, which maximizes the expected long-term return (reward) from any initial state, formalized as

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(St, At) \mid \pi\right] \tag{2.2}$$

This involves calculating value functions [42]:

- **State Value Function:**

$$V^{\pi}(s) = \mathbb{E}\left[R_{t+1} + \gamma V^{\pi}(S_{t+1}) \mid S_t = s\right] \tag{2.3}$$

  represents the expected return from state s under policy π, emphasizing the importance of both immediate rewards and future rewards discounted by γ.

- **Action Value Function:**

$$q^{\pi}(s,a) = \mathbb{E}\left[R_{t+1} + \gamma q^{\pi}(s_{t+1}, a_{t+1}) \mid S_t = s, A_t = a\right] \tag{2.4}$$

estimates the expected return of taking action a in state s and then continuing to follow the chosen policy π.

Policy Opimization using MCTS

MCTS is an algorithm that can be used to optimize policies within MDPs [28] by employing forward random sampling, a technique that simulates various action paths to evaluate their potential outcomes and estimate their long-term rewards. This method allows MCTS to explore the decision space efficiently, focusing the searching on the more promising paths, thus to optimize the policy within the constraints of finite state and action spaces of MDPs. More specifically, by simulating the consequences of actions from the current state and using the results to inform decision-making, MCTS effectively captures the domain's MDP structure, without explicitly going through all states and transitions. This approach is particularly useful in complex environments where exhaustive exploration is computationally infeasible, enabling the algorithm to approximate optimal policies by balancing between exploration of new paths and exploitation of known successful paths.

In conclusion MCTS stands out as one of the most commonly used and powerful online methods , dynamically adjusting actions based on ongoing simulations and evaluations of reachable states. Its functionality is going to be explained in detail in the next paragraph 2.1.3.

2.4 The MCTS Algorithm in Detail

MCTS representing MDPs

MCTS solves MDPs [33] by representing them as a search tree where nodes represent states of the MDP's state space, and edges represent actions from the action space. In this tree, each node is connected to as many edges as there are valid state-action pairs for that state, implementing the transition probabilities of the MDP. Nodes store a score symbolizing the utility of that state regarding solving the MDP, approximating with this way the state-value function. The algorithm enhances its understanding of state and action value functions by performing Monte Carlo simulations within a subtree of the node, continuously improving its predictions after each simulation [1, 25].

Algorithm Structure

A visual illustration of an MCTS phases is provided in the Figure 2.3 below. This process is repeated X times, where X represents the number of maximum iteration counts. The figure was taken from [2].

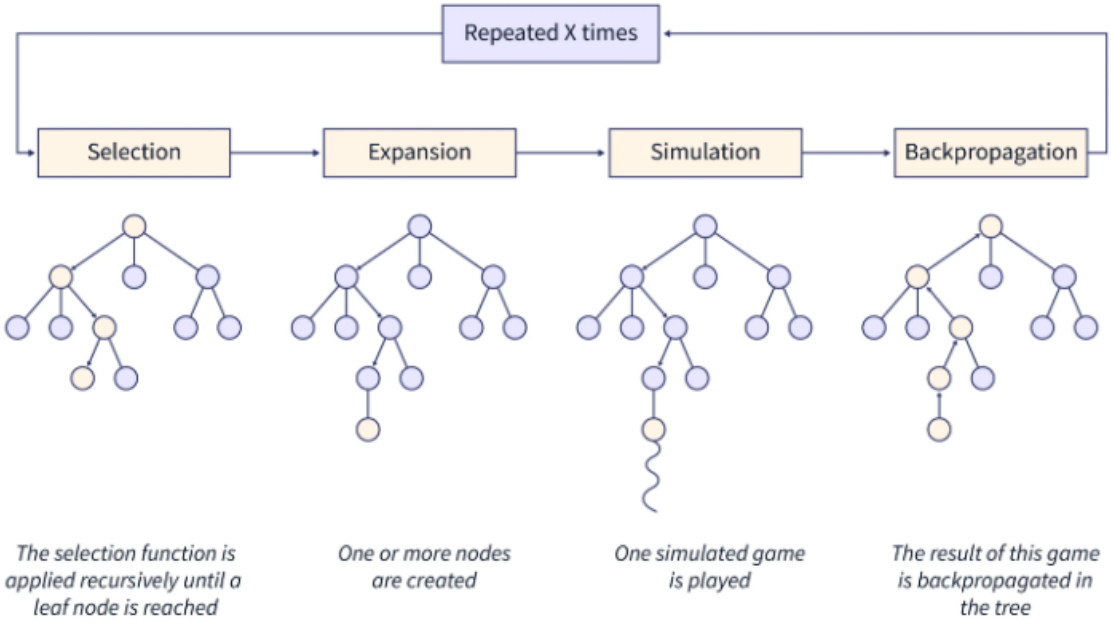


Figure 2.3: Representation of MCTS phases [2].

- **Selection Phase:**

  In the selection phase, the algorithm traverses the tree from the root node, making decisions at each node until it reaches a leaf node that has not been fully explored. The decision of which child node to visit at each step is guided by the Upper Confidence Bound (UCB) for Trees - Upper Confidence Trees (UCT) [43] formula:

  The Upper Confidence Bound for Trees (UCT) for a state s and action a is given by:

$$\mathrm{UCT}(s,a) = Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \tag{2.5}$$

  where:

  - $Q(s,a)$ represents the average reward received after taking action a from state s.
  - $N(s)$ is the total number of times state s has been visited.
  - $N(s,a)$ is the number of times action a has been taken from state s.
  - and C is a constant that balances exploration and exploitation.

  The selection strategy ensures that the algorithm explores nodes with high potential rewards (exploitation) and also investigates less-visited nodes to discover new strategies (exploration). C is the constant that determines the exploitation-exploration trade-off. A large C means that the policy is giving big values on unvisited states, thus prioritizing the exploration term. By contrast, a small C leads to a more greedy policy, with more exploitation on visited states with high rewards and not so much exploration on unvisited states. The policy is set to select in each iteration the node with the highest UCT value, except the last iteration, that it selects the action

with the highest average reward greedily. That is because, after the last iteration the algorithm comes to an end, so there is not going to happen any more exploration in other unvisited states. This phase is crucial for efficiently navigating the search space by focusing on promising areas while avoiding premature convergence on suboptimal paths.

- **Expansion Phase:**

  Once the selection phase reaches a leaf node that represents a non-terminal state and is not fully expanded, the expansion phase begins. Here, one or more child nodes are added to the tree, each representing a possible action from the current state. The criteria for expansion, depend on most cases in the number of times the node has been visited compared to a small threshold N , that it must be greater than it , to expand. This phase is process of broadening the search tree's horizon, adding new paths to be evaluated through simulation.

- **Simulation Phase:**

  During the simulation phase, the algorithm performs a rollout from the lastly expanded node by simulating a sequence of actions until a terminal state is reached or a certain depth limit is achieved. The simulation policy in this phase, involves random selections or heuristic-based choices, to approximate the probable outcomes of actions. The purpose of the simulation is to evaluate the potential utility of the new paths added during the expansion phase, providing a rough estimate of their value without requiring exhaustive exploration.

- **Backpropagation Phase:**

  After the simulation phase ends, the result or else the value-score of the terminal node, is propagated back through the tree along the path taken during the selection phase, so that all ancestors of the terminal node get the new information. This backpropagation updates the estimated value functions (Q values) and visit counts (N values) for each node along the path, updating the tree's knowledge base. Through this process, the algorithm iteratively adapts to which actions lead to the best outcomes, enhancing its decision-making accuracy.

Iterative Tree Construction

The MCTS tree starts with a root node representing the current state. As the algorithm iterates deeper, it expands the tree by adding nodes (states) and edges (actions). It does this by using a particular tree policy, in our case UCT. Each node can have as many children as there are possible actions from that state, with leaf nodes representing terminal states of the MDP.

MCTS builds its search tree through a cyclic process of those four phases. This iterative approach allows the tree to evolve and adapt, gradually getting closer to the most promising areas of the decision space. The algorithm progresses in iterations, each adding to the tree's breadth and depth, until a stopping criterion such as a time limit or a specified number of iterations is met. This ensures that computational needs are concentrated on exploring and exploiting the most relevant parts of the decision space and avoiding the exhaustive search of the state space, making the process both efficient and effective.

MCTS can be stopped in any number of iterations. More iterations mean a deeper tree and closer to optimal solutions, but also it takes more time. Fewer iterations result in a quicker process but might not find the best solution. This flexibility allows MCTS to balance between accuracy and speed, depending on what's needed for every problem.

A visual illustration of MCTS flowchart is provided in the Figure 2.4 below. The figure was taken from [2].

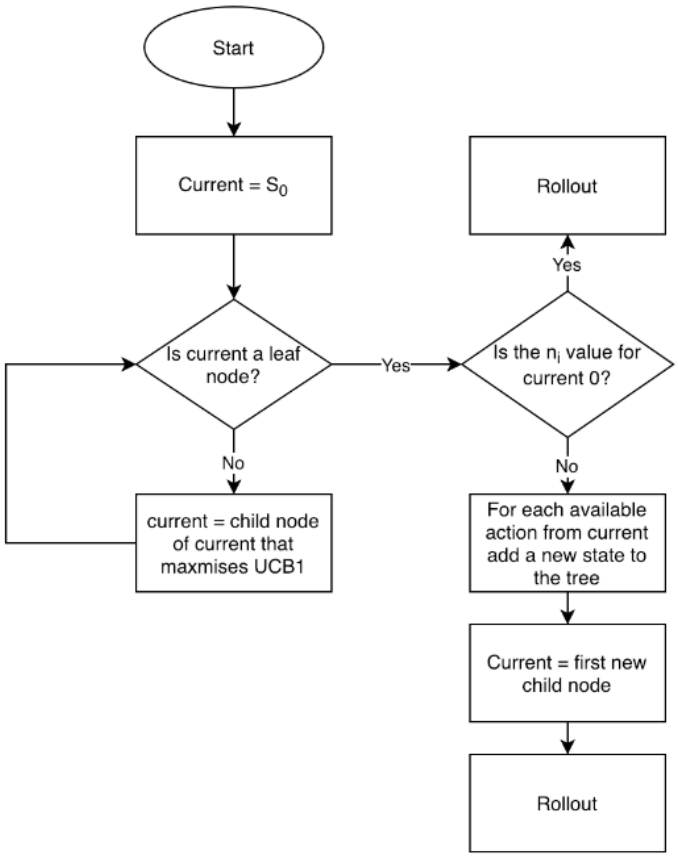


Figure 2.4: Flowchart of MCTS [2].

2.5 Deep Learning Foundations and Neural Network Architectures

This section introduces the core principles of deep learning and neural network architectures, explaining essential concepts such as machine learning fundamentals, various network designs, activation and loss functions, and the intricacies of training and optimizing these networks.

2.5.1 Machine Learning

Machine Learning (ML) [19] is a sub-field of AI that concentrates on developing algorithms and models capable of learning from data. This makes computers to be able to

make predictions or decisions without being explicitly programmed for specific tasks. Instead, ML systems evolve and enhance their performance over time through experience and learning. This approach is very different from traditional programming and offers a dynamic and adaptable framework for handling complex problems.

The evolution of ML is marked by the transition from rule-based systems to models that learn from data. Early efforts in AI tried to encode human knowledge into machines directly, but this proved to be impractical for complex tasks. ML allows for a more sophisticated approach to understanding and interacting with the world, leveraging data to uncover patterns and insights that would be impossible to detect manually. More specifically, the introduction of neural networks and the subsequent development of deep learning models have dramatically expanded the capabilities of ML, enabling breakthroughs in various technological areas, such as computer vision, natural language processing, and pattern recognition.

Types of Learning

ML can be categorized into three primary types of learning methods:

- **Supervised Learning:** This involves learning a function that maps an input to an output based on example input-output pairs [20]. It uses a training set that consists of labeled examples, where each example is a pair consisting of an input object (typically a vector of features) and a desired output value (the label). The goal is to learn a model that, given unseen inputs, can accurately predict the corresponding output.
- **Unsupervised Learning:** Unlike supervised learning, unsupervised learning deals with training sets of unlabeled data [44]. The system tries to learn the patterns and the structure from the data without any explicit instructions on what to predict. It includes tasks such as clustering and dimensionality reduction, where the aim is to find inherent structures in the input data.
- **Reinforcement Learning (RL):** RL is a type of learning where an agent learns to make decisions by taking actions in an environment to achieve some goals [17]. The agent learns from its own, from the consequences of its actions, rather than from explicit teaching, adjusting its strategy to maximize some kind of cumulative reward.

Supervised Learning

In this thesis we focus and use supervised learning, therefore we present the relevant background in more detail below. As mentioned already, supervised learning algorithms are designed to learn a mapping from inputs to outputs, given a labeled dataset. This dataset comprises examples of inputs paired with the correct outputs, which the algorithm uses to learn the mapping. Once trained, the model can apply this mapping to new, unseen inputs to predict the outputs. In practical applications, the challenge of this method lies not only in designing and training models but also in ensuring their ability to generalize from the training data to new, unseen data. This involves not just capturing the underlying patterns in the training data but also avoiding overfitting, where the model learns

the noise in the data rather than the intended patterns.

A visual illustration of how Supervised Learning method works is provided in the Figure 2.5 below. The figure was taken from [3].

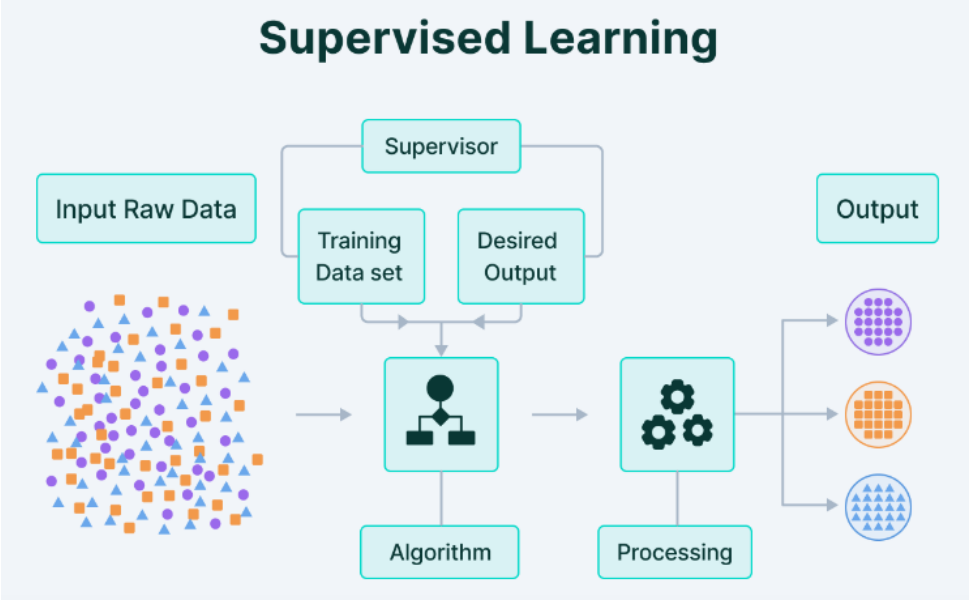


Figure 2.5: Supervised Learning Method Visual Explanation [3].

Supervised Learning is divided mainly into two categories [45]:

- **Classification:** In classification tasks, the output variable is a category, such as spam or not spam in email filtering, or cat, dog, in image recognition. The goal is to accurately assign each input to one of the predefined categories. Classification can be either two dimensional with only two classes or it also can be multi-class, meaning that the task is to classify inputs into one of several categories.
- **Regression:** Regression tasks involve predicting a continuous quantity. For example, predicting the price of a house based on its features is a regression problem. The output is a continuous value representing the magnitude of some characteristics.

One of the most common scenarios of using ML algorithms and specifically Classification is the one of handwritten digit recognition. In this scenario, each digit is represented as an image, which in turn is represented by a vector, denoted x, filled with real numbers. The challenge is to design a system capable of accepting this vector as input and then accurately identifying the digit it represents, ranging from 0 to 9. The task is difficult to process due to the significant variation in individual handwriting styles.

Attempting to resolve this issue through manually crafted rules for distinguishing digits based on shapes is impossible [46]. That is because it leads to a continuously expanding array of rules and exceptions for each individually different written digit that yields to unsatisfying results.

Instead, Supervised Learning applies a more effective strategy, where a large set of digit images (e.g., the popular MNIST dataset), referred to as a training set, is utilized to

adjust the parameters of a model that can adapt. Each digit within the training set is pre-identified, by a process of individual inspection and labeling. Each digit's category is represented by a simple code called a target vector t, which identifies the digit. The operation of Classification algorithm, results in a function, denoted as y(x), which accepts a new digit image x as its input and outputs a vector y, that has similar form to the target vectors. The configuration of the function y(x) is created during the learning or training phase, which is informed by the training data. Once the model has been completely trained, it gains the ability to detect the identities of digit images in a test set, which are distinct from the images seen during training. The key objective in this pattern recognition [47] procedure is the model's ability to generalize. Meaning, that it can accurately classify new examples of digits that are not part of the training set.

A visual illustration of simple examples from a 2D linear Regression and a linear 2-Category Classification is provided in the Figure 2.6 below. The figure was taken from [3].

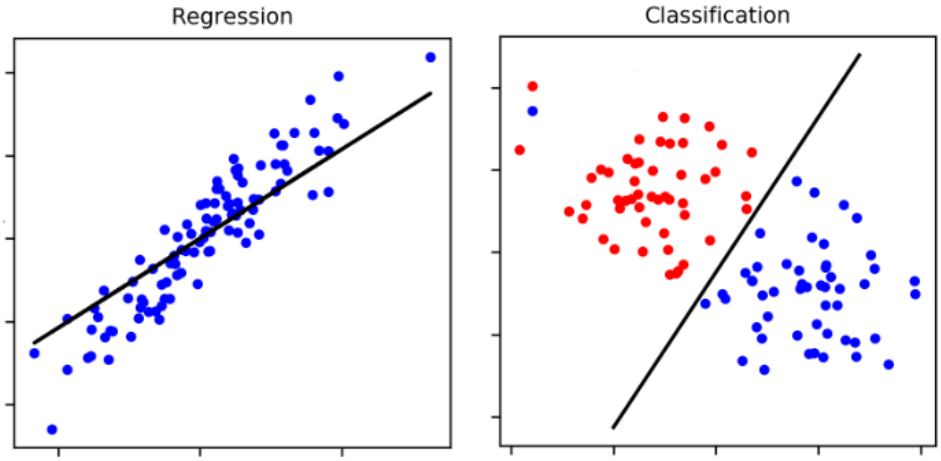


Figure 2.6: Example of a 2D Linear Regression and Linear 2-Category Classifier [3].

2.5.2 Artificial Neural Networks

Neural Networks (NNs) [21] are a computational model based on the structure and functions of biological neural networks. Information that flows through the network affects the structure of it, because a neural network learns, based on that input and output. Additionally, NNs [48] are inspired by the biological structure of perceptrons in the human brain and can work in various levels of abstraction, which makes them incredibly effective for problems that cannot be approached by rule-based systems.

Perceptron

The earliest and most simple type of NN is called perceptron and it is designed to simulate the function of a single neuron. It works by taking input features and multiplying them by weights, summing them up, and then applying an activation function to determine the output. It is used for binary classification tasks.

The perceptron's function is:

$$y = f\left(\sum_i w_i x_i + b\right) \tag{2.6}$$

- **Input Features** (x_i): The variables or attributes used as input for the perceptron.
- **Weights** (w_i): Parameters that the perceptron algorithm adjusts during training to improve prediction accuracy.
- **Bias** (b): A constant added to the weighted sum to adjust the output independently of the input values.
- **Activation Function** (f): A function applied to the weighted sum, including the bias, to determine the perceptron's output. Commonly, a function that is used to normalize the output y.

A visual illustration of a comparison between a biological neuron (left) and a perceptron (right) next to each other is provided in the Figure 2.7 below. The figure was taken from [4].

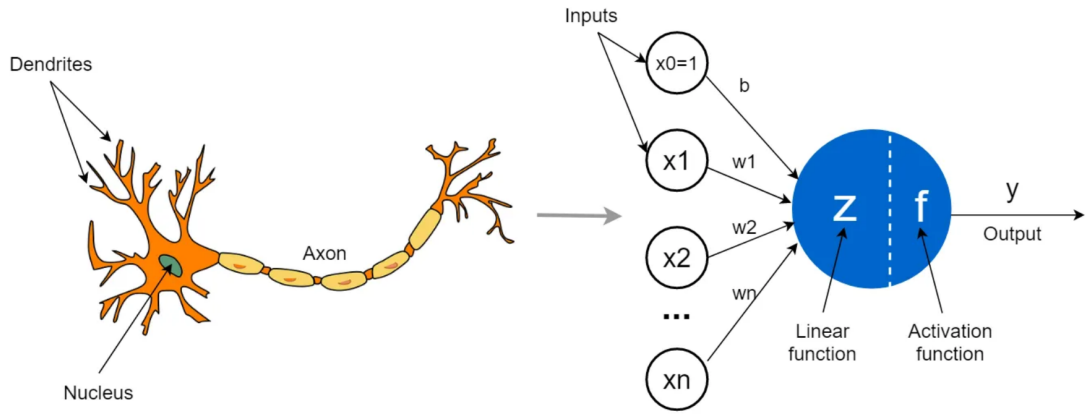


Figure 2.7: Comparison Between a Biological Neuron and a Perceptron [4].

2.5.3 Deep Learning Architectures

Deep learning [22] is a subset of ML that employs ANN with many layers to model complex patterns in large amounts of data. The evolution of deep learning architectures advanced upon realizing the potential of NNs through their depth. This means stacking numerous layers of perceptrons together, creating like these multilayer neurons. This depth allows the network to learn more abstract features at higher layers, a phenomenon absent in shallow networks. With each layer, the network can transform its inputs in more complex ways, effectively building a hierarchy of learned features.

The structure of Deep Neural Networks(DNN) consists of [22]:

- **Input Layer:** This is where the network receives its input data.
- **Hidden Layers:** One or more layers are stacked between the input and output layers, and are responsible for the network's complex computations. They are called "hidden" because they do not directly interact with the external environment.

- **Output Layer:** This layer produces the final outcomes of the network based on the processed information from the hidden layers.

A visual illustration of a fully connected feed forward NN, with n inputs, n hidden layers and n outputs is provided in the Figure 2.8 below. This showcases the asymmetry of different exploration depths between the various paths that have been expanded. The figure was taken from [4].

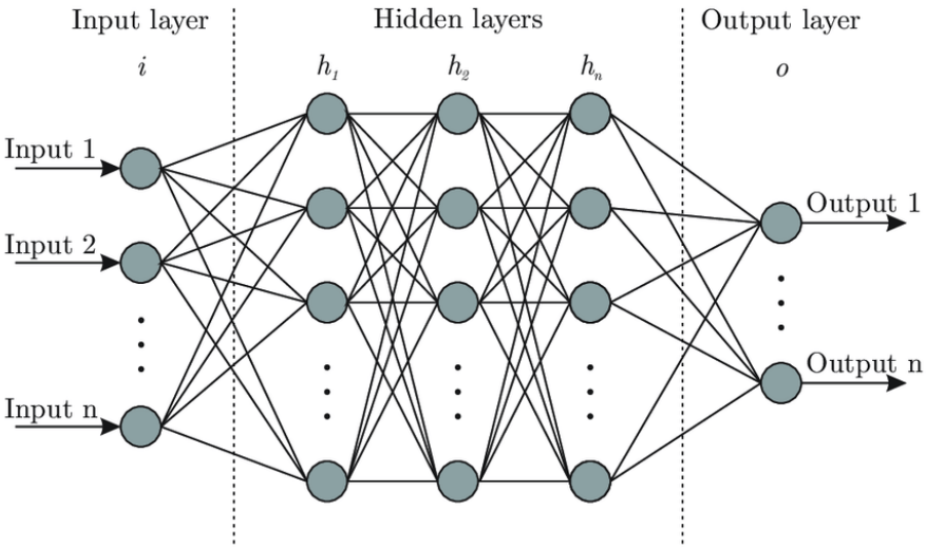


Figure 2.8: Example of a Fully Connected, Feed Forward NN [4].

Here are some different NNs types that have been implemented:

- **Feedforward Neural Networks:**Data flows in one direction, from input to output, without no cycles or loops in the network [49].
- **Convolutional Neural Networks (CNNs):** They use special layers called convolutional layers to process data arranged in grids, perfect for recognizing images and videos [50].
- **Recurrent Neural Networks (RNNs):** Designed to handle sequential data, these networks have loops to allow information to persist [51].

2.5.4 Activation Functions

Activation functions [52] are crucial elements within NN that introduce non-linear properties to the system. Without non-linearity, a NN would essentially act as a linear regression model, unable to handle the complex patterns found in most real-world data. These functions allow NN to learn and perform tasks at a higher level of computation. Their work is to determine whether a neuron should be activated or not, influencing the network's output and also, they help in controlling the output range, helping in predictions, and

making the training process stable and efficient.

These are some of the most commonly used activation functions:

- **Linear Activation Function:** This function maintains the proportionality of the input value, directly affecting the output in a linear fashion.

$$f(x) = cx \tag{2.7}$$

- **Logistic Sigmoid Activation Function:** It compresses the output to a range between 0 and 1, making it useful for probabilities.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

- **Hyperbolic Tangent Activation Function:** Outputs values between -1 and 1, useful for models where the direction of the effect is important.

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.9}$$

- **Rectified Linear Unit (ReLU):** It allows only positive values to pass through, introducing non-linearity while preventing vanishing gradient issues [53].

$$f(x) = \max(0, x) \tag{2.10}$$

- **Leaky ReLU:** It allows a small, positive gradient when the unit is not active, preventing dead neurons.

$$f(x) = \max(cx, x) \tag{2.11}$$

  where c is a small constant like 0.01.

- **Softmax:** This function outputs a probability distribution over various classes, making it suitable for multi-class classification problems in NNs. The Softmax function is typically applied in the final layer of a network to normalize the outputs, ensuring that they sum to one and represent probabilities that the input belongs to each class.

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{2.12}$$

  where x_i is the score (logit) for class i and K is the total number of classes.

A visual illustration of the mentioned activation functions (linear, sigmoid, tangent, ReLU, LeReLU, Softmax) plotted all in the same axis is provided in the Figure 2.9 below.

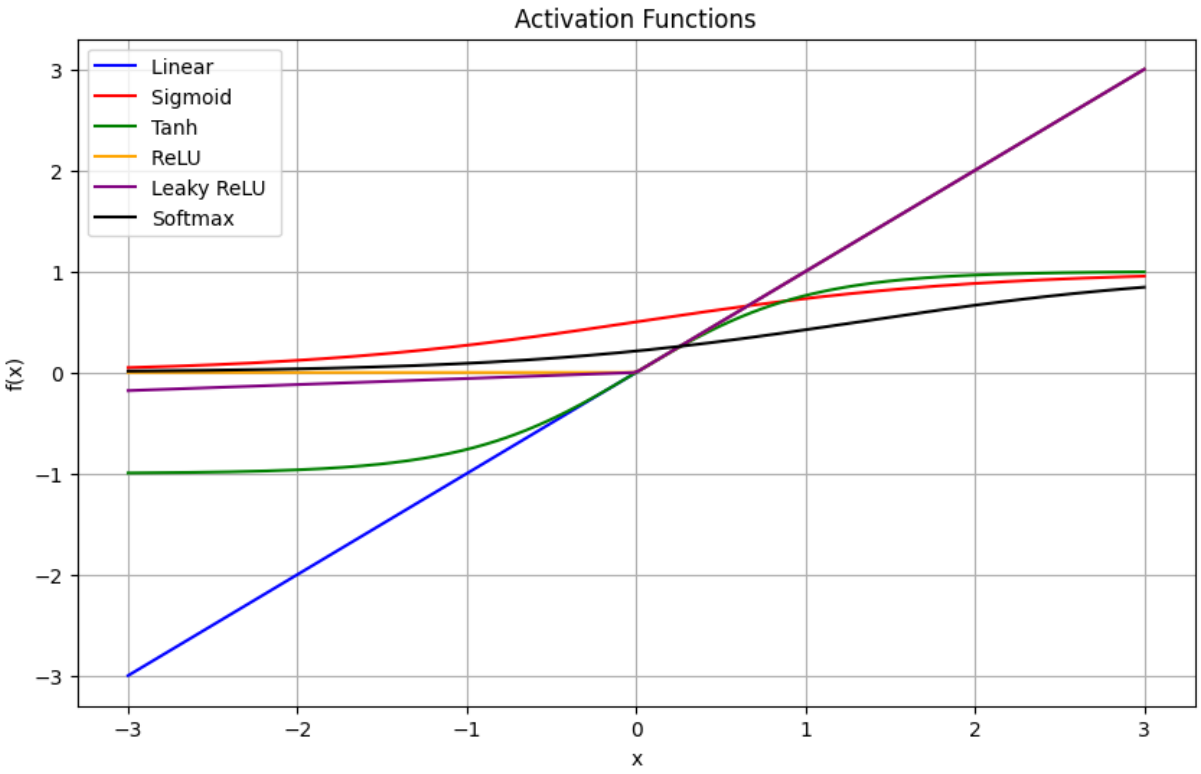


Figure 2.9: Graph of Linear, Sigmoid, Tangent, ReLU and LeReLU Activation Functions.

2.5.5 Loss Functions

When a NN begins its training process, its weights are initially assigned random values. This network takes input data and passes it through its various layers to generate an output. The generated output is then evaluated using a loss function, which compares it to the desired or expected outcome, effectively measuring the network's performance, calculated in accuracy.

Loss functions [54] play a leading role in the training of NNs, being a measure of how well the model's predictions match with the actual data. A loss function quantifies the difference between the predicted outputs of the network and the true outputs, providing a metric for the performance of the model. The choice of loss function depends on the specific task that is needed each time and it significantly influences the efficiency and effectiveness of the learning process.

These are some of the most known and used loss functions:

- **Mean Squared Error (MSE)**: Used for regression tasks, it calculates the average squared difference between the predicted values and the actual values [55]. It's defined as:

$$\text{MSE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.13}$$

  where y is the vector of n true values, and $\hat{y}$ is the vector of n predictions.

- **Cross-Entropy Loss**: Often used for classification tasks, it measures the performance of a classification model whose output is a probability value between 0 and

1. The loss increases as the predicted probability diverges from the actual label [56]. In binary classification, it's defined as:

$$\text{Cross-Entropy}(y, \hat{y}) = -\sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{2.14}$$

- **Hinge Loss**: Typically used for finding the decision boundary that separates different classes in the dataset with the largest possible margin, primarily used in Support Vector Machinesm [57]. It's defined as:

$$\text{Hinge Loss}(y, \hat{y}) = \max(0, 1 - y_i \cdot \hat{y}_i) \tag{2.15}$$

where y represents the true class labels as +1 and -1, and $\hat{y}$ are the predicted values.

2.5.6 Training and Optimization

Learning with Backpropagation

The learning process of the NN depends on its ability to adjust its weights to minimize the loss. This optimization is primarily achieved through an algorithm known as backpropagation [58]. Backpropagation works by calculating the loss function's gradient regarding each weight in the network and applying the chain rule of calculus.

If you have a NN function $y = f(x; w)$ where w represents the parameters (or weights) of the model, and a loss function $L(y, \hat{y})$ that measures the difference between the predicted output y and the true output $\hat{y}$, the gradient of the loss function with respect to the weights can be computed as follows:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w} \tag{2.16}$$

This calculation determines how much each weight contributed to the error. The error is then fed back through the network, allowing the system to update the weights in a direction that reduces the overall loss. The adjustments are made in a way that, over time, the network's output becomes more and more accurate in comparison to the expected results. This iterative process of forward prediction and backward error correction continues until the network's predictions are close to the actual values, making sure that the network is properly trained. The process of training NNs is also iterative and may require numerous epochs or else, passes through the full dataset, to achieve well enough performance.

Optimizers

Different optimizers [59] are used in this procedure above and they play a critical role in the training of NNs by using the gradients computed through backpropagation. These gradients inform the optimizer how to adjust the weights to minimize the loss function. The goal of any optimizer is to handle the multidimensional loss function and locate the set of weights that results in the lowest possible loss. This process of optimization is pivotal, because it directly influences the speed at which the network learns and the quality of the performance it achieves.

Here are some of the most known and used optimizers:

- **Stochastic Gradient Descent (SGD):** Traditional optimizer that updates each weight using the negative gradient multiplied by the learning rate [60].The equation that is used is:

$$W_{t+1} = W_t - \eta \cdot \nabla L(W_t) \tag{2.17}$$

  Where, W_{t+1} represents the updated weights at time $t+1$, W_t are the current weights at time t, η is the learning rate, and $\nabla L(W_t)$ is the gradient of the loss function with respect to the weights at time t.

- **Momentum:** Builds on SGD by using the direction of previous gradients to accelerate SGD in the relevant direction.

- **Adagrad:** Modifies the learning rate for each weight based on past gradients, useful for sparse data.

- **RMSprop:** Addresses Adagrad's aggressive, monotonically decreasing learning rate.

- **Adam (Adaptive Moment Estimation):** Combines ideas from RMSprop and Momentum by computing adaptive learning rates for each weight [61]. It is the optimizer that is most used, because it performs well in a very large number of different tasks.

A visual illustration of the loss function landscape using Gradient Descent is provided in the Figure 2.10 below.

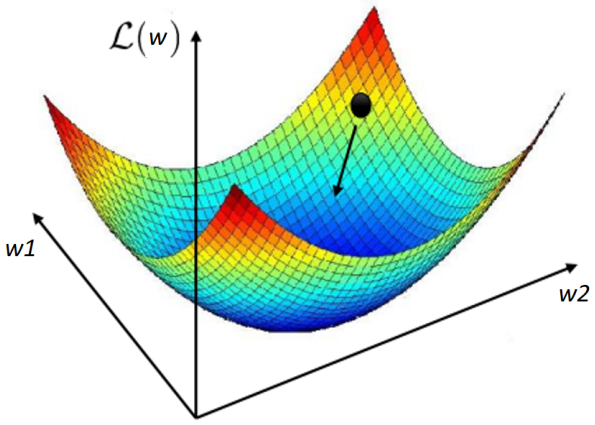


Figure 2.10: Visualization of the Loss Function Landscape using Gradient Descent.

Training Process

Finally, we present the complete training process of a NN. This process involves several key stages, each crucial for enabling the model to learn from data and make accurate predictions. Below is an overview of these stages:

- **Initialization:** The training process begins with the random assignment of weights to various connections between neurons. This step sets the initial state of the network before learning commences.

- **Feedforward Propagation:** Inputs are then fed into the network, passing through multiple layers. Each neuron processes the input and passes it forward to the next layer, culminating in an output.
- **Loss Calculation:** At the output layer, the network's prediction is compared to the actual target using a loss function. This function measures the accuracy of the predictions by quantifying the error.
- **Backpropagation:** The calculated loss is propagated back through the network. During this stage, the weights and biases are updated to minimize the error in predictions, effectively "learning" from the mistakes.
- **Iteration:** This process is repeated with multiple batches of data. Through iterative adjustments of the model parameters, the network progressively improves its output accuracy.

By systematically going through these stages, a NN learns to model complex patterns and make predictions with increasing precision. A visual illustration of the NN training process is provided in the Figure 2.11 below. The figure was taken from [5].

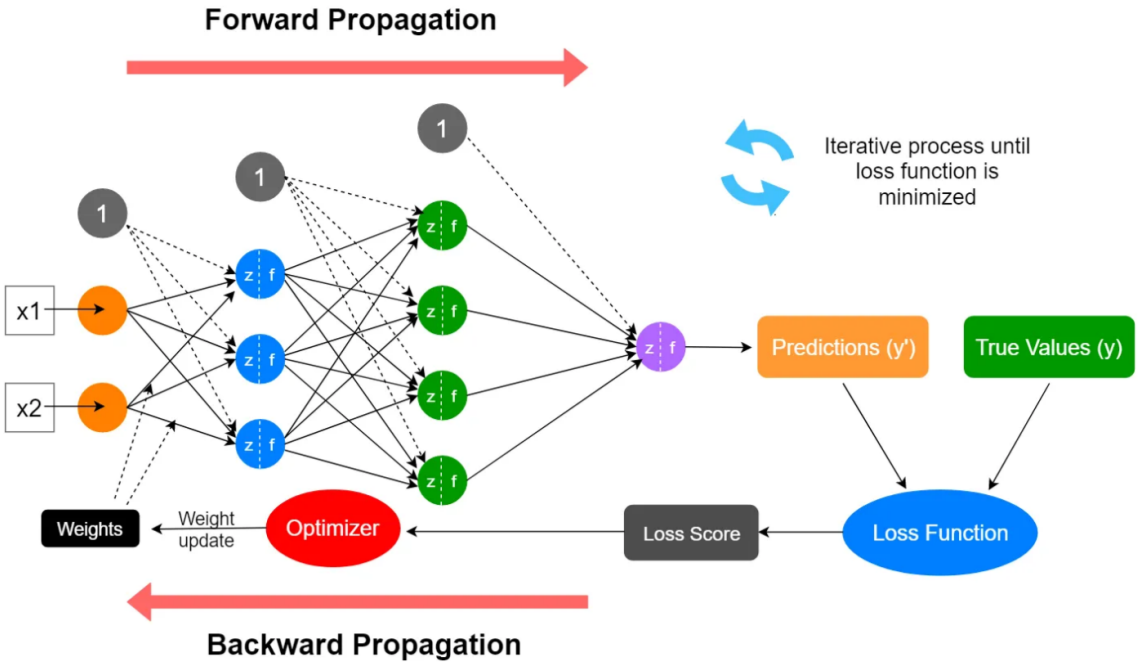


Figure 2.11: NN Training Process [5].

2.6 Probabilistic Learning and Calibration in Neural Networks

This section discusses probabilistic learning and the calibration of neural networks, focusing on methods to assess and verify model accuracy and confidence. This includes techniques for learning from probabilistic data, applying model calibration strategies, and utilizing reliability diagrams to ensure the model's performance aligns with expected outcomes.

2.6.1 Model's Accuracy and Confidence

In the realm of NNs, accuracy and confidence are two fundamental concepts that play crucial roles in evaluating and interpreting model performance.

Accuracy

Accuracy [62] is a measure of a model's performance that quantifies the percentage of correct predictions out of the total predictions made. In the context of NNs and classification tasks in particular, accuracy provides a metric to assess how well the model identifies the correct class for input data. It is calculated by comparing the prediction of each sample with their true label. High accuracy means that the model is effective at its task, making accurate predictions across a range of inputs. More specifically, accuracy is used in the training process of the NN and we can separate it into model accuracy (or simply accuracy) and validation accuracy.

Model accuracy measures the percentage of correct predictions made by the model over a specific dataset. It is critical for understanding how well the model has learned the patterns in the training data. However, high model accuracy can sometimes be bad, due to the potential for overfitting, where the model learns the training data too well, including its noise and outliers, making it less effective at predicting new, unseen data.

Validation accuracy is measured on a separate dataset not seen by the model during training, known as the validation set. This set is used to evaluate the model's performance and generalization capability. Validation accuracy is a more reliable measure of how well the model will perform on new, unseen data. It helps in tuning the model's hyperparameters and in making decisions about when to stop training to avoid overfitting.

Confidence

While accuracy provides a broad measure of model performance, confidence gets into the model's predictions on an individual level. Confidence [63], in the context of NNs, refers to the probability that a given prediction is correct. It is a measure of the model's certainty about its output, with higher confidence levels indicating greater certainty. In a classification task, confidence of each class is represented by the class prediction probability. In other words, confidence offers insight into the reliability of individual predictions, allowing users to know how much trust they can place in each output. Furthermore, confidence

can identify the model weaknesses, highlighting cases where the model is unsure of its decisions and may require further training.

2.6.2 Learning from Probabilistic Data

Multiclass Classification (Probabilistic Output Predictions)

In this thesis we deep in the field of supervised multiclass classification. This process involves the model making predictions across multiple classes, where each input (X) from a dataset (X) is associated with a label (Y) from a set of possible classes (Y = {1, 2, ..., K}). These classes represent the different categories into which the inputs can be classified. Unlike binary classification, where predictions are more straightforward, multiclass scenarios require the model to understand and predict based on the probabilistic distribution of data. This means that for each prediction, the model provides a set of probabilities, one for each class, indicating the likelihood of the input belonging to those classes. The model's prediction is typically the class with the highest probability.

Probabilistic Data (Randomness in Outputs)

Learning from probabilistic data outputs poses a different challenge in the field of ML. Probabilistic outputs refers to situations where the same input can lead to different outcomes with certain probabilities. Many real-word scenarios address this situation, especially when talking about stochastic processes and non-deterministic algorithms, that use randomness. More specifically, when dealing with probabilistic data in NN models, particularly in classification tasks, the conventional measure of accuracy can sometimes be misleading due to the inherent uncertainty in the data. At this point, it should be clarified that probabilistic input data, has nothing to do with probabilistic outputs (as they are in classification tasks) , these are two distinct concepts.

Impact on NN Accuracy

The traditional accuracy metric in NN models is calculated by comparing the predicted class (the class with the highest predicted probability) against the actual (true) class label for each input. However, in the context of probabilistic data, this method can unfairly penalize the model.

For instance, say a particular input has a 60% chance of being in class A, 20% chance of being in class B and 10% chance of being in class C. Now lets take 10 same input samples (and assume theoretically expected distribution of outputs) this means that the true output label is going to be 6 times class A, 2 times class B and 1 time class C. But the NN, will always return as predicted output the class with the highest probability, thus class A. This in terms of standard accuracy measures is incorrect, because by comparing the predicted output with the true labels is only going to end up in 60% accuracy, even though the prediction was reasonable given the probabilistic nature of the data and the NN was working correctly.

This scenario highlights a serious limitation of using accuracy as the only metric for evaluating model performance on probabilistic data. The model's prediction for the maximum

probability class may not always align with the single true label assigned to the sample, leading to seemingly lower accuracy. However, this does not necessarily mean the model is performing poorly. This leads us to try to prove the model's well-functioning with some extra metrics.

2.6.3 Model Calibration

Given these challenges, it's crucial not only to consider the model's accuracy but also to examine its confidence in predictions and, importantly, how well-calibrated these confidence levels are. NN model calibration [36] is the process of adjusting the output of a NN so that its predicted probabilities accurately reflect the true probabilities of those outcomes.

Getting into more details, our primary goal to achieve a well-calibrated model is to ensure that the confidence level $\hat{P}$ is calibrated accurately, meaning it should reflect the actual likelihood of the prediction $\hat{Y}$ being correct. For example, in an ideal scenario, if we make 100 predictions each with a confidence of 0.8, we would expect 80 of those predictions to be accurate. We define this notion as perfect calibration, expressed mathematically as $P(\hat{Y} = Y|\hat{P} = p) = p$ for all p in the range $[0, 1]$, assuring that the model's confidence matches the actual probability of correctness across all predictions.

However, achieving this level of perfect calibration is practically infeasible especially in a multiclass setting due to the continuous nature of the probability variable $\hat{P}$, making it impossible to calculate this probability directly from a finite number of samples. This limitation makes it necessary to use different empirical methods and metrics, such as reliability diagrams, so we can approximate and assess the model's calibration effectively.

2.6.4 Reliability Diagrams

Reliability diagrams [36] serve as a graphical method to assess the calibration of models particularly in classification tasks, visualizing how a model's predicted confidence aligns with its actual accuracy. In an ideally calibrated model, these diagrams would show a straight line along the diagonal, indicating a perfect match between confidence and accuracy. Points above the diagonal indicate underconfidence, thus the model's predictions are more accurate than the model's confidence suggests. The other way around, points below the diagonal suggest overconfidence, thus the model believes its predictions are more likely to be correct than they actually are.

A visual illustration of a reliability diagram with examples of an overconfident (blue) and an underconfident (orange) models is provided in the Figure 2.12 below. This showcases the comparison with the perfect calibration line (1:1). The figure was taken from [6].

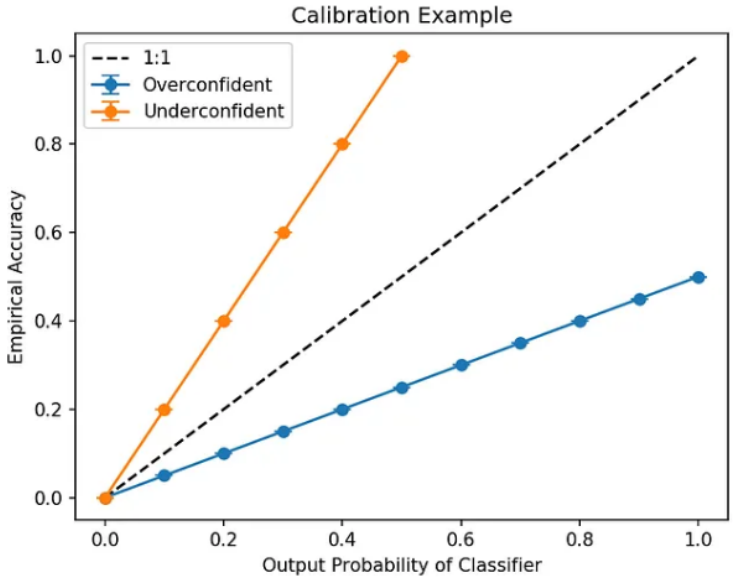


Figure 2.12: Example of Overconfident and Underconfident Models, comparing with Perfect Calibration (1:1) in Reliability Diagram [6].

To empirically evaluate model accuracy based on predicted confidence levels, we divide predictions into M equally sized intervals, or bins, based on their confidence score [36]. For each bin B_m, representing the range $\left(\frac{m-1}{M}, \frac{m}{M}\right]$, we calculate its actual accuracy. This is done by comparing each prediction within the bin to its actual outcome, with

$$\text{acc}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} 1(\hat{y}_i = y_i), \tag{2.18}$$

where $\hat{y}_i$ and y_i denote the predicted and true labels, respectively. This measure provides an unbiased and consistent estimation of the probability $P(\hat{Y} = Y | \hat{P} \in I_m)$.

Additionally, we define the average confidence for bin B_m as

$$\text{conf}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} \hat{p}_i, \tag{2.19}$$

where $\hat{p}_i$ represents the confidence level for each prediction. The accuracy and average confidence of a bin should ideally be equal for a model to be considered perfectly calibrated. However, it's important to note that reliability diagrams focuses on the relationship between confidence and accuracy and do not provide information on the distribution of samples across bins.

This is why in addition to reliability diagrams, another useful metric is the histogram of confidence. A histogram of confidence for a NN displays the distribution and the number of total samples of the model's prediction confidences across the entire dataset, providing a visual summary of how many samples each of the different confidence bins contains.

A visual illustration of a random example of a histogram of confidence diagram, where x axis is the confidence level and y axis is the number of samples, is provided in the Figure 2.13 below. The figure was taken from [7].

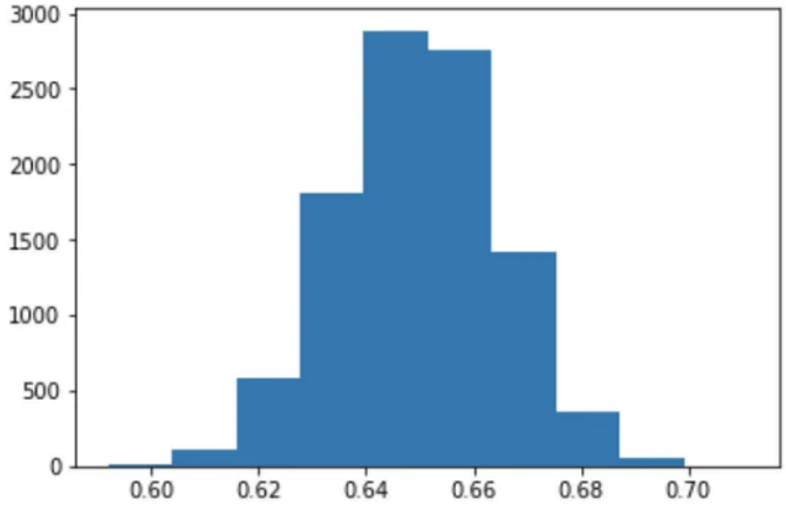


Figure 2.13: Random Example of Histogram of Confidence, X axis is the Confidence Level and Y axis is the Number of Samples [7].

2.7 Integrating MCTS and NNs for Enhanced Decision Making

The integration of deep learning techniques with search algorithms, into the domain of strategic and decision-making problems signifies a groundbreaking period in AI studies. Drawing inspiration from the remarkable success it found, firstly in Computer Go [28], and afterwards in Hex [35], that particularly combined DNNs with MCTS, we are exploring the further applicability and potential of DNNs and specifically we are focusing on supervised learning methods, to enhance move prediction and evaluation across a wide spectrum of applications characterized by complex decision trees and extensive branching factors.

It is now understandable that the great success of MCTS in solving different strategy problems mainly comes from its skill in carefully selecting the best options from a vastly expanded tree of choices. This ability is because its dynamic mechanism of updating node evaluations based on outcomes derived from simulated sequences of moves. So, a lot of the improvements in MCTS are about making it better at picking which options might lead to success by learning from these simulations. As a result, a lot of the improvements in MCTS are about making these choices smarter by integrating offline prior knowledge. This approach enhances the search bias towards more favorable outcomes based on historical data and advice from experts. In the next paragraphs, we analyze how it is possible to integrate MCTS with DNNs having the role of the expert, providing this prior knowledge to the search [28, 35, 64].

2.7.1 Data Collection and Offline Training

In the realm of training NN for advanced decision-making, the data that have been collected, play a pivotal role in the performance of the algorithm. There are two different approaches to collecting data for the NN in these cases, deriving data from a deterministic expert system or generating datasets through self-play simulations. We choose the second approach with self-play [35].

Data from a Deterministic Expert System

The first method involves collecting data from an expert system designed to solve state-action scenarios with high precision. This system utilizes a set of complicated rules to determine the optimal moves and strategies across various states. Despite its deterministic nature, ensuring consistent and high-quality data output, requires for the process, significant computational demands due to the depth and complexity of its rule-based analysis.

Self-Play Simulation Data

This method uses self-play simulations as a dynamic method for data generation. In this technique we create data from running simulations with decisions driven by the search algorithm that we are using, in this case MCTS. The adaptive nature of self-play fosters the generation of a rich and varied dataset, reflecting a wide array of game scenarios and outcomes. This approach lets us quickly gather a large amount of good enough quality data that capture the decisions and strategies used in the simulation across different states and conditions.

Offline Training of the NN

A critical aspect of this integration is that the NN is trained offline [35, 28]. Unlike online learning, where models are updated in real-time as new data from the simulation, becomes available, the NN is trained on a dataset of simulation scenarios and outcomes before its integration with MCTS. This training involves supervised learning methods, particularly multiclass classification, where the network learns to predict the outcome and the best moves from an input state, from the historical data. More specifically, the NN gets a state: s of the MCTS as an input and it produces a distribution of probabilities over actions given this state as an output: $P(s, a)$.

Once trained, the NN's model is saved and loaded to guide the MCTS in real-time scenarios. The separation of training and application phases ensures that the time computational overhead of deep learning does not significantly delay the speed of the MCTS during simulation. This approach cleverly divides tasks for each component, to operate within their strengths. Combining the former knowledge of the NN with the dynamic search exploration of the MCTS.

2.7.2 Enhancing MCTS with Neural Networks

The integration of the NN's prior knowledge happens through the selection phase of the MCTS algorithm. The selection that determines the tree policy is now modified to be NN-guided selection.

Neural Guidance in the Selection Phase

Getting into detail, we know that the selection phase is pivotal in MCTS, guiding the algorithm's exploration of the decision tree towards the most promising nodes. By traditional means, this phase uses the UCT formula to balance exploration and exploitation. However, NN-guided MCTS uses Predicted Upper Confidence Trees (PUCT) instead [35, 28]. This approach exploits the predictive power of the NN to guide the selection phase of MCTS, enriching the traditional search process with a layer of strategic foresight based on the probability distribution $P(s, a)$ provided by the NN. This probability distribution indicates the potential success of every possible action a from the state s, making an evaluation between more and less promising moves. The higher the probability of one action, the more promising it is. This way, by adding this prior knowledge into MCTS, the search process is transformed from a basic UCT search to a more sophisticated version, PUCT, enhanced with prior predictions.

Numerous variants of the PUCT formula exist and can be found in [28]; each with subtle distinctions regarding the additional score term that quantifies the influence of the NN. Initially, we employed the widely-used original variant, labeled as variant 0 in figure 2.14. Following brief experimentation with other variants, we determined that variant 4 from figure 2.14, yielded the most favorable outcomes for our autonomous-driving problem. Notably, variant 4 does not contain a squared root operation over the number of state/action visitations in the denominator. Therefore, state visitation affects more strongly the related term with respect to variant 0.

The mathematical formulation of PUCT (variant 4) modifies the standard UCT selection criterion by adding a term that represents the NN's predictions:

$$\text{score}(s, a) = Q(s, a) + C_b \cdot \sqrt{\frac{\ln N(s)}{N(s, a)}} + C_{pb} \cdot \frac{P(s, a)}{N(s, a) + 1} \tag{2.20}$$

where:

- $Q(s, a)$ stands for the average reward obtained after taking action a from state s, representing the exploitation component of the strategy.
- $N(s)$ is the total number of visits to the state s, and $N(s, a)$ is the number of times action a has been taken from state s, which are used to calculate the exploration factor.
- $P(s, a)$ represents the prior probability of selecting action a in state s, as estimated by the NN. This term introduces the concept of neural guidance into the selection process.

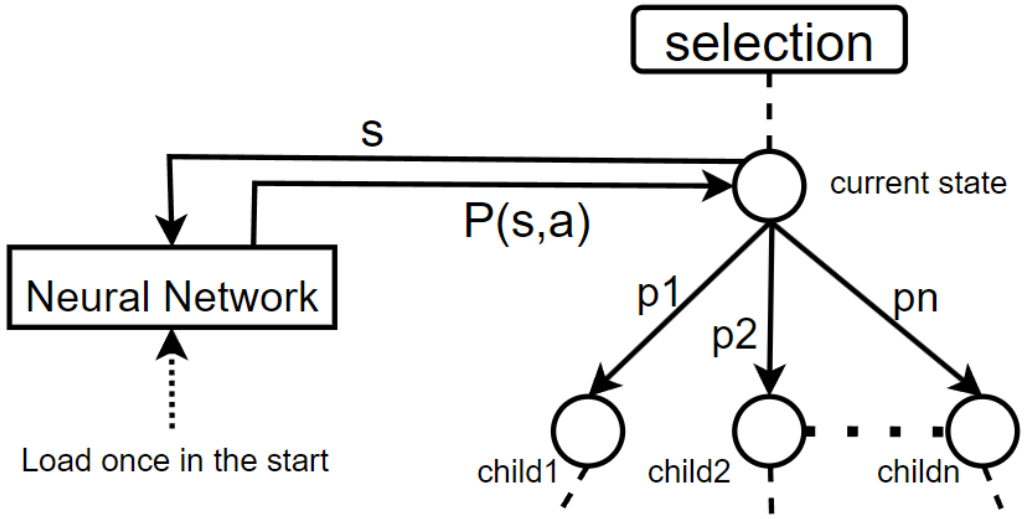
- C_b and C_{pb} are two constants that balance the relative importance of exploitation, exploration, and neural guidance.

A table of the different PUCT variants is shown in Figure 2.14 , taken from [28]. Each variant is associated with a different index number.

PUCT	0	$c\ P(s,a)\frac{\sqrt{N(s)}}{1+N(s,a)}$
	1	$c\ P(s,a)\sqrt{\frac{2\ lnN(s)}{N(s,a)}}$
	2	$c\ P(s,a)\frac{\sqrt{N(s)}}{N(s,a)}$
	3	$c\ P(s,a)^{\mu}\sqrt{\frac{N(s)}{1+N(s,a)}}$
	4	$c\ \frac{P(s,a)}{1+N(s,a)}$
	5	$c\ P(s,a)\frac{\sqrt{N(s)+1}}{N(s,a)+1}$
	6	$c\ P(s,a)\frac{N(s)}{1+N(s,a)}$
	7	$c\ P(s,a)\frac{\sqrt{N(s)+\epsilon}}{N(s,a)+1}$
	8	$c\ P(s,a)\sqrt{\frac{lnN(s)+1}{1+N(s,a)}}$
	9	$c\ P(s,a)\sqrt{\frac{N(s)}{1+N(s,a)}}$
	10	$c\ P(s,a)\sqrt{\frac{lnN(s)}{1+N(s,a)}}$

Figure 2.14: Variations of tree policies based on UCT. Different groups of selection formulae are divided by two horizontallines and each member of a group has a variant number.

A visual illustration of how the NN-Guided selection works is provided in the Figure 2.15 below. $Child_i$ with the highest PUCT value is then selected. Where, s is the state, $P(s,a)$ is the distribution over all n actions and pi is the prediction probability of $child_i$. The $child_i$ from the set of all n possible children with the highest PUCT value is selected.



$$\text{Select: } child_i = \underset{i\in\{1,2,..,n\}}{argmax}\{ PUCT_i \} = \underset{i\in\{1,2,..,n\}}{argmax}\{ UCT_i + f(p_i) \}$$

Figure 2.15: Representation of NN-Guided Selection, where s is the state, $P(s,a)$ is the distribution over all actions and pi is the prediction probability of $child_i$. The $child_i$ with the highest PUCT value is then selected.

2.7.3 Benefits of the Integrated Approach

The integration of $P(s,a)$ into the selection phase fundamentally shifts the dynamics of MCTS. It introduces a deep learning informed bias towards actions that the NN identifies as having higher potential, even before any simulations are run from those nodes. This pre-determined valuation uses the accumulated knowledge encoded in the NN and guides the search accordingly, gaining this way many advantages over the plain MCTS algorithm.

- **Enhanced Exploration Efficiency:** NN-Guided MCTS guides the exploration of the decision space by leveraging NN predictions to direct the search to take actions towards more promising branches of the tree. Unlike plain MCTS, which relies on random simulations, potentially leading to inefficiencies and some times in sub-optimal solutions, the NN's prior knowledge ensure a more focused and efficient exploration process by prioritizing moves with a higher likelihood of success.
- **Accelerated Convergence to Optimal Strategies:** Furthermore, this integration accelerates the procedure of the identification of strong moves and strategies, thereby speeding up the policy improvement process. While plain MCTS gradually refines its policy through trial and error, NN-Guided MCTS is able to bypass less promising strategies, achieving faster convergence to optimal solutions, especially in complex scenarios with extensive branching factors. In other words, given that the two algorithms return the same solution, NN-Guided MCTS is going to need less computational time to reach it.
- **Improved Decision Quality under Computational Constraints:** Finaly, NN-Guided MCTS enhances decision-making by enabling high-quality decisions with

fewer simulations. This approach differs from plain MCTS, which requires extensive exploration to achieve similar quality to the solutions, making NN-Guided MCTS particularly more efficient in scenarios with limited computational resources or where rapid decisions are essential. In other words, given the same number of iterations for the two algorithms, the NN-Guided MCTS is going to delve deeper into the decision tree, thus it is going to find a better more informed solution.

2.7.4 Challenges and Considerations

While the integration of NN guidance in MCTS offers significant advantages, it does not come without its challenges and considerations. The most significant concern in implementing this advanced approach, is about the computational demands of NNs, especially when compared to the relatively lightweight computational requirements of traditional plain MCTS.

The main challenge associated with NN-guided MCTS, is the speed of NN predictions and what are the potential solutions to minimize this issue [35]. More specifically, NN particularly those with deep architectures necessary for capturing a complex environment, are naturally very slower than the straightforward algorithms that drive plain MCTS. The time taken by NNs to make a single prediction from the current state, despite being in the range of milliseconds, can accumulate over hundreds of iterations and multiple states in each iteration, significantly slowing down the MCTS process. This delay is problematic, especially in scenarios where real-time decision-making is crucial or when computational resources are limited.

Chapter 3

Related Work

This chapter reviews existing literature in autonomous driving, focusing on both lane-based and lane-free traffic systems. It outlines the general advancements in the field, then provides a detailed analysis of specific studies relevant to our research. The review extends to exploring Monte Carlo Tree Search (MCTS) applications in various domains, with a particular emphasis on motion planning techniques in autonomous driving. Additionally, the chapter details the foundational work and prior studies that have directly influenced the development of our thesis, offering a comprehensive background that sets the stage for our contributions.

3.1 Autonomous Driving

This section delves into the realm of autonomous driving, providing a thorough overview of existing research related to both traditional lane-based and innovative lane-free traffic systems.

3.1.1 Autonomous Driving in Lane Based Traffic

One of the core challenges in autonomous driving within lane-based traffic is motion planning, due to the computational difficulty in finding exact solutions and navigating vehicles through structured urban paths. The literature suggests various approaches to tackle this issue.

Motion-planning techniques can be categorized into three broad types of methodologies, namely: Variational methods, Graph-Search methods, and Incremental search methods [65]. These different strategies aim to navigate vehicles effectively through complex traffic scenarios, to address the difficulties of autonomous vehicle planning in a structured traffic environment.

- **Variational Methods:** approach motion planning as an optimization problem, representing it through a high-dimensional function that represents various vehicle dynamics, such as speed and distance from other vehicles. These methods apply non-linear optimization to find solutions, with some employing numerical techniques

like Euler's method [66] or using polynomials for trajectory approximation [67]. However, they often stack to sub-optimal solutions in local optima.

- **Graph-Search Methods:** visualize the vehicle's path and interactions within its environment as a graph, helping like this the search for minimum cost paths. There are various techniques to construct these graphs, integrating heuristic, geometric, and sampling methods to emulate the complexity of real-world traffic networks. Despite being useful, these methods are constrained because they depend on using a predefined list of movement patterns, limiting the exploration of potential paths.
- **Incremental Search Methods:** dynamically generate a reachability graph or tree by sampling the configuration space, expanding until it covers an area based on a specific goal. This approach is particularly noted for its adaptability and the use of randomized techniques to navigate the vast configuration space. Among the most accurate algorithms in this category are the expansive spaces tree (EST) planning algorithm [68] and the Rapidly-exploring Random Trees (RRT) [69], both of which stand out in identifying possible path trajectories in high-dimensional systems.

Notably, a very interesting approach is the one that uses Model Predictive Control (MPC) [70], proposing a driving environment uncertainty-aware motion planning framework. MPC's integrate various aspects such as risk assessment for vehicle stability, position accuracy, obstacle avoidance, and adherence to vehicle dynamics constraints. This method employs a 4-DOF (Degrees of Freedom) vehicle dynamics model to evaluate the risk of a vehicle rolling over. It also improves the method for estimating uncertainties, making it more accurate. This is based on the Extended Kalman Filter (EKF), enhancing like this the autonomous driving system's safety and reliability in complex traffic scenarios.

3.1.2 Autonomous Driving in Lane Free Traffic

Lane-free autonomous driving represents a significant paradigm shift from traditional, lane-based systems, promising to optimize traffic flow and enhance safety. Historically, lanes simplified driving tasks by reducing the driver's need to monitor all surrounding vehicles (front,back,left,right), effectively optimizing movement and improving safety. However, the complexity of lane-changing maneuvers is responsible for a notable percentage of traffic accidents, showcasing its limitations of this system.

Lane-free driving, e.g. as envisaged by like the TrafficFluid Concept [11] which suggests a lane-free environment with "nudging" effect among cars, aims to address these challenges by employing sophisticated control methods and optimization strategies, allowing vehicles to navigate without the constraints of predefined lanes. However, the transition to a lane-free environment introduces new complexities in motion planning and vehicle control, necessitating innovative algorithms and models to ensure fluid vehicle movements and safety.

Over the last few years, several studies on lane-free traffic environments have explored various control methods to optimize agent policies. More specifically, strategies based in Control Theory were suggested in [11, 13], while others have employed Multi-Agent Decision Making approaches [14, 15]. For instance, [11] proposed a strategy for maneuvering vehicles in a lane-free environment based on heuristic rules that simulate "forces" to enable

overtaking and adapt to different scenarios. Furthermore the work of article [13] focused on developing a two-dimensional cruise control system using principles of Control Theory for lane-free traffic. Moreover, [15] introduced an optimal control strategy for lane-free vehicles, particularly emphasizing model predictive control, where each vehicle plans its path considering the future movements or else trajectories of nearby vehicles. On the other hand, [14] addressed the challenge by leveraging the max-plus algorithm to create a dynamic collaborative communication network among vehicles, threw a graph structure.

Additionally, some studies have turned to Deep Reinforcement Learning for policy optimization, necessitating the environment's description through a Markov Decision Process (MDP). The work of Karalakou has developed an MDP for a lane-free, ring-road environment [8] and evaluated it using the Deep Deterministic Policy Gradient (DDPG) algorithm. In this research, this previously established MDP framework will serve as the foundation of the representation of Monte Carlo Tree Search (MCTS) as an MDP in autonomous lane-free driving environment. Concluding, the study in [16] introduced an algorithm that utilizes implicit imitation Deep-RL, drawing on the experiences of mentor agents and the defined MDP to expedite training through the use of state transition data from expert mentors.

3.2 MCTS-Based Planning Techniques

Beyond its landmark contributions in strategic game decisions and planning in uncertainty that were detailed before [26, 27, 28] in Section 2.2, MCTS has been used in a variety of significant advancements across different AI domains. These achievements underscore MCTS's versatility and its transformative impact on complex problem-solving. Here are a few notable diploma thesis, in different domains where MCTS has made its mark, setting the stage for further exploration and innovation.

To begin with, [71], explores a Bayesian approach to generate personalized recommendations by learning from user preferences through non-intrusive feedback. Utilizing a Markov Chain Monte Carlo algorithm, it adeptly models uncertainty and adjusts to sparse data by observing behavior patterns. The method, applied to online hotel bookings, yields promising, personalized results.

Another study [72] applies MCTS to the game of "Diplomacy", evaluating eight MCTS agent variants using the Upper Confidence Trees (UCT) for optimal exploration and exploitation. Enhanced with a domain-specific heuristic, these agents were tested against top competitors, showing that MCTS can outperform the leading Diplomacy agent, DBrane, in multi-agent settings.

Continuing with in the realm of strategic games, this work [73] introduces a novel application of MCTS to "Settlers of Catan", employing reinforcement learning and bandit methods to balance decision-making. The agent, uniquely considers the game's complete rules and enables player negotiations, outperforms traditional methods, particularly when integrating strategies based on human behavior for initial placements.

Finally, this paper [74] explores enhancing online planning for partially observable Markov decision processes (POMDPs) with the POMCP algorithm, by integrating state-variable constraints through hard constraint networks and probabilistic Markov random fields. The case study, based on Rocksample, demonstrates significant performance boosts, with improvements up to 50% in average discounted return, highlighting the value of incorporating prior knowledge into the planning process.

3.2.1 MCTS and Autonomous Lane-Based Driving

The MCTS algorithms, known for their incremental search methods, have various applications, particularly within autonomous driving scenarios. One notable application is simulating the behavior of other vehicles on the road using a prediction model, taking into account human intentions [29]. Another study implemented a learning-based continuous MCTS approach using KB-Trees [30]. In addition to that, there is also an event-based approach , that uses probabilistic models during the MCTS's selection phase to predict the actions of other drivers [31]. Beyond modeling the behavior of other vehicles, MCTS has been used for different purposes in autonomous driving, such as predicting maneuvers based on image inputs [32].

3.2.2 MCTS and Autonomous Lane-Free Driving

Finally and most importantly, the use of MCTS for guiding autonomous vehicles in a lane-free setting was first introduced by Giankoulidis [34], a student from the Technical University of Crete and member of the TrafficFluid research team. More specifically, it develops an MCTS-MDP framework specifically tailored for autonomous lane-free driving, where the algorithms operate within it. Then, the study explores the effectiveness of plain MCTS for individual vehicle decision-making, focusing on objectives like collision avoidance and maintaining desired speeds. Additionally, it extends the model to a multi-agent setting by incorporating a Factor Value MCTS that utilizes Coordination Graphs, allowing for dynamic interaction and communication among vehicles.

This thesis builds on Giankoulidis's initial work, aiming to refine and enhance his approach. The first step involves improving and optimizing the existing plain MCTS method, inside the established MCTS-MDP framework. After that, we further develop the model by integrating MCTS with Neural Network (NN) knowledge and guidance. We compare our results directly with Giankoulidis's MCTS algorithm, clearly highlighting our improvements and showcasing the advancements in an understandable and detailed way.

Chapter 4

Our Approach

In this chapter, the discussion transitions to our specific work and methodologies. It explains the development and implementation of an MCTS-MDP framework tailored for lane-free autonomous driving [34], discussing various strategies such as an enhanced version of the plain MCTS that was developed in [34] and our own Neural Network-Guided MCTS. The chapter elaborates on the distinct phases of our approach, from data collection and training to integration and practical testing, underscoring the novel applications designed to navigate the complexities of lane-free environments.

4.1 MCTS for Autonomous Lane-Free Driving

In this section we navigate through the adaptation of Monte Carlo Tree Search (MCTS) algorithm within the lane-free traffic environment. We provide a detailed analysis of how MCTS is incorporated into real-world traffic simulations and offer a step-by-step breakdown of the MCTS represented as a MDP. More specifically, we delve into the state and action spaces, examine the reward function, and explain the roles of nodes and edges within the search tree. Finally, we understand how MCTS interfaces with the complex and dynamic variables of lane-free traffic management. This analysis and approach is also a part of Pantelis Ginakoulidis previous work in [34].

4.1.1 Lane-Free Traffic Environment

In our study, we model an open highway environment to simulate traffic scenarios, wherein multiple automated vehicles are driving together in the simulation. Vehicles function based on our agent that uses a decision-making algorithm, in our case MCTS or NN-Guided MCTS, that searches for the best actions to take inside this MCTS-MDP formulation [34]. More specifically, in order to attain the best outcomes with respect to traffic efficiency, it has been decided that each vehicle will independently execute its own MCTS algorithm respectively as the ego vehicle. Our agent's goal is to reach its unique desired speed and avoid collisions by complicated maneuvers and nudging among other vehicles.

Our agent has the ability to monitor its own position as well as that of the nearby vehicles' positions (x, y) and speeds (v_x, v_y), observed as two-dimensional vectors that reflect lon-

gitudinal (x-axis) and lateral (y-axis) components. All vehicles within this environment are standardized in terms of size and movement dynamics, and each one independently sets, randomly at the start of each simulation, a target speed or else desired speed v_d from a predefined range $[v_{d,\min}, v_{d,\max}]$, a parameter that is also observable to our agent. The control inputs for our agent are the longitudinal and lateral accelerations (a_x, a_y), which directly influence acceleration/deceleration (gas/break) and steering (left/right) directions, respectively.

For the simulation of this lane-free traffic scenario, we leverage an extension of the Simulation of Urban MObility (SUMO) tailored for lane-free traffic [75]. The open highway framework, is represented in Figure 4.1 below and as we can see it is a straight road with vehicles entering from the left side (start of simulation) and exiting from the right side (termination of simulation).

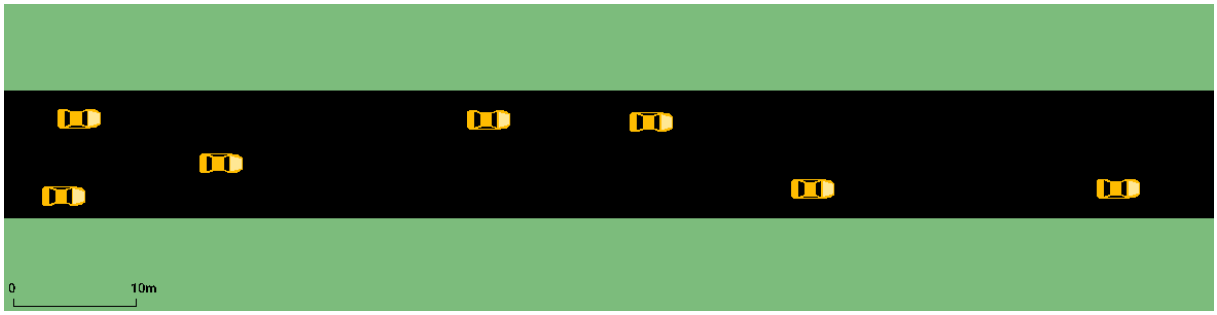


Figure 4.1: SUMO Interface for Simulations.

4.1.2 State Space

To effectively implement the MCTS algorithm for our specific problem, which involves leveraging the domain's Markov Decision Process (MDP) structure to optimize the policy within its constraints, it is essential to define some fundamental principles. The first thing that must be defined is the state space $\mathcal{S}$. To do that, we must define the attributes characterizing a vehicle c.

A vehicle c is characterized by a collection of attributes: $c = \{p_x, p_y, v_x, v_y, l, w, d_v\}$, where each attribute is defined as follows in Table 4.1 below:

Parameter	Definition
p_x (m)	Position in x-axis
p_y (m)	Position in y-axis
v_x (m/s)	Speed in x-axis
v_y (m/s)	Speed in y-axis
l (m)	Vehicle length
w (m)	Vehicle width
d_v (m/s)[1]	Desired speed

Table 4.1: A vehicle's definition.

At any given moment t, represented from a time-step in a SUMO simulation, the state of a particular vehicle c encapsulates all its current moment characteristics, as position and speed in both x and y axis, and also its length, width and desired speed, that they stay unchanged (for that particular vehicle) for any time step. In addition to that, a state also holds the same information about other surrounding vehicles (around the targeted ego vehicle) within a longitudinal distance from it, that is within the predefined visibility distance d.

At this point, with all these parameters defined, we can formally describe a state s as a set $s = \{c_m, \Gamma = [c_1 \dots c_n]\}$, where c_m is the ego vehicle, and Γ is the set of neighboring vehicles c_m. It is very important to emphasize that in our approach the ego vehicle, is capable of identifying other neighboring vehicles not only in front of it (where $d_x = x_m - x_i > 0$), but also in the back of it (where $d_x = x_m - x_i < 0$). This goes in contrast with Giankoulidis's approach of MCTS [34], that a state only included the neighboring vehicles in front of the ego and not at all the ones in the back. Finally, a terminal state is reached when the ego vehicle either collides with another vehicle or if it exceeds the road boundaries.

A visual illustration of the state space is provided in the Figure 4.2 below. The figure was taken from [8].

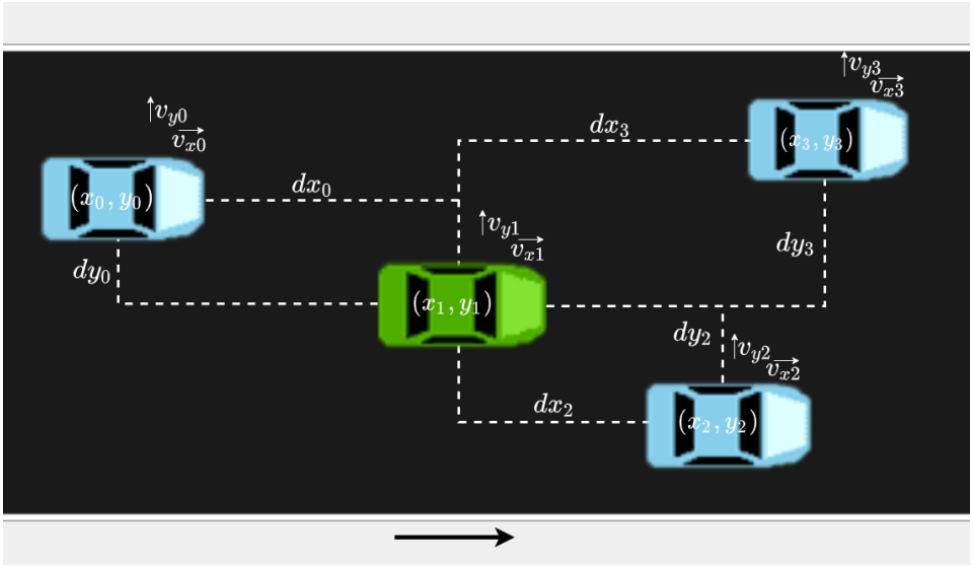


Figure 4.2: State Space Representation as it was implemented in [8].

4.1.3 Action Space

The action space of a vehicle is defined by the range of longitudinal and lateral accelerations it can execute at any given moment, t. Theoretically, a vehicle c can adopt any action $\alpha = \{a_x, a_y\}$, where a_x represents the longitudinal acceleration and a_y the lateral acceleration. These accelerations can vary within a spectrum from $-a_l$ $(\mathrm{m/s^2})$ to $+a_h$

[1] A number between a predefined range $[v_{d,\min}, v_{d,\max}]$ that is set for each vehicle in the beginning of the simulation randomly. It represents the optimal (maximum) speed that each vehicle tries to reach and maintain.

(m/s^2), with a_l and a_h being real numbers that reflect the vehicle's capabilities and characteristics. The values of these accelerations include the entire range of real numbers, where a positive number means acceleration and a negative deceleration.

For the purposes of constructing a search tree within our algorithm, the action space $\mathcal{A}$ must be discrete, encapsulating a finite set of possible accelerations: five distinct possibilities for longitudinal acceleration and three for lateral acceleration. An action, α, is thus defined as a pair of selected accelerations, $\alpha = \{\alpha_x, \alpha_y\}$, where α_x is chosen from the set of longitudinal accelerations and α_y from lateral accelerations, with each able to take value only from the predefined sets as described below:

- Longitudinal Acceleration Values (m/s^2): $\alpha_x = \{-5, -2, 0, 2, 5\}$
- Lateral Acceleration Values (m/s^2): $\alpha_y = \{-1, 0, 1\}$

With this discretization, the action space $\mathcal{A}$ contains all possible combinations of the longitudinal and lateral accelerations, resulting in a total of 15 distinct actions available to the vehicle. This approach simplifies the decision-making process for the vehicle, enabling a manageable yet effective set of maneuvers to choose from.

A visual illustration of the action space with the 15 discrete actions, that are generated from the combinations of lateral and longitudinal accelerations, is provided in the Figure 4.3 below.

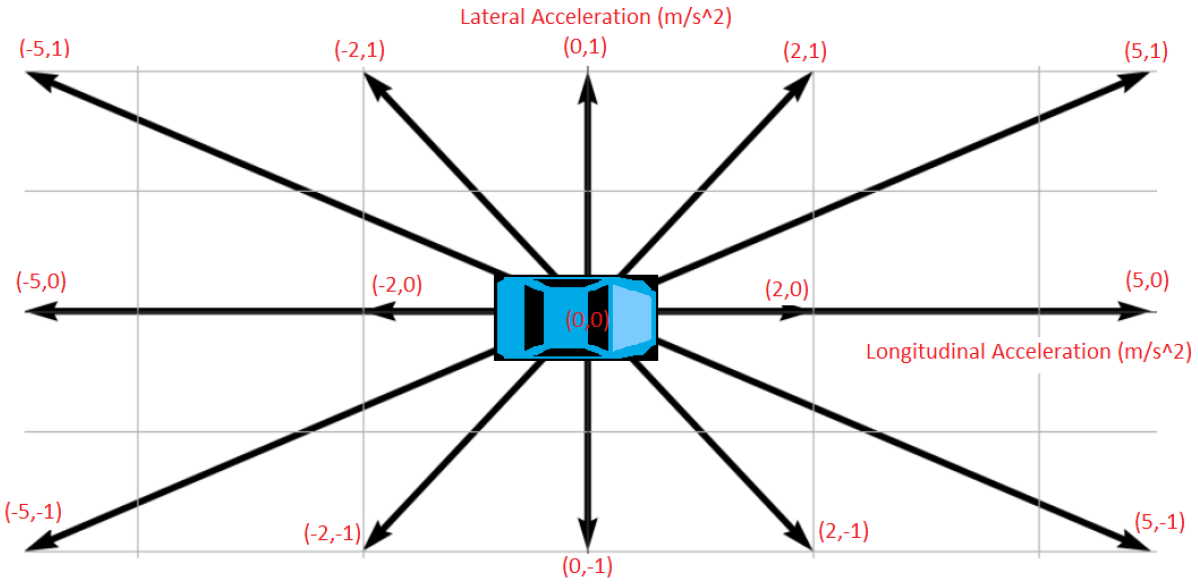


Figure 4.3: Action Space Representation, 15 Possible Discrete Actions (ax,ay).

4.1.4 Reward Function

The reward function plays a pivotal role in guiding the algorithm towards finding the optimal policy, because it is responsible for quantifying the outcomes of different actions under different states. The two main goals of our agent is the avoidance of collisions with other vehicles and the maintenance of a speed that is as close as possible to its desired speed. In our approach, one simulation or playout of the MCTS simulation phase, is

stopped either if a terminal state is reached either after n forward steps, where n is a predefined integer value.

Collision Avoidance Objective

For the purpose of collision avoidance, to begin with, our heuristic approach incorporates artificial potential fields, as suggested in [11]. In addition to this, we also consider any actual collisions that occur during the simulation phase by resulting in a negative impact on the score. More specifically, when a collision occurs involving the ego vehicle, the simulation is prematurely terminated before reaching our maximum of n steps or timestamps. The deeper into the simulation a collision occurs, the lower the penalty (negative score) applied to the current state's score is. This is based on the rationale that later collisions in the simulation indicate a lesser likelihood of their occurrence.

It is also essential to consider that some actions might lead to states with a higher probability of collision, even if actual collisions are infrequent or not happening at all from that state in the simulation. To address this, we integrate a negative value associated with such actions to achieve collision avoidance. The artificial potential fields provide a quantifiable measure of the collision risk between two vehicles by assessing their current positions and speeds. This risk is calculated by the function f_{ij}, representing the potential collision between vehicles i and j.

In our approach, we calculate a total cumulative "potential collision" score for the simulation. This score adds up the potential collisions of all vehicle pairs that include the ego vehicle in the terminal state of the simulation, but it does this only if the vehicles are within the predefined distance d.

The collision avoidance part of the reward function is given by:

$$g(s) = \begin{cases} \frac{D}{n_s} + \sum_{v=1}^{M} f_{i,v}, & \text{if } \text{dist}(i,v) < d \text{ and } n_s < n, \\ \sum_{v=1}^{M} f_{i,v}, & \text{if } \text{dist}(i,v) < d \text{ and } n_s \geq n, \\ 0, & \text{otherwise,} \end{cases} \tag{4.1}$$

where:

- D: is a positive constant.
- n_s: is the number of steps simulated before a collision occurs. In instances where a collision does take place, the fraction $\frac{D}{n_s}$ represents the collision's significance based on the time-step the collision occurred. In the rest of the cases where a collision did not take place, this factor is removed.
- n: is the maximum number of total simulation steps in one MCTS simulation cycle (if a collision occurs before step n, then the simulation ends prematurely).
- $\text{dist}(i, v)$: is the longitudinal distance between the two vehicles i and v.
- M: The number of vehicles within the distance less than $MaxD$, thus $\text{dist}(i, v) < MaxD$, for any two vehicles i, v and a predetermined constant $MaxD$.

- $f_{i,v}$: quantifies the risk of a potential collision between vehicles i and v, according to artificial potential fields.

Desired Speed Maintenance Objective

Regarding the second objective, the outcome of a vehicle's action within the simulation is deterministic. In this scenario, when a vehicle is at a particular state and an action is taken, the resulting speed of the next state is determined by the kinematic equation $v_{t+1} = v_t + a_t T$. Similarly, the vehicle's updated position is also deterministic and it is calculated by $x_{t+1} = x_t + v_t T + \frac{1}{2} a_t T^2$, where x_t represents the position at time t, v_t denotes the speed, a_t signifies the applied acceleration at that time, and T is the constant time period.

Given the predictability of an action's result, the evaluation of a state's score remains unaffected by the simulation outcomes. Instead, we are focusing on the comparison between the vehicle's actual longitudinal speed and its targeted desired speed. Our goal is to minimize the variance between the vehicle's actual speed $v_t(s)$, and the vehicle's desired speed, v_d.

For the desired speed maintenance objective, the heuristic is defined as:

$$f(s) = \frac{\epsilon}{|v_t(s) - v_d| + \epsilon}, \tag{4.2}$$

where:

- $v_t(s)$: is the actual speed of the vehicle at state s.
- v_d: is the desired vehicle speed.
- ϵ: is a small positive constant to ensure that the function is well-defined even when the actual speed equals the desired speed.

Objectives Integration

The total reward for a simulation, taking into account both collision avoidance and speed maintenance, is calculated as a weighted sum:

$$G(S) = \alpha \cdot g(s) + \beta \cdot f(s), \tag{4.3}$$

where:

- α: A coefficient weighing the importance of collision avoidance, where a larger value places a higher priority on preventing collisions.
- β: A coefficient weighing the importance of maintaining the desired speed, where a larger value emphasizes adherence to the desired speed.

Both α and β are non-negative and are heuristic parameters that can be tuned during the simulation to balance the objectives. The attention to these factors is vital in the formulation of the MCTS algorithm's strategy, ensuring that the vehicle not only operates safely, avoiding collisions, but also works efficiently to its desired speed.

4.2 Plain MCTS in Lane-Free Traffic

In this section, we dive into our specific structure of the MCTS search tree and the way it integrates within the lane-free traffic environment. We focus on the structure of the tree and explain the meaning of each component within the context of our autonomous driving domain.

4.2.1 Navigation in Autonomous Driving Decision Space

In the previous sections, we defined the components essential to represent the MCTS, including the state and action spaces, along with the reward function. However, we must also analyze the architecture of the search tree itself, the way it is created, and the way we navigate through the problem space. To construct the search tree, it's important to explain first what makes up the nodes and the edges, specifically now for our autonomous driving environment.

In our approach, a node is characterized by some key elements:

- a specific state s (as it was analyzed in Section 4.1.2), which is basically the main characteristic of the node, including some other ones that help us to create and navigate the tree.
- the action a that transitions to the state s, thus this allows to recognize the node's parent node and helps us to navigate the search tree.
- the potential actions available from the state s, which in our scenario includes the entire action space, all 15 possible moves for any given child node. This helps us to create and expand the tree.

An action represents an edge of the tree and in context of driving, it means an application of accelerations (lateral and longitudinal), to the vehicle involved (as it was analyzed in Section 4.1.3). When a new action is taken from a node (state), it means a transition from it to a new child node (depending on what was the action). This allows the vehicle to take any subsequent actions from the current state and growing like this the tree by creating new nodes.

Additionally, we integrate the reward function, $G(s)$, as defined by equation 3.3, to calculate the score of a single simulation. This score represents the value or else the reward of a node and it is used in the selection phase in the Upper Confidence Trees (UCT) formula (equation 2.5). More specifically, it is used to calculate the exploitation factor $Q(s, a)$, by measuring the average reward received, after taking action a from state s.

4.2.2 Algorithm Implementation

Presented below in Algorithm 1 is a high-level implementation of the plain MCTS algorithm. The process begins by initializing the root node of the tree, which stands for the

current state of the ego vehicle at each time step in the SUMO simulation. It is important to understand that we run a complete MCTS cycle for every single time step in the simulation. Additionally, we have the flexibility to decide how many of the vehicles in the simulation will use MCTS algorithm for their decision-making or if they will just follow the standard navigation patterns and behavior as outlined in [11]. In our case, with the goal of improving the overall vehicle flow and traffic, so we can achieve the best results, we have chosen to make each vehicle independently run its own MCTS.

To continue with the algorithm's functionality, MCTS iterates over 4 key phases, selection, expansion, simulation, backpropagation which are described in Section 2.4 and visualized in Figure 2.3 We dive deeper into each of these phases in the following sections. The loop of these phases continues, creating like this the search tree, until it reaches the predefined limit of maximum iterations and it is terminated. When it comes to the last iteration, the algorithm picks greedily the final action based purely on which one offers the highest reward, without factoring in any exploration factors. After this, the algorithm identifies the best action for the vehicle's current state in the simulation. This action is then executed, and the corresponding changes in speed are applied to the vehicle, through its lateral and longitudinal accelerations.

Algorithm 1: A High Level MCTS implementation

```
iterations ← 0
root ← initialize_tree ()
while iterations ≤ MAX ITERATIONS do
    selected_state ← select (root)
    if is_terminal (selected_state) then
        continue
    end
    if number_of_visits (selected_state) ≥ MIN VISITS then
        expanded_state ← expand (selected_state)
    end
    else
        expanded_state ← selected_state
    end
    score ← simulate (expanded_state)
    backpropagate (expanded_state, score)
    iterations ← iterations + 1
end
```

4.2.3 Selection Policy

Presented below, Function 1 embodies the first step in the MCTS cycle, where the goal is to traverse the tree from the root to a leaf node by selecting optimal child nodes at each step. This selection is guided by the UCT score, a balance of exploration and exploitation, as it was analyzed in Section 2.4. More specifically, for each child of the current state, a UCT score is calculated (referenced in Equation 2.5), targeting to select the child with the highest potential based on average score from past performance from previous simulations

and unexplored opportunities. The chosen best child is the one with the highest UCT score, indicating it as the most promising node for further exploration or expansion.

Algorithm 1: Function 1: Selection Phase of MCTS

Input: *state*
children ← get_children (*state*)
for *child* ∈ *children* **do**
  score ← UCT (*state*, *child*) (2.5)
end
best_child ← $argmax_{child}$ (*score*)
return *best_child*

4.2.4 Expansion Specifics

The expansion phase represented below Function 2 is pivotal for advancing the search tree. When a leaf node is reached, one new child node at each time will be added to the tree, representing like this the possible future actions from that state. This is accomplished by generating a new action, executing this action to transition to a new state, and then adding this new state as a child of the current node. We have to note that the expansion process happens in a symmetric way each time. Meaning that it is designed to sequentially explore actions, a way that if a node is expanded with action a_i, the subsequent expansion will explore action a_{i+1}, and so on. Such a methodical progression ensures that the expansion is both orderly and exhaustive, effectively canvassing the spectrum of potential actions.

Furthermore, we have to clarify that a state is only expanded if it has acquired a minimum number of visits. This happens to ensure that enough information has been collected about a node, allowing for a more informed decision before committing resources to further exploration. This strategy helps prevent the premature expansion of paths that may be sub-optimal, in case the node does not show enough potential.

Algorithm 1: Function 2: Expansion Phase of MCTS

Input: *state*
action ← generate_next_action ()
execute_action (*action*)
child_state ← create_new_state (*state*, *action*)
state ← add_child (*child_state*)
return *child_state*

4.2.5 Simulation Policy

The Simulation phase represented below Function 3, also known as the playout or rollout, is where the algorithm simulates a path from the newly expanded node to a terminal condition or until it reaches a predefined maximum state depth M. Basically, our simulation advances M time steps ahead (or fewer if it comes to a terminal state) in the SUMO simulation and then it is terminated. Actions executed at each state during this

phase are randomly sampled from a uniform distribution across the available action space. This simple way lets us estimate the potential value of the expanded node without the computational cost of deeper, strategic analysis.

Once an action is executed, the algorithm simulates a new state for the ego vehicle, which also includes other neighbor vehicles that have also moved. In this simulation, we cannot predict the exact movements of other vehicles, because we have no knowledge of their movement strategy. Thus, an estimation must be made for what are the new neighbor vehicles positions and what are their new speeds. The simple assumption that we make is that no accelerations in either direction were applied to other vehicles than the ego. Therefore, their speed remains unchanged (lateral and longitudinal speeds are the same as before) and their positions are updated using the kinematic equation $x_{i+1} = x_i + v_i \times T$, applied across all intermediary states.

The simulation's outcome contributes valuable information, with scores calculated based on specific metrics such as collision avoidance and maintenance of the desired speed that are indicative of the simulation's success or failure, as described in Section 4.1.4 and equations 4.1, 4.2 and 4.3.

Algorithm 1: Function 3: Simulation Phase of MCTS

```
Input: state
while is_NOT_terminal (state) do
    if state_depth ≥ MAX ROLLOUT DEPTH then
        break
    end
    action ← generate_random_action ()
    execute_action (action)
    playout ← generate_playout (action, state)
    state_depth ← state_depth + 1
end
collisions_score ← g (state, state_depth, neighbors, playout) (4.1)
desired_speed_score ← f (state, desired_speed, playout) (4.2)
score ← G (collisions_score, desired_speed_score) (4.3)
return score
```

4.2.6 Backpropagation Specifics

After the simulation phase returns a score representing the potential of the chosen path, the Backpropagation phase represented in Function 4, updates the scores of the nodes along the path from the expanded node back to the root. This process ensures that the tree's statistics reflect the latest simulation results. Each node visited during the backpropagation updates its score with the new data, ensuring that the tree's overall knowledge grows more accurate with each iteration, making like this more informed future decisions.

Algorithm 1: Function 4: Backpropagation Phase of MCTS

```
Input: state, score
state ← update_score (score)
current ← get_parent (state)
while current ≠ null do
    current ← update_score (score)
    current ← get_parent (current)
end
```

4.3 Neural Network Guided Monte Carlo Tree Search

In this section, we thoroughly discuss the integration of Neural Network (NN) guided MCTS in lane-free traffic and analyze the challenges we faced and how we tackled these issues. More specifically, the NN is integrated into the selection phase of MCTS. Traditionally, this phase uses the UCT formula to balance exploration and exploitation. However, by employing the Predicted Upper Confidence Trees (PUCT) approach, we utilize the NN's predictive power to guide the selection process. The NN provides a probability distribution $P(s, a)$, indicating the potential success of each action a from state s. This probability helps evaluate and prioritize more promising moves, transforming the basic UCT search into a more sophisticated PUCT, enriched with prior predictions. This integration enhances the decision-making process by strategically focusing on actions predicted to be most successful, thereby improving the overall efficiency and effectiveness of the MCTS algorithm.

4.3.1 Data Collection

To begin with, we need to collect data in order to feed the constructed NN for learning purposes. Specifically, we intend to construct our dataset through self-play simulations, utilizing data from simulations run using the plain MCTS algorithm. This dataset is crucial for supervised learning and is structured to include inputs and their corresponding correct labels. Here, an input represents a state, while the label denotes the best action as determined by MCTS. Each true label corresponds to one of the fifteen possible actions within our action space, uniquely identified by an action index ranging from 1 to 15.

State in Dataset Representation

The representation of a state involves specific attributes, including:

- **Ego vehicle values:** characterized by parameters $\{p_y, u_x, u_y, w, l, d_u\}$, a total of 6 values. It is important to note that the x position is excluded from training data for the NN as it does not influence the decision-making process regarding the action taken. For a detailed description of what each parameter signifies you can see Table 4.1.

- **Neighbor vehicle values:** which account for the 4 nearest vehicles in front and

the 4 directly in back within a 50-meter visibility range. These are defined by parameters $\{dx, dy, u_x, u_y, w, l, d_u\}$ for each vehicle, summing up to 56 values. Here, dx and dy are calculated as the differences in the x and y positions between the ego vehicle and its neighbors, enhancing the NN's ability to train more effectively by simplifying state-action pattern recognition.

In scenarios where there are more than 4 vehicles either in front or behind within the visibility range, priority is given to those closest to the ego vehicle due to their greater impact in ego vehicle's decision-making. Conversely, if fewer than 4 vehicles are present, "virtual" neighbor vehicles are introduced with predefined parameters, ensuring the state vector's completeness. These virtual neighbors are assigned a dx value exceeding the visibility range, making them not visible to the ego vehicle, which helps it to identify and distinguish real from virtual neighbors in the dataset. So finally, our dataset comprises 63 elements: 62 values represent a single state, and 1 value denotes the best action.

4.3.2 Training and Accuracy

After successfully generating our dataset, we proceed to create our NN. This NN is essentially a classifier designed to take as input a state from the SUMO simulation and output a distribution of probabilities across the action space. This capability enables the NN to effectively predict the most likely action to be taken in any given state.

Proceeding to the next step, we trained our NN, yet despite optimization and fine-tuning of our training parameters, the network's accuracy reached only 72%. This raised concerns about the network's ability to effectively learn from our dataset. Through further experimentation and analysis, we identified the root cause of this limitation: the inherent probabilistic nature of our data, a direct result of the randomness embedded within the MCTS algorithm.

Randomness in MCTS Data

More specifically, during the MCTS simulation phase, random rollouts are executed, wherein random actions are taken that lead to many different child states with each time for a predefined maximum state depth. Consequently, identical states frequently result in different optimal actions, yet the NN simplifies this complexity by selecting the most probable action, without considering the full probability distribution. By contrast, the resulting weights of the NN provide a deterministic policy, and we assess the network's accuracy by assuming a deterministic MCTS agent that generates the dataset.

To illustrate, consider an outcome distribution where the probabilities are 0.68 for Action 1, 0.22 for Action 2, 0.1 for Action 3, and 0 for Actions 4 to 15. Under an ideal sample distribution, the NN would only achieve a 68% accuracy, despite accurately predicting the probability distribution. This is the reason why we achieve this low accuracy. However, just understanding this limitation is not enough and it is important to validate that our NN is correctly functioning and has the capability to learn from the provided data. This validation will be achieved through the calculation of the model's calibration, ensuring the NN's efficacy despite the challenges presented by the probabilistic nature of our dataset.

4.3.3 Reliability Diagrams for Neural Network Evaluation

Starting our analysis, as extensively discussed in Section 2.6.3, our approach involves dividing the dataset into 10 distinct bins. This split is based on the probability of the most probable prediction in the distribution for each sample. The division of bins follows equal confidence levels ranging from [0, 0.1), [0.1, 0.2), and so forth up to [0.9, 1]. Each bin accumulates predictions from samples falling within its specific confidence level. After that we use the Equation 2.19, to calculate the actual confidence for each bin. This calculation is achieved by averaging the confidence levels of all predictions stored within each bin. Subsequently, we must also calculate the accuracy of each bin individually. This step is accomplished by using the Equation 2.18, making the comparison between the predictions of a bin and their corresponding true labels. Finally, to prove that our model is calibrated correctly, calibration accuracy, it should approximate the ideal calibration. As mentioned in Section 2.6.3, optimal calibration is achieved when the accuracy is equal to the confidence for all bins.

Detailed accuracy metrics for each bin are presented in the Table 4.2 below.

Bin	Confidence	Accuracy
1	0.0-0.1	None
2	0.1-0.2	0.159
3	0.2-0.3	0.268
4	0.3-0.4	0.380
5	0.4-0.5	0.487
6	0.5-0.6	0.590
7	0.6-0.7	0.683
8	0.7-0.8	0.773
9	0.8-0.9	0.865
10	0.9-1.0	0.977

Table 4.2: Accuracy in Bins with varying Confidence Levels.

It is noteworthy that the bin representing confidence levels between 0 to 0.1 is empty of accuracy metric. This happens because there are actually no sampled predictions with so low confidence, so the first bin is empty.

Reliability Diagram

To help in the visualization of our model's calibration, we employ reliability diagrams, as referenced in Section 2.6.4. These diagrams plot the comparison between our model's calibration and the benchmark of perfect line calibration. A visual illustration of our models reliability diagram is provided in the Figure 4.4 below. We observe from the diagram that our model closely aligns with the ideal calibration line, underscoring its well-calibrated nature.

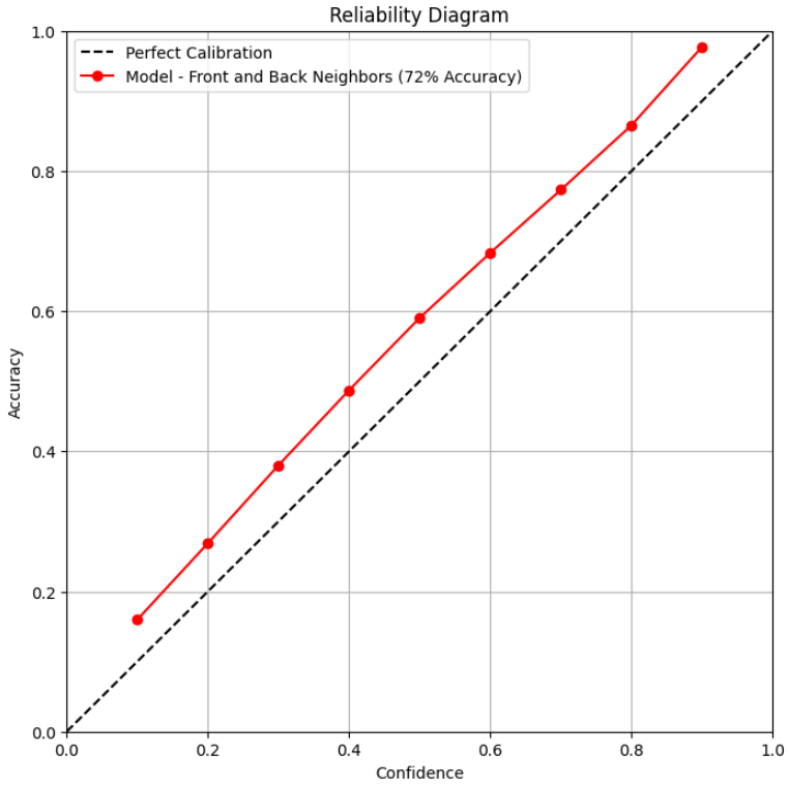


Figure 4.4: Reliability Diagram - Model's Calibration and Perfect Calibration.

Confidence Histogram

However, our analysis does not conclude with this observation. Although reliability diagrams clarify the relationship between confidence and accuracy, they do not provide information regarding the distribution of sample predictions across the bins. To address this gap, we create a histogram of confidence. This histogram serves as a visual representation, detailing the quantity of samples contained within each confidence bin. This illustration of our models histogram of confidence is provided in the Figure 4.5 below.

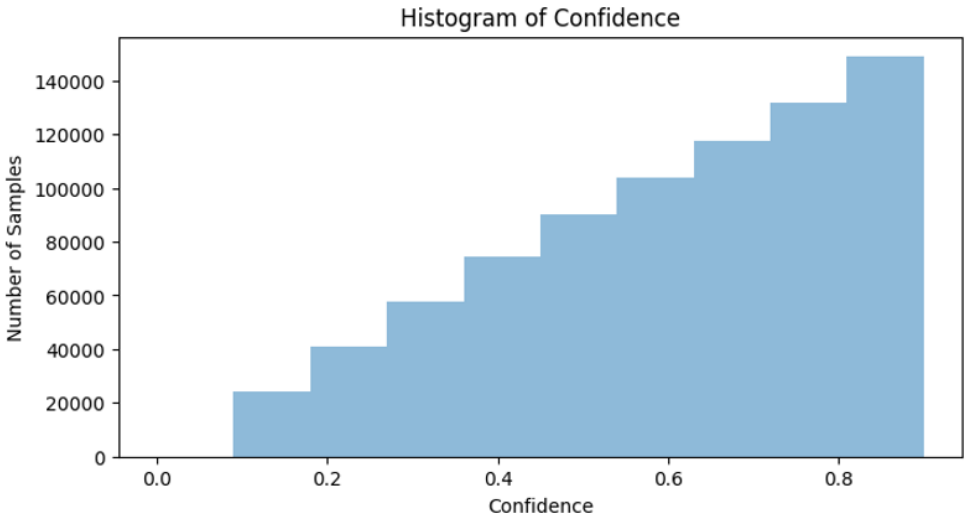


Figure 4.5: Histogram of Confidence - Number of Samples per Bin.

We understand from the figure above that, the samples are distributed in the correct way, as they start from zero and then they gradually increase in each next bin. This means that we have less predictions with low confidence and more with high, affirming the effectiveness of our model's calibration process.

Finally, a straightforward and convincing evidence that our network is capable of learning and performs well is observed when we eliminate the randomness inherent in the MCTS algorithm, resulting in significantly higher accuracy. Specifically, when collecting data from MCTS simulations that excluded random rollouts by setting the maximum rollout state depth to 1, thereby making the next state always deterministic and devoid of randomness, our NN trained with this data achieved an accuracy of 95.8%. This outcome strongly supports our assertion.

4.3.4 Integration

In the next sections, we delve into the detailed process of integrating the NN into the MCTS algorithm, particularly during the selection phase. We methodically explain the steps undertaken not only to develop our algorithm but also to showcase the many various versions of it, before concluding into the final and best version. This iterative process aimed at fine-tuning and optimizing not just the algorithm's performance but also its execution speed. So in this part of our discussion, we focus on the construction and the optimization process, transitioning from a plain MCTS to an NN-Guided MCTS.

The process of integrating the NN with the MCTS, happens during the selection phase and it is described in further detail in Section 2.7.2 and illustrated in Figure 2.15. This integration transforms the selection phase into a guided selection process. Specifically, the Predictive Upper Confidence Trees (PUCT) algorithm, as defined in Equation 2.20, incorporates a weighted sum that includes the prior knowledge $P(s, a)$ provided by the NN. Our focus now shifts towards elaborating on how the algorithm is constructed and operates in practice, moving beyond the theoretical background foundations that were discussed previously.

4.3.5 Model Loading

To initiate our discussion, it's essential to note that at the beginning of our program, before the simulation starts, the NN that was trained and saved previously is loaded. This loading happens once at the program's start, enabling the NN to apply its learned knowledge as needed throughout the simulation. Importantly, the NN's training is conducted offline, prior to the guidance stage, and the model is loaded using a function named `load_model()`.

4.3.6 Predictions: Initial Approach

Single Predictions in Current State

Moving forward, the initial step in our process involves the creation of a function de-

signed for generating predictions, denoted as $P(s, a)$. This predictive function, called **`make_single_predictions(state, model)`**, takes as inputs the current state of the SUMO simulation of the ego vehicle (a vector of 62 elements) and the loaded model. It then returns as output the probability distribution of the different actions (a vector of 15 elements) for that specific state.

The first version of our algorithm was designed to process each step of the SUMO simulation by capturing the current state and constructing from it the 62-element input vector required by the NN for generating predictions. Following this step, the constructed input was fed into the **`make_single_predictions`** function, producing a 15-element predictions output indicative of the different action probabilities. This output predictions, along with the current state, was then given as an input in the MCTS's selection phase, to a function now named **`select_predictions(state, predictions)`**. This function employed the PUCT tree policy to return the best child action based on the predictions.

- **Inefficient and Slow Functionality:**

  However, this initial approach was very slow (high in temporal complexity) for several reasons that need to be addressed. The primary issue was that each time the algorithm entered the selection phase, it needed to repeat the entire process from the beginning. To elaborate, for each selection, it was necessary to reconstruct the input vector from the current state and recalculate the predictions. Given that in each MCTS iteration, in order to navigate the tree until a leaf node is selected for expansion, there are multiple repetitions of the selection process, and very often there are revisits of previously encountered states. Thus, this method resulted in redundant calculations (both in constructing inputs and generating predictions), performing the same operations multiple times.

- **Functionality and Speed Improvement:**

  To reduce this inefficiency, a pivotal adjustment was made: storing each node's predictions distribution as additional information within the node itself. Therefore, when encountering a new state for the first time, the standard predictions calculation procedure is executed. Conversely, if a state has been revisited, a verification check is done to confirm this, and the predictions are retrieved from the node's already stored predictions. This simple modification significantly accelerated the algorithm's speed by eliminating the need to repeatedly calculate identical values, thereby streamlining the overall process. Naturally, this requires more memory, but the overhead in memory usage is negligible in practice.

- **Inefficient and Slow Predictions:**

  Following the optimization of the algorithm's external procedures, our focus shifted towards enhancing the efficiency of the **`single_predictions`** function by examining ways to speed its internal processes. Through exhaustive testing, we identified a key area for improvement: transitioning from making single predictions to adopting a batch prediction approach. Initially, the **`single_predictions`** function operated by calling a Python script from C++, where the programming environment of MCTS is implemented. This Python script was responsible for generating predictions based on the trained NN, as briefly outlined in Section 4.1.1.

Our analysis revealed that a significant portion of the prediction process's time was consumed not by the computational efforts of generating predictions within Python but rather by the overhead associated with calling the Python script itself. Recognizing this, it became clear that taking advantage of the network's inherent capability for batch processing could offer a notable efficiency boost. This is because, NNs are well-suited to processing multiple inputs in parallel, allowing for the simultaneous calculation of prediction distributions for various states.

- **Predictions Speed Improvement:**

  Therefore, instead of persisting with the method of generating single predictions for each state, which necessitated repeated calls to the Python script, we proposed a strategic shift towards batch predictions. By gathering multiple states and computing their predictions in a single batch, we could significantly reduce the frequency of script calls. This approach not only minimizes the overhead associated with these calls but also maximizes the computational efficiency of the prediction process, leading to a faster and more streamlined algorithm.

Batch Predictions in Child Expansion

Our initial strategy for implementing batch predictions involved generating predictions in batches of 15, corresponding to each potential child action of a node. More specifically, when expanding a node, we employed the `make_batch_predictions` function to compute the predictions for its 15 potential child states and stored each one of these predictions within the respective child nodes. This approach ensured that when revisiting any of these child states, we could directly use the previously calculated predictions, eliminating the need for recalculations. An exception to this method was the root node, which, lacking a parent node, required a single prediction as was explained before, to determine its initial set of predictions.

- **Speed Improvement:**

  In Table 4.3, we present a comparative analysis of the time required to make a single prediction versus the time needed for different batch predictions. The results showcase a significant improvement in speed efficiency, by shifting to batch processing for 15 predictions at once, the average time required for a single prediction is 13 times faster in comparison with doing 15 single predictions.

  It is important to acknowledge that this method involves pre-calculating predictions for child states that may not ultimately be visited within the simulation, leading to the generation of some redundant information. However, this drawback was deemed inconsequential in the broader context of performance enhancement. The overall process was significantly accelerated, making the occasional calculation of unused predictions a worthwhile trade-off for the substantial gains in speed and efficiency.

Batch Tree Predictions in Algorithm's Start

Exploring other ways to enhance the speed of our predictions procedure led us to the development of the following adjustment. Rather than generating batches of predictions to

store in each node individually, we instead rely on a more efficient approach: calculating the entire set of predictions for the MCTS state space in one go and storing them in an external predictions matrix. This method represents the maximum usage of the NN batch processing capabilities to their fullest extent in our problem.

This optimization is executed by initially constructing the complete tree state space, involving the reconstruction of all necessary inputs for the NN from the states, up to a certain predefined maximum depth of the tree. Subsequently, a tree batch prediction is made for all these states, with the resulting predictions being stored within a matrix. This crucial step is carried out once at the root node, prior to running the MCTS algorithm. Thereafter, whenever predictions for a specific state are required, they are retrieved directly from this predictions matrix, which contains all the needed information. This approach results in significant time savings, which will be detailed in the following section. Additionally, it not only eliminates the need to store predictions within each node separately, but also ends the necessity for any predictions calculations during the MCTS runtime.

Detailed speed metrics for each batch predictions are presented in the Table 4.3 below.

Predictions Batch Size	Tree Depth	Time needed in (ms)
1	1	26
16	2	30
241	3	44
3616	4	182

Table 4.3: Time in milliseconds needed for varying Predictions Batches.

4.3.7 Algorithm Implementation

Finally, here we provide an overview of the algorithm's operational mechanics.

- Initially, prior to initiating MCTS process, we start from the root node and reconstruct the inputs for the state space tree. This reconstruction employs a Breadth-First Search (BFS) approach, which facilitates an orderly numbering of our predictions matrix, making it more straightforward to manage.

The decision to construct the full tree up to a maximum depth of 3 was decided by experimental observations. It was noted that the common practice of visiting and expanding a state typically in most cases, does not exceed depth 3, with expansions to depth 4 occurring only on rare occasions. Essentially, expanding a state at depth 3 results in a total tree depth of 4, but the necessity to visit and expand a state at depth 4 (thereby extending to a total depth of 5) is very uncommon. Consequently, the additional time required by the NN to generate predictions up to depth 4 is deemed inefficient, as it would result in multiple predictions for states that are unlikely to be visited and thus they will remain unused.

This principle is illustrated in Table 4.3, where the exponential increase in tree branching with each additional depth level leads to a corresponding sharp rise in computational time from depth 3 to 4. The total number of states at depth 3 sums up to 241 (calculated as 1 +

15 + 15×15), while at depth 4, it escalates to 3616 (241 + 15×15×15). This exponential increase in the number of states accounts for the drastic change in computational time, underscoring why extending predictions beyond depth 3 is not worth having from a time-efficiency perspective.

- In the subsequent step, we are generating predictions for the entire tree inputs using the **`make_batch_tree_predictions`** function. Following the generation of these predictions, they are methodically stored within the predictions matrix. Both of these procedures are executed once, starting from the root node, and are completed prior to initiating the MCTS algorithm.
- Following the setup phase, as the algorithm initiates, we begin by calculating the depth of the current state to ascertain whether it falls within the depth of our pre-calculated tree (thus it is smaller than or equal to the maximum depth). If this criterion is met, we proceed to trace the path leading to the current state, starting from the root node. This trace includes identifying each ancestor of the current state (parent node, grandparent node, and so on). This step is crucial for locating the current state's predictions within the predictions matrix. By utilizing the traced path, we compute the index of the current state, which corresponds to the specific location within the predictions matrix where the state's predictions are stored.
- Subsequently, we retrieve the prepared predictions from the matrix. As in the previous workflow, these predictions are then fed into the **`select_predictions`** function, which utilizes them to guide the search process.
- In rare scenarios where the current state's depth exceeds the depth of our pre-calculated tree (thus it is greater than the maximum depth), we just make a single prediction for that specific state. This is a one-time procedure meant to generate the necessary predictions for states beyond the pre-defined depth limit.

A high level implementation of the NN-Guided MCTS is shown below Algorithm 2, as well as the new refined selection phase of the algorithm Function 1. Note that the red lines of code, indicate the differences the new algorithm has, in comparison with the plain MCTS.

Speed Improvement

The efficiency gains from this final refined algorithm are substantial. As detailed in Table 4.3, the prediction time in this latest version is observed to be 10.2 times faster compared to the previous 15-batch prediction approach, and an impressive 132.6 times faster than the original single prediction method. This marked improvement underscores the significant advancements achieved in optimizing the algorithm's speed.

Algorithm 2: A High Level NN-Guided MCTS implementation

```
model ← load_trained_neural_network_once ()
-----------------------------------------------------------------------
actions_per_state ← LATERAL ACTIONS · LONGITUDINAL ACTIONS
iterations ← 0
root ← initialize_tree ()
input_data_tree ←
 reconstruct_inputs_tree (MAX DEPTH, root, FRONT&BACK NEIGHBORS)
predictions ← make_batch_predictions (model, input_data_tree)
while iterations ≤ MAX ITERATIONS do
    if state_depth ≤ MAX DEPTH then
        path_to_state ← find_path (root)
        state_index ← find_index (path_to_state, state_depth, actions_per_state)
        state_prediction ← predictions [state_index]
        selected_state ← select_with_prediction (root, state_prediction)
    end
    else
        input_data_single ← reconstruct_input (root, FRONT&BACK NEIGHBORS)
        state_prediction ← make_single_prediction (model, input_data_single)
        selected_state ← select_with_prediction (root, state_prediction)
    end
    if is_terminal (selected_state) then
        continue
    end
    if number_of_visits (selected_state) ≥ MIN VISITS then
        expanded_state ← expand (selected_state)
    end
    else
        expanded_state ← selected_state
    end
    score ← simulate (expanded_state)
    backpropagate (expanded_state, score)
    iterations ← iterations + 1
end
```

Algorithm 2: Function 1: Selection Phase of NN-Guided MCTS

```
Input: state, prediction
children ← get_children (state)
for child ∈ children do
    score ← PUCT (state, child, prediction) (2.20)
end
best_child ← argmax_child (score)
return best_child
```

4.4 Deep Neural Network without Search

Finally, in this section, we present a simple alternative for comparison purposes that exclusively leverages the knowledge of the NN in a greedy manner, without any underlying search mechanism like MCTS. This approach is adopted to assess the standalone potential and efficiency of the NN, isolated from the complexities of the MCTS framework.

In practice, this means that at each current state encountered, the algorithm just performs a single prediction as previously outlined. Based on this prediction, it selects the action with the highest probability from the action distribution $P(s, a)$ as the optimal action to execute. This process showcases a straightforward, yet inherently greedy algorithm that may not yield the most effective outcomes due to its simplistic approach.

Despite its potential limitations, this algorithm is constructed to facilitate a comparative analysis of its performance against both the traditional MCTS and the advanced NN-Guided MCTS algorithms. The goal is to examine the efficacy of relying only on NN knowledge in decision-making scenarios, providing insights into the benefits and drawbacks of such a strategy.

Algorithm 3: Usage of the Neural Network without Search (Greedy Approach)

model ← load_trained_neural_network_once ()

---

root ← initialize_tree ()
input_data_single ← reconstruct_input (*root*, FRONT&BACK NEIGHBORS)
state_prediction ← make_single_prediction (*model*, *input_data_single*)
children ← get_children (*root*)
for *child* ∈ *children* **do**
  score ← *state_prediction* [*child*]
end
selected_state ← $argmax_{child}$ (*score*)

Chapter 5

Experimental Evaluation

In this section, we first present the environment setup and simulation parameters utilized in our experiments, alongside detailing the agent and hyperparameter configurations critical to the algorithms' operation. Then, we discuss our experimental evaluation process step by step, highlighting the procedure that we followed in each phase. Our analysis primarily focus on comparing outcomes derived from four central algorithms: the original plain Monte Carlo Tree Search (MCTS) developed in previous work, as referenced in Section 3.2.2, our enhanced version of plain MCTS, the NN-Guided MCTS, and lastly, the straightforward Neural Network (NN) approach devoid of any search mechanism.

Subsequently, we examine the performance of each method across a set of scenarios with ranging difficulty. Finally, our systematic evaluation highlights the observed benefits and/or limitations of each alternative depending.

5.1 Environment and Simulations Setup

In this section of our work, we detail the specific settings and parameters of the lane-free simulation environment that were essential for conducting our experiments. For those seeking a deeper understanding of the lane-free environment, an examined detailed description of the problem can be found in Section 4.1.1.

The Table 5.1 presented above enumerates the different parameters that were selected for our SUMO simulations. These include the physical attributes of the vehicles involved, the characteristics of the roadway and the relation of simulation time to actual time. For readers interested in an explanation of either the vehicle-specific or road-related parameters, further information is available in Section 4.1.1 and Table 4.1.

Parameter	Value
Simulation Time (sec)	3600
Simulation Time-Step (sec)	0.25
Vehicle Length (m)	3.5
Vehicle Width (m)	1.6
Veh/h	5400, 9400, 12000
Departure Speed (m/s)	25
Desired Speed (m/s)	[25, 35]
Road Length (m)	500

Table 5.1: SUMO Simulation Parameters.

5.2 Agent and Hyper-parameter Setup

In this section, our focus shifts to the details of the parameters relevant to our agent. This includes the various parameters of the MCTS algorithm that required fine-tuning, as well as those necessary for the dataset creation and training of our NN for its ultimate integration into the NN-Guided MCTS algorithm. Additionally, we detail some technical elements that comprise our agent, along with information related to learning and simulation in lane-free traffic environments.

- Initially, the plain MCTS algorithm for lane-free traffic environments was developed in prior work [34] that we expand upon in this thesis.
- The development of the NN relies on Python, utilizing the Tensorflow-Keras library [76]. This choice was motivated by the library's capability to accelerate NN predictions, which was particularly beneficial for batch processing, which requires significant computational resources for parallel operations. The acceleration is primarily achieved through GPU utilization, facilitated by CUDA [77] and cudnn [78] technologies. For tasks such as dataset standardization, preprocessing, and management we employed the Pandas [79] and Scikit-Learn [80] libraries. The optimization of the model architecture and the fine-tuning of parameters were conducted using Optuna [81]. Moreover, the visualization of our results was accomplished with the Matplotlib library [82].
- The integration of the NN within the MCTS framework was achieved with Pybind11 [83], a library that enables the execution of Python scripts from C++ during actual runtime.

NN Architecture

The architecture of our deep NN is of the feed forward type. It comprises an input layer with 62 units representing a MCTS state, followed by three hidden layers containing 512, 256, and 128 neurons, respectively. An output layer with 15 units represents the potential actions of MCTS. Additionally, dropout layers are interspersed among these layers to help

prevent overfitting.

The Tables 5.2, 5.3 and 5.4 below list the diverse parameters that were critical to the functionality and development of our algorithms, offering insight into the technical fine-tuning of our agent. More specifically, they list the NN's architecture and training parameters, the MCTS algorithm parameters and the program parameters used for development.

Parameter	Value
Input Layer	62 (one MCTS state)
Hidden Layers	512, (0.3 dropout), 256, (0.3 dropout), 128
Output Layer	15 (MCTS possible actions)
Type	Feed Forward
Activation Function	ReLU (hidden layers), Softmax (output layer)
Optimizer	Adam
Batch Size	64
Epochs	50
Learning Rate	start 0.001 and descending
Accuracy	72%(rollout depth 6) , 95.8% (rollout depth 1)
Front Neighbors (dataset)	4
Back Neighbors (dataset)	4
Visibility Target (m)	50

Table 5.2: NN Architecture and Training Parameters.

Parameter	Value
D	10
ϵ	1
Min Visits	5
Max Rollout Depth	6
Lateral Actions	-1, 0, 1
Longitudinal Actions	-5, -2, 0 , 2, 5

Table 5.3: MCTS Parameters.

Parameter	Value
Device	GeForce 3060 Ti
cudnn capability	8.6
Python Version	3.10.8
Tensorflow Version	2.9.1

Table 5.4: Program Parameters.

5.3 Experiments: Setup and Description

In this section, we outline the different experiments that evaluate all methods. These experiments were structured to assess the algorithms across various metrics-that are important to autonomous driving, as well as their adaptability to different configurations of the MCTS algorithm, specifically concerning the number of iterations. This extensive testing, enables a thorough comparison of the algorithms, illustrating their performance and behavior across diverse conditions.

5.3.1 Algorithms and Metrics

Examined Algorithms

The algorithms subjected to our experiments, each discussed in detail within our study, include:

- **MCTS (front):** This first variant of simple MCTS is the preexisting implementation, which is characterized by its limitation where the ego vehicle is only aware of neighbor vehicles in front of it.
- **MCTS (front&back):** Represents our enhanced version of the plain MCTS algorithm, which additionally includes information of nearby vehicles located on the back. This results in a more comprehensive situational awareness, thereby improving decision-making capabilities.
- **NN-MCTS:** Our final version of the algorithm that synergizes the predictive insights from NNs prior knowledge with MCTS, thereby augmenting the search process with data-driven guidance.
- **NN:** A straightforward implementation of the NN without incorporating any search methodologies. This version serves to evaluate the NN's standalone efficacy in decision-making without the computational support of MCTS.

These algorithms were subjected to a variety of tests, each designed to probe different aspects of their functionality and impact on autonomous driving performance. Through these experiments, we aim to derive insights into the advantages and potential limitations of each algorithm.

Assessed Metrics

The performance of our algorithms will be assessed using several key metrics, designed to evaluate their effectiveness in various aspects of driving simulation. These metrics include:

- **Collisions:** This metric counts the total number of collisions that occur during a simulation. In the context of this study, a collision is defined as an incident where two vehicles come into contact with one another. It is important to note that in the lane-free environment, collisions do not impact the ongoing route or traffic flow of the involved vehicles or any others neighbor vehicles. Instead, they continue on their predefined paths without causing traffic slowdowns or jams. Fewer collisions indicate superior algorithmic decision-making, showcasing an enhanced ability to dodge obstacles, to nudge and maneuver safely between other vehicles.
- **Speed Average:** This metric calculates the average (mean) speed of all vehicles throughout the simulation. Given that the speed of vehicles is selected randomly from a uniform distribution between 25 and 35 m/s, the theoretical ideal average speed, assuming a perfect distribution and scenarios where vehicles maintain their preferred speed without interference, would be around 30 m/s. However, this ideal is impossible to achieve in practice due to the presence of multiple vehicles, the initial

and resultant traffic conditions, and various decision-making processes like braking, slowing down, and executing maneuvers to prevent collisions. These factors lead to a reduction in the average speed. Therefore, we expect an average speed lesser than 30 m/s. As closer we get to that number, it is suggested that the algorithm is making decisions that are closer to optimal, enabling vehicles to travel near their desired speeds at every moment of the simulation.

- **Average Delay:** This metric represents the average of the delay values across all vehicles in the simulation. The delay for each vehicle is calculated by the difference between the actual time the vehicle spends traveling in the road and the ideal time calculated based on the vehicle's desired speed, as shown:

$$delay = actual\ time - ideal\ time,\ \text{where}\ ideal\ time = \frac{road\ length}{desired\ speed}\ .$$

  Average delay is computed by taking the mean of all individual delay measurements. A high average delay indicates general inefficiencies due to suboptimal algorithmic decisions. A low average delay suggests that the vehicles, on average, are able to travel close to their desired speeds, indicating effective algorithm performance.

By analyzing these metrics, we can gain insights into how well each algorithm performs under different simulation conditions, focusing on their ability to minimize collisions, maximize speed efficiency, and reduce delays.

5.3.2 Environment and MCTS Settings

Examined Flow Demands on an Open Highway

Our algorithms will undergo testing across varying levels of vehicle flow, incrementally increasing to examine their efficacy under progressively challenging scenarios. As the vehicle flow per hour rises, it signifies a denser traffic situation on the road at any given moment, thereby complicating the decision-making processes for the vehicles. With more vehicles sharing the road space, vehicles are necessitated to execute more delicate maneuvers and exercise increased caution (as braking more frequently) to avoid collisions.

An inevitable consequence of denser traffic and the increased risk of collisions is a reduction in the average speed of vehicles and an increase in the average delay time. Higher vehicle flows with aggressive vehicle policies can also lead to potentially unavoidable collisions in particularly complex traffic situations. To comprehensively assess the performance of our agents, we subject them to tests in three distinct vehicle flow rates:

- **5400 veh/h:** Represents a low level of traffic flow, serving as a baseline for evaluating how our algorithms manage under typical conditions.
- **9400 veh/h:** An intermediate traffic scenario with many situations that vehicles need to properly maneuver in order to overtake or make space. This flow rate tests the algorithms' capabilities to maintain efficiency and safety in denser traffic.
- **12000 veh/h:** The highest level of traffic flow in our tests, challenging the algorithms to operate also in potentially congested conditions, assessing their decision-making for maneuvering and collision avoidance in intense situations.

By analyzing the algorithms' performances across these different flow rates, we can showcase their scalability in handling various levels of complexity.

MCTS Iterations

Continuing our experimental setup, we conduct a series of simulations with a varying number of MCTS iterations for each algorithm. This variation aims to capture the differential capabilities of our algorithms, observing their performance across a spectrum from a small to a large numbers of iterations. Through this approach, we aim to identify each algorithm's performance peak, where it yields the best results, as well as understand its behavior under constrained iteration conditions. This investigation is crucial for determining the efficiency of each algorithm in navigating the MCTS tree to identify the most effective actions with the least computational effort.

The core of this experimental design is to ascertain the minimum number of iterations required by each algorithm to achieve optimal results. An algorithm that requires fewer iterations to find the best solution is considered more efficient, indicating lower computational resource demands and faster search processes. Therefore, comparing the performance outcomes of each algorithm across various iteration counts will reveal their relative efficiency and speed. More importantly, with this information we examine to what extend and in which conditions the inclusion of NN accelerates performance with respect to the standard MCTS.

It is important to note that while the first three algorithms, which incorporate variations of MCTS, will be assessed across a range of iteration numbers, the plain NN approach, devoid of MCTS integration, will be evaluated in a singular experiment per scenario. This is because the plain NN's performance does not depend on iteration numbers.

The iteration counts selected for our experiments represent a sampled range, chosen to provide a comprehensive overview of each algorithm's performance across different iteration scales. The specific iteration numbers chosen for testing are as follows:

- **30 iterations:** Represents the lowest number of iterations, testing the algorithms' immediate response capabilities.
- **50 iterations:** A slightly higher iteration count, allowing for a little more nuanced exploration of the search space, but it is still very low.
- **100 iterations:** Offers a moderate level of depth in the search process, balancing speed and thoroughness.
- **200 iterations:** Provides a deeper exploration, potentially uncovering more effective strategies.
- **500 iterations:** A significant increase, aimed at testing the algorithms' performance under extensive search conditions.
- **750 iterations:** Approaches the upper limit of our testing range, challenging the algorithms with a dense search requirement.

- **1000 iterations:** The highest iteration count in our experiments, designed to assess the algorithms' ultimate performance and decision-making quality.

Through these experiments, we gain valuable insights into the scalability and efficiency of each algorithm under varying computational constraints.

At this point where we have detailed the concept of iterations, we must clarify that the dataset for training the NN was derived from MCTS simulations with 1000 iterations. This particular number of iterations was chosen to ensure that the data reflects the best performance of the MCTS, thereby providing a higher-quality policy for the NN's learning process.

Simulations Executed

The simulations we conducted, span a duration of 1 hour (3600 sec) each. For each combination of iterations number and vehicle flow, we ran simulations with 5 distinct simulation seed values for every algorithm. This allowed us to account for variance due to the stochastic nature of MCTS and for vehicles (i.e., their desired speed objective) of each scenario.

From these series of experiments, we accumulated the data to calculate the mean values and standard deviations for the metrics of interest: collisions, average speed, and average delay.

To encapsulate the entirety of our experimental work, the total count of 1-hour simulation runs conducted is broken down as follows:

- For the three MCTS-based algorithms, we explored 3 levels of vehicle flow, evaluated across 7 sampled iteration counts, and tested with 5 different experiment seeds. This results in a substantial number of individual simulation runs.
- Additionally, for the plain NN algorithm, which does not vary with iteration numbers, we conducted 15 simulations, accounting for the different vehicle flows and experiment seeds.
- **Cumulatively, the total number of 1-hour simulation runs amounts to:** (3 MCTS algorithms) x (3 vehicle flows) x (7 sampled iterations) x (5 experiment seeds) + (15 for the NN algorithm) = 330 simulations.

5.4 Results and Analysis

In this section, we present the results obtained from the extensive simulations discussed earlier. These results will be plotted in three distinct types of graphs for each vehicle flow: collisions-iterations, average speed-iterations, and average delay-iterations. Each graph will plot the data for all four algorithms we tested, allowing for direct comparison and immediate visual discernment of differences among the algorithms.

5.4.1 Graphs Visualization Details

It is important to note that in all our graphs, the iterations axis is displayed on a logarithmic scale. This choice is deliberate, it enhances visibility of the algorithm's performance at lower iterations numbers, where more significant variances in results are observed due to the algorithms not performing optimally. Conversely, at higher iterations numbers, the results across different algorithms tend to converge and show minimal variation. Therefore, a detailed view of the higher iteration range is less critical for our analysis, as the performance improvements stabilize.

Moreover, each data point represented in these graphs corresponds to the mean values derived from five different seed experiments for each iteration sample. This approach ensures that our results are robust and reflect the average outcome rather than an outlier. Specifically, for the collisions-iterations graphs, we have also included error bars that represent the standard deviation from the mean, providing an insight into the variability of the results. This was deemed necessary for the collision metric due to its potentially wider variation in outcomes, especially for a small number of iteration where (as expected) MCTS inherently exhibits higher variance. However, for the average speed and delay graphs, standard deviations were excluded because they were consistently minimal and would not be visually detectable on the graph, thus adding no significant value to the visualization.

Through these graphical representations, we aim to provide a clear visualization of how each algorithm performs under different conditions, highlighting their efficiency and effectiveness in handling varying traffic densities and operational scenarios.

5.4.2 Collisions

We begin by presenting the results from the collisions-iterations diagrams for each different vehicle flow. Below, tables are provided that document the exact number of mean collisions, along with the $\pm$ (standard deviations) from all algorithms for every iteration sample and for each vehicle flow. These tables serve as a detailed resource for readers who may wish to examine more specific results.

Collisions for 5400 veh/h

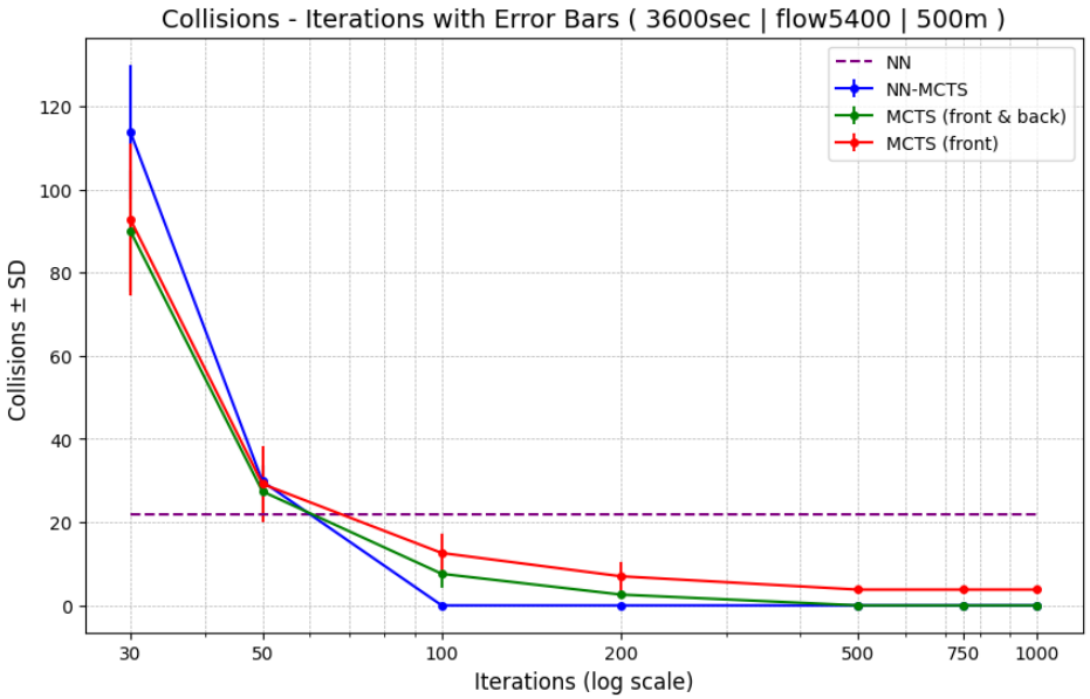


Figure 5.1: Graph of Collisions±SD for NN-MCTS, MCTS (front&back), MCTS (front) and NN across different Iterations for 5400 veh/h.

NN-MCTS	MCTS (front&back)	MCTS (front)	Iterations
0	0	3.8 ± 0.84	1000
0	0	3.8 ± 0.84	750
0	**0**	$\mathbf{3.8 \pm 0.84}$	**500**
0	2.6 ± 0.9	7 ± 3.39	200
0	7.6 ± 3.4	12.6 ± 4.77	**100**
29.8 ± 3.9	27.4 ± 6.3	29.2 ± 9.04	50
113.8 ± 16.2	90 ± 14.6	92.8 ± 18.28	30

Table 5.5: Comparative Results of Collisions±SD for NN-MCTS, MCTS (front&back), and MCTS (front) across different Iterations for 5400 veh/h.

Collisions for 9400 veh/h

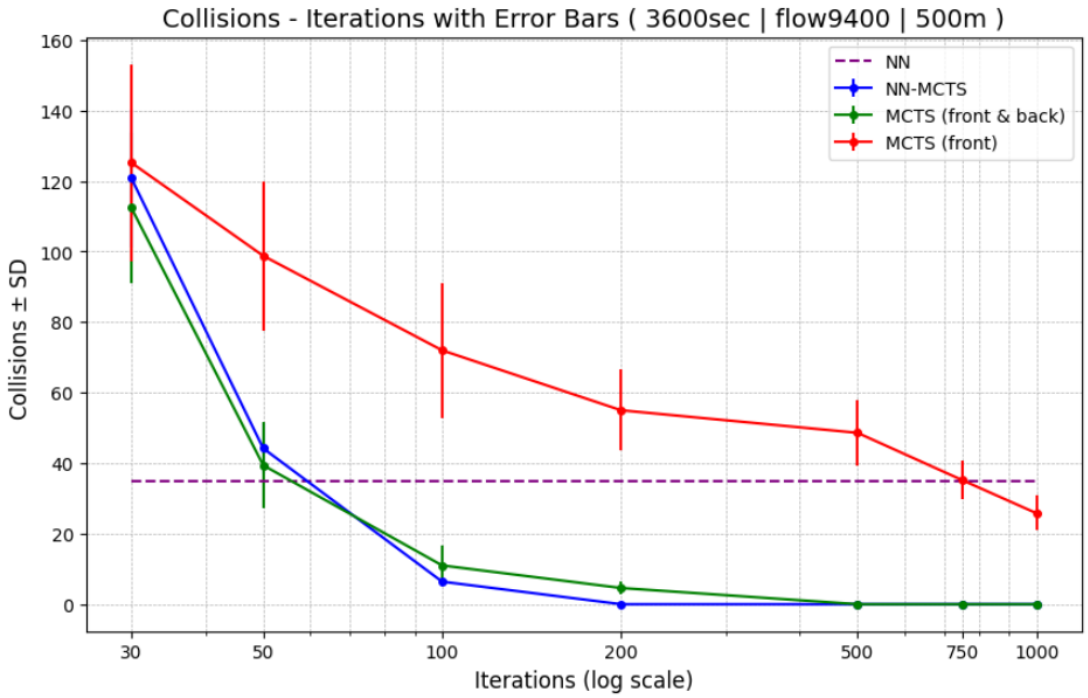


Figure 5.2: Graph of Collisions±SD for NN-MCTS, MCTS (front&back), MCTS (front) and NN across different Iterations for 9400 veh/h.

NN-MCTS	MCTS (front&back)	MCTS (front)	Iterations
0	0	**25.8 ± 4.92**	**1000**
0	0	35.2 ± 5.54	750
0	**0**	48.6 ± 9.37	**500**
0	4.6 ± 1.82	55 ± 11.47	**200**
6.4 ± 1.52	11 ± 5.52	72 ± 19.07	100
44.2 ± 7.09	39.4 ± 12.3	98.8 ± 21.1	50
120.8 ± 19.43	112.4 ± 21.33	125.2 ± 27.93	30

Table 5.6: Comparative Results of Collisions±SD for NN-MCTS, MCTS (front&back), MCTS (front) across different Iterations for 9400 veh/h.

Collisions for 12000 *veh/h*

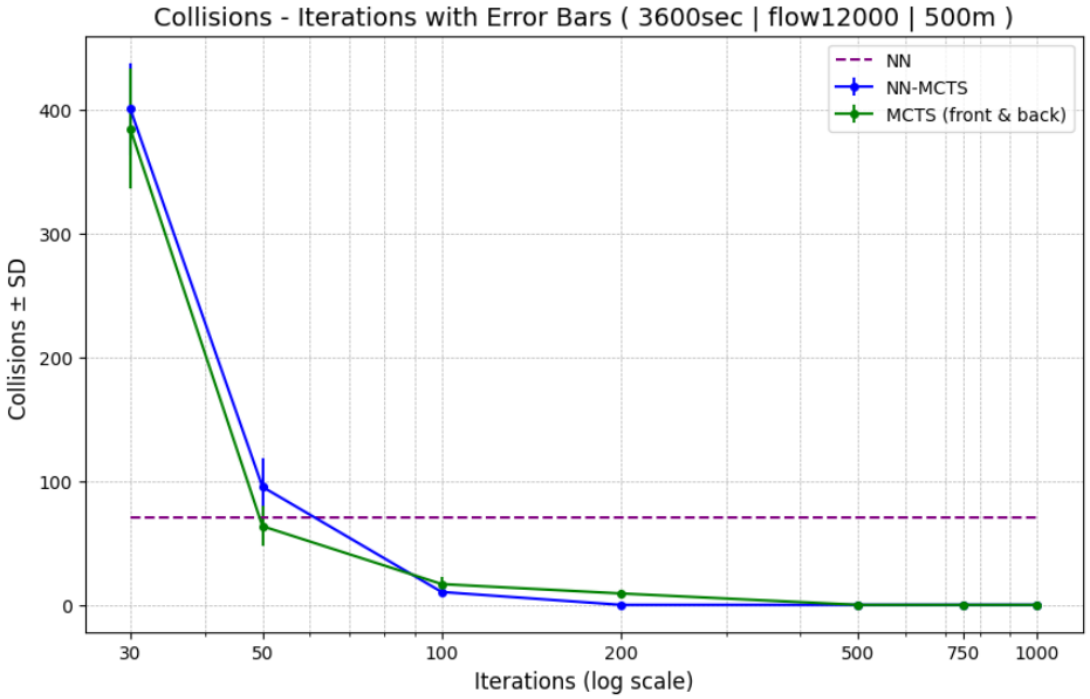


Figure 5.3: Graph of Collisions±SD for NN-MCTS, MCTS (front&back), MCTS (front) and NN across different Iterations for 12000 *veh/h*.

NN-MCTS	MCTS (front&back)	MCTS (front)	Iterations
0	0	-	1000
0	0	-	750
0	**0**	-	**500**
0	9.2 ± 2.58	-	**200**
10.4 ± 2.6	16.8 ± 5.4	-	100
95.4 ± 23.1	63.6 ± 16.21	-	50
400.8 ± 37.29	384.8 ± 48.33	-	30

Table 5.7: Comparative Results of Collisions±SD for NN-Guided MCTS, MCTS (front&back), and MCTS (front) across different Iterations for 12000 *veh/h*.

5400 veh.flow/h.	9400 veh.flow/h.	12000 veh.flow/h.
22	35	61

Table 5.8: NN Collisions for Each *veh/h*.

Results at Minimum Number of Iterations

From the Figures 5.1, 5.2 and 5.3, we first observe that every MCTS algorithm starts with a relatively high level of collisions which decreases as the number of iterations increases. This result aligns with expectations, as increasing iterations allow the algorithm to make more sophisticated decisions, explore the state space tree with higher depth, the state space tree, and consequently discover more informed and effective solutions to avoid collisions.

Notably, each algorithm reaches a point where the number of collisions minimizes, often approaching 0. This inflection point suggests that the algorithm has nearly optimized its performance with the given computational resources. After reaching this minimum level of collisions, the results tend to stabilize and further increases in iterations do not significantly alter the number of collisions, indicating that the algorithm has reached its operational peak in terms of collision avoidance.

As outlined in Section 5.3, our primary aim is to ascertain at which iterations number each algorithm achieves this stabilized or optimal performance. Additionally, we seek to compare these optimal solutions across different algorithms to understand their relative effectiveness and efficiency. This comparative analysis will provide valuable insights into the capabilities and limitations of each algorithm under varying conditions of vehicle flow and computational effort.

Comparison: MCTS (front) vs MCTS (front&back)

In our comparative analysis, the initial focus is on contrasting the MCTS (front) version [34] with our improved MCTS (front&back) model. Across all vehicle flows, MCTS (front&back) consistently outperforms MCTS (front) in terms of collision reduction, as shown in Tables 5.5, 5.6 and 5.7 .

- **5400 *veh/h*:** Both algorithms stabilize at 500 iterations. However, MCTS (front&back) achieves superior results with 0 collisions, whereas MCTS (front) still records an average of 3.8 collisions even at its most optimized state.
- **9400 *veh/h*:** Once again, MCTS (front&back) maintains 0 collisions at 500 iterations, indicating well performance even under increased traffic density. By contrast, MCTS (front) exhibits a significant number of collisions at the same number of iterations, 48.6 collisions, which only marginally improves at its peak performance of 1000 iterations to 25.8 collisions.
- **12000 *veh/h*:** MCTS (front&back) continues to excel by maintaining 0 collisions at 500 iterations. By contrast, MCTS (front) struggles with this higher traffic density, leading to a scenario where it fails to run effectively, simulations indicate that all vehicles end up colliding at the front of the road, preventing the simulation from proceeding and not executing at all.

Moreover, a noteworthy observation is the larger standard deviations associated with MCTS (front), suggesting that its performance and decision-making are not only worse on average but also showcases higher variance and therefore a less reliable policy. The

results are not consistently stable and worsen under intensified traffic conditions.

These findings clearly demonstrate that our enhanced version of plain MCTS, which includes the capability to observe both front and back neighbors, significantly improves the algorithm's performance. It is evident that MCTS (front), with its limited visibility, lacks sufficient information in each state to effectively handle complex scenarios. While it may perform acceptably under lower traffic conditions, its effectiveness diminishes exponentially as vehicle flow increases, ultimately failing to function under the most challenging conditions (12000 vehicle flow).

Comparison: MCTS (front&back) vs NN-MCTS

Continuing our comparative analysis, we now examine the performance of NN-Guided MCTS (NN-MCTS) versus MCTS (front&back) to determine the efficacy of NN integration in enhancing the algorithm's performance. According to the data from Tables 5.5, 5.6 and 5.7, both algorithms achieve 0 collisions at their best operating points across all vehicle flows.

- **5400 *veh/h*:** NN-MCTS reaches 0 collisions remarkably quickly, requiring only 100 iterations. By contrast, MCTS (front&back) still reports 7.6 collisions at this iteration count and needs 500 iterations to achieve 0 collisions.
- **9400 *veh/h*:** NN-MCTS again demonstrates superior efficiency by stabilizing at zero collisions after just 200 iterations. Meanwhile, MCTS (front&back) only attains 0 collisions at 500 iterations, with 4.6 collisions reported at the 200 iteration mark.
- **12000 *veh/h*:** The pattern persists with NN-MCTS stabilizing at 0 collisions at 200 iterations. MCTS (front&back), however, suffers 9.2 collisions at the same number of iterations and requires up to 500 iterations to eliminate collisions and reach 0.

Additionally, the results indicate an increase in collisions at lower iteration numbers as vehicle flow increases, suggesting that more complex traffic scenarios challenge the algorithms' ability to quickly converge on better policies for the agents. This highlights the increasing difficulty in collision avoidance under denser traffic conditions without sufficient iterations for deeper search. To put it simply, when algorithms do not have a sufficient number of iterations, they perform worse in each vehicle flow increment.

These findings illustrate that NN guidance significantly boosts MCTS performance by leveraging prior knowledge to accelerate the search process, achieving optimal results with fewer iterations. Specifically, NN-MCTS requires 400 fewer iterations to reach 0 collisions in a 5400 vehicle flow and 300 fewer iterations in the more challenging flows of 9400 and 12000 vehicles. This efficiency gain not only demonstrates the NN's effectiveness in guiding the MCTS but also highlights the potential for computational savings. By reducing the number of necessary iterations, NN-MCTS cuts down on the computational complexity, making it a more efficient choice in scenarios where time-restricted decision-making is crucial.

This comparative analysis clearly shows that integrating NN with MCTS provides a significant advantage, particularly in terms of achieving faster and more reliable results in

collision avoidance across various traffic densities.

Additional Comparison: MCTS (front&back) vs NN-MCTS in Low Iterations

Interestingly, for a small number of iterations (30 and 50), the plain MCTS appears to perform slightly better than NN-MCTS at these early stages of the search operation.

This phenomenon can be attributed to the operational mechanisms of NN-MCTS, which leverages NN-derived prior knowledge to guide its search towards potentially better outcomes from the outset. However, when the search process is prematurely stopped at these low iteration counts, this added bias in the search from the NN may initially lead to worse outcomes. Consequently, at these early stopping points, NN-MCTS might find itself in suboptimal positions compared to plain MCTS, which progresses more conservatively and thus may achieve better immediate rewards under these limited conditions.

Plain MCTS, lacking the acceleration granted by NN guidance, adopts a slower, more gradual exploration strategy. This method can lead to decisions that yield higher rewards in the short term compared to the more ambitious but undeveloped paths considered by NN-MCTS. This is because the NN in NN-MCTS is trained on data from simulations running 1000 iterations, where the paths to high-quality solutions are more clearly defined. Therefore, the early termination of the algorithm prevents it from reaching the more refined decisions that would come with continued iterations.

In essence, while NN-MCTS is designed to fast-track to high-quality solutions based on its pre-trained knowledge, stopping it too soon, such as at 30 or 50 iterations, means it does not fully capitalize on this advanced starting point. On the other hand, the more methodical approach of plain MCTS does not suffer as much from these early cutoffs, as its strategy is not based on projecting forward from an advanced knowledge base. This scenario underscores the importance of allowing sufficient iterations for NN-MCTS to fully leverage its NN insights, which are pivotal for achieving the long-term outcomes it is capable of under optimal operation conditions.

NN Performance and Comparisons

We now turn our attention to analyzing the performance of the plain NN without integrated search methods. Firstly, it is important to note that the NN operates independently of iterations, as previously discussed. Consequently, in the graphical representations, its performance across different vehicle flows is depicted as a straight line, indicating consistent results irrespective of the iteration count.

Observations reveal that the NN outperforms all other algorithms at low iteration counts, specifically at 30 and 50 iterations, across all vehicle flows. For instance, at a vehicle flow of 9400, the NN outperforms the MCTS (front) algorithm at all iterations except at 1000. This superior performance of the NN at low iterations is attributed to the fact that other algorithms have not yet had sufficient iterations to optimize their solutions effectively.

However, as the iteration count increases from 100 and up, the other algorithms begin to delve deeper in search of more robust solutions, leading to a noticeable increase in

collisions for the NN. This outcome aligns with expectations given that the NN employs a very greedy algorithmic approach, which follows a fixed strategy without any exploration component. Consequently, the NN tends to stick to suboptimal and less efficient solutions as it repeatedly follows the same decision paths.

Interestingly, despite the NN being trained on data generated from MCTS (front&back) simulations at 1000 iterations, it does not replicate the exact outcomes one might anticipate. This divergence is partially due to the inherent inaccuracies of the NN, which can occasionally misspredict the correct predictions. More significantly, the difference arises from the inherent randomness within the MCTS algorithm, which introduces variability in the solutions. Thus, consistently selecting the action with the highest predicted probability does not guarantee the best outcome, highlighting a limitation of relying only on a deterministic approach in scenarios that benefit from adaptive exploration and decision-making.

5.4.3 Average Speed and Delay Results

Average Speed for all Scenarios

To further our analysis, we next focus on the diagrams illustrating the speed average-iterations across all vehicle flows. Accompanying these visual representations, we provide a Table 5.10 that provides detailed information about the speed average at each sampled iteration for all algorithms, capturing their optimal performance at 1000 MCTS iterations.

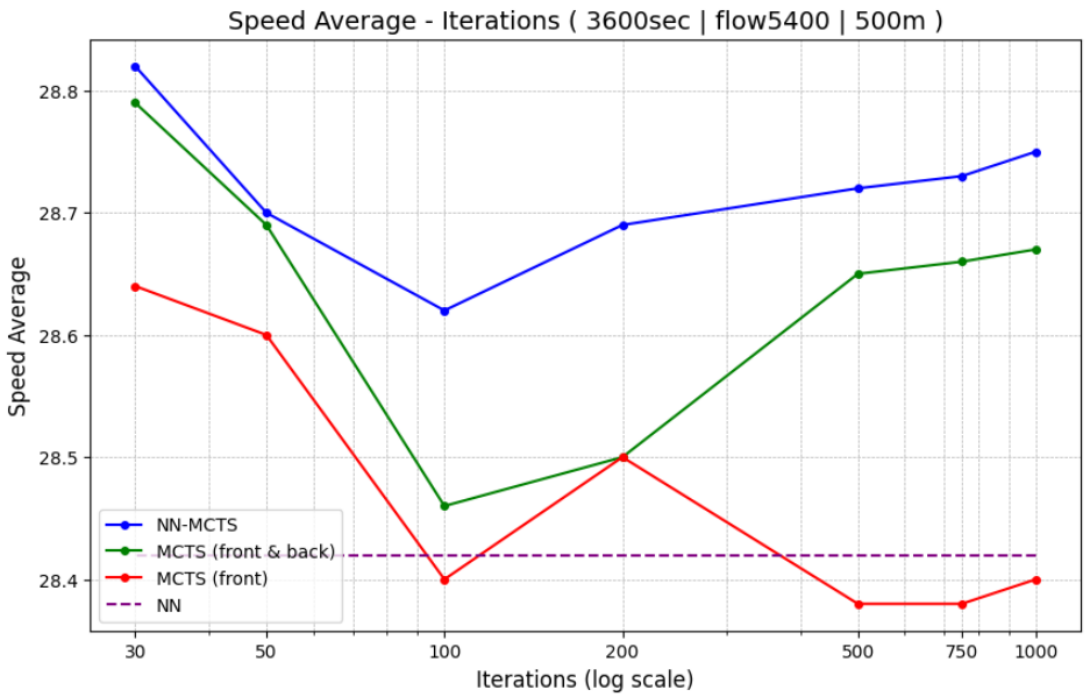


Figure 5.4: Graph of Speed Average for NN-MCTS, MCTS (front&back), MCTS (front) and NN across different Iterations for 5400 veh/h.

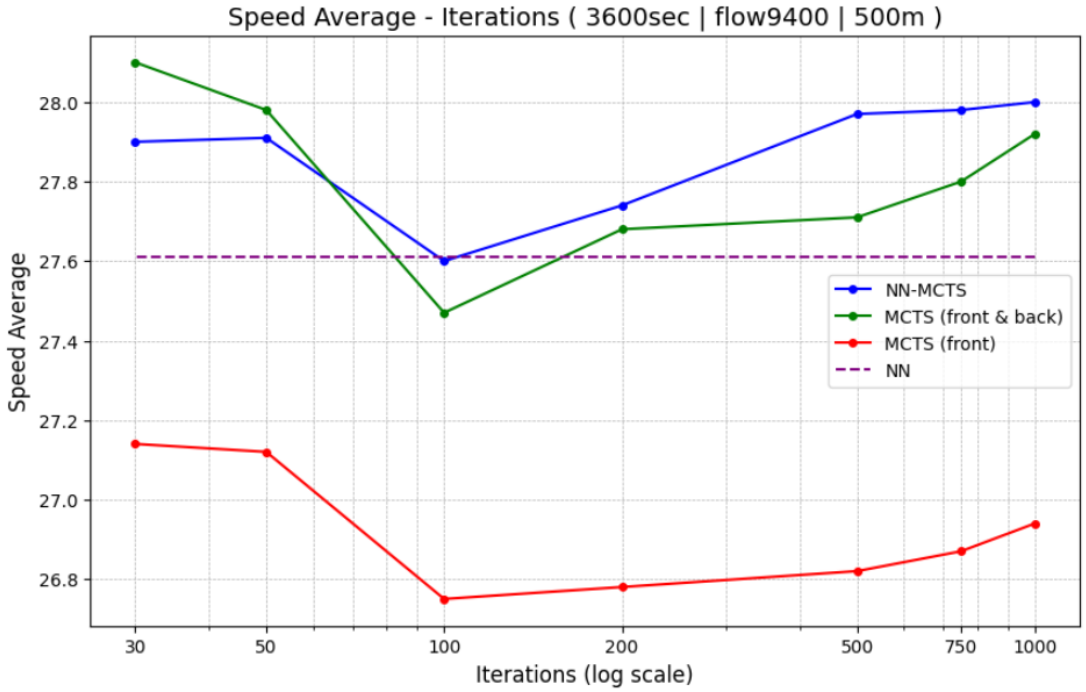


Figure 5.5: Graph of Speed Average for NN-MCTS, MCTS (front&back), MCTS (front) and NN across different Iterations for 9400 veh/h.

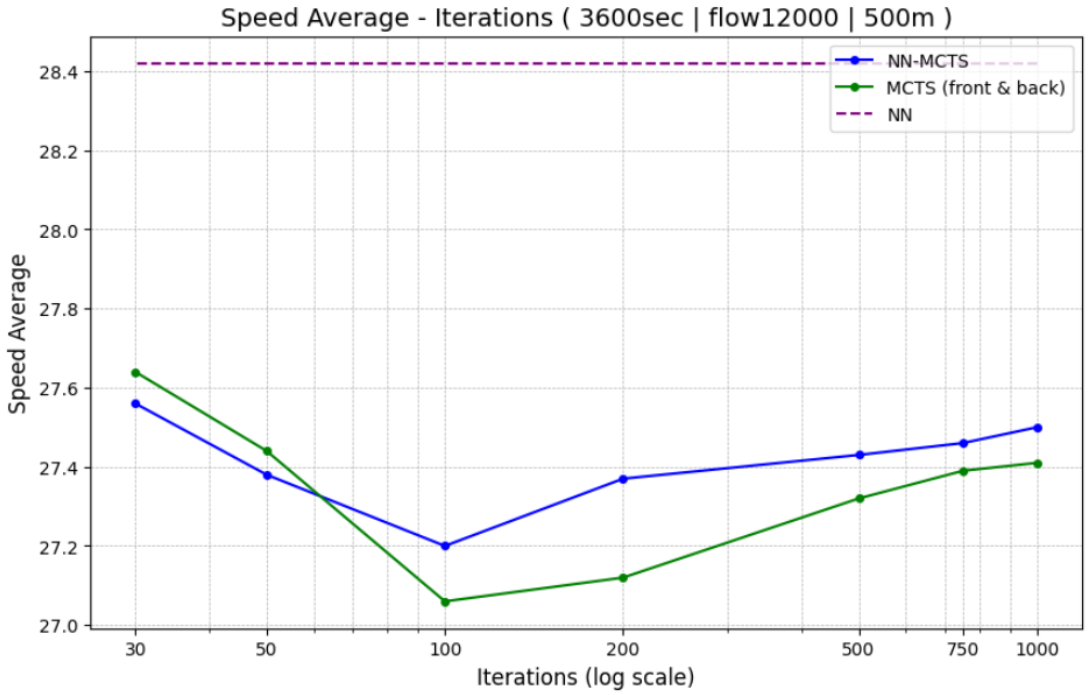


Figure 5.6: Graph of Speed Average for NN-MCTS, MCTS (front&back), MCTS (front) and NN across different Iterations for 12000 veh/h.

Veh/h	NN-Guided MCTS	MCTS (front&back)	MCTS (front)	NN
5400	28.75	28.67	28.4	28.42
9400	28	27.92	26.94	27.61
12000	27.5	27.41	-	26.98

Table 5.9: Speed Average in m/s for NN-Guided MCTS, MCTS (front&back), MCTS (front) and NN for different Vehicle Flows.

Delay Average for all *veh/h*

Subsequently, we present the diagrams for the delay average-iterations for all vehicle flows, similarly detailing metrics in a table for the delay average in the same conditions.

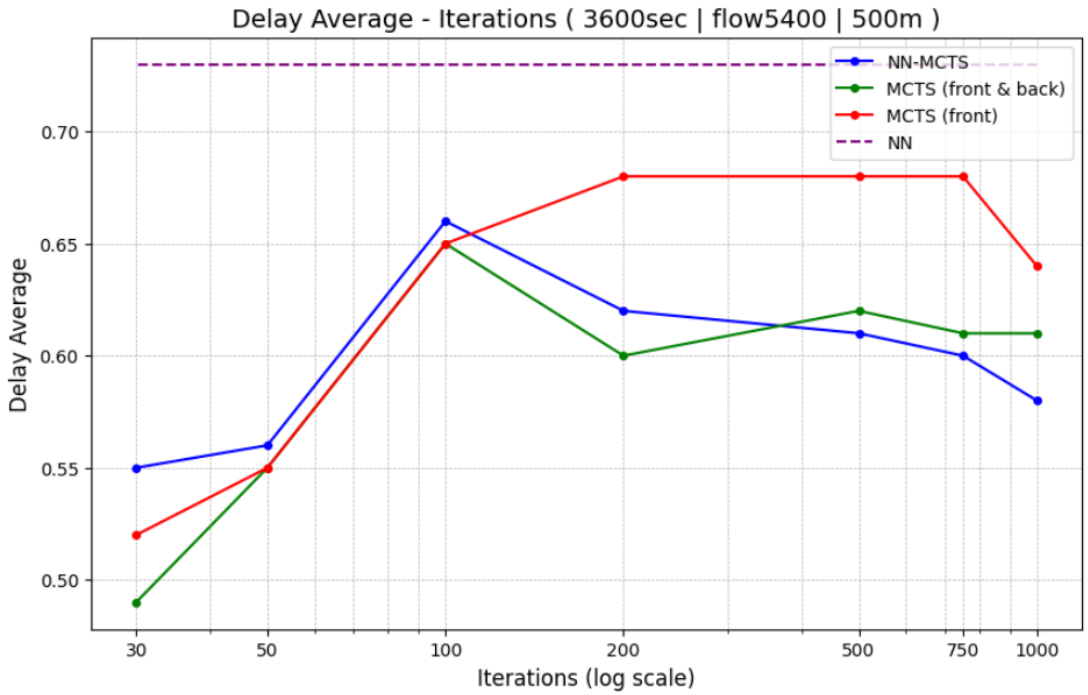


Figure 5.7: Graph of Delay Average for NN-MCTS, MCTS (front&back), MCTS (front) and NN across different Iterations for 5400 veh/h.

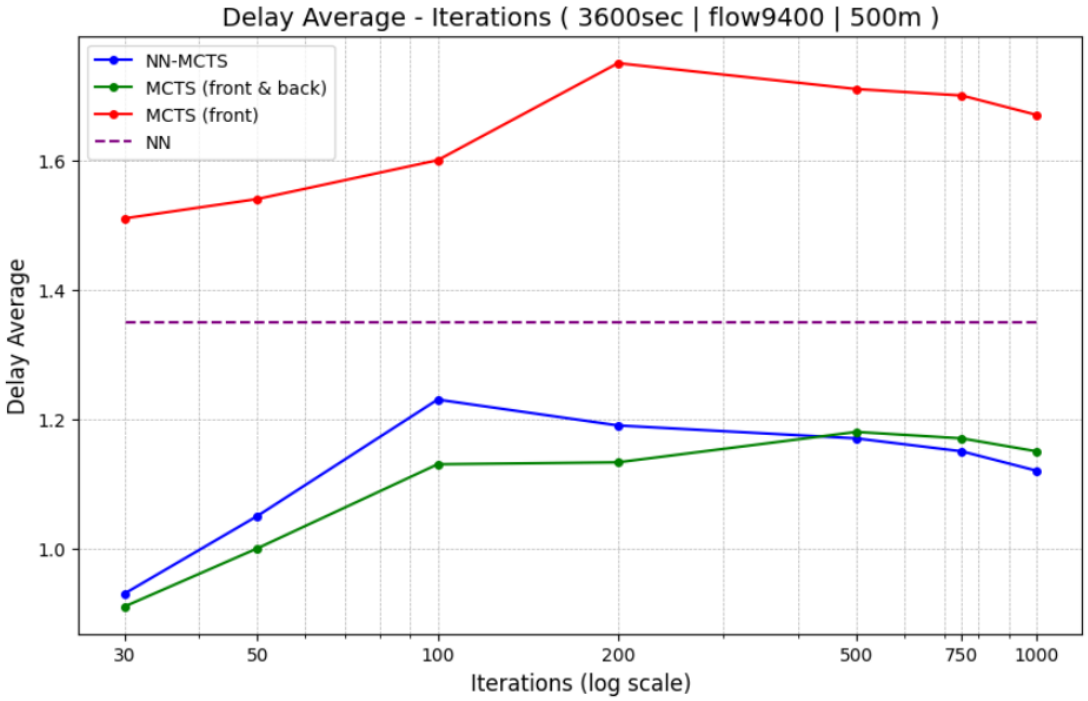


Figure 5.8: Graph of Delay Average for NN-MCTS, MCTS (front&back), MCTS (front) and NN across different Iterations for 9400 veh/h.

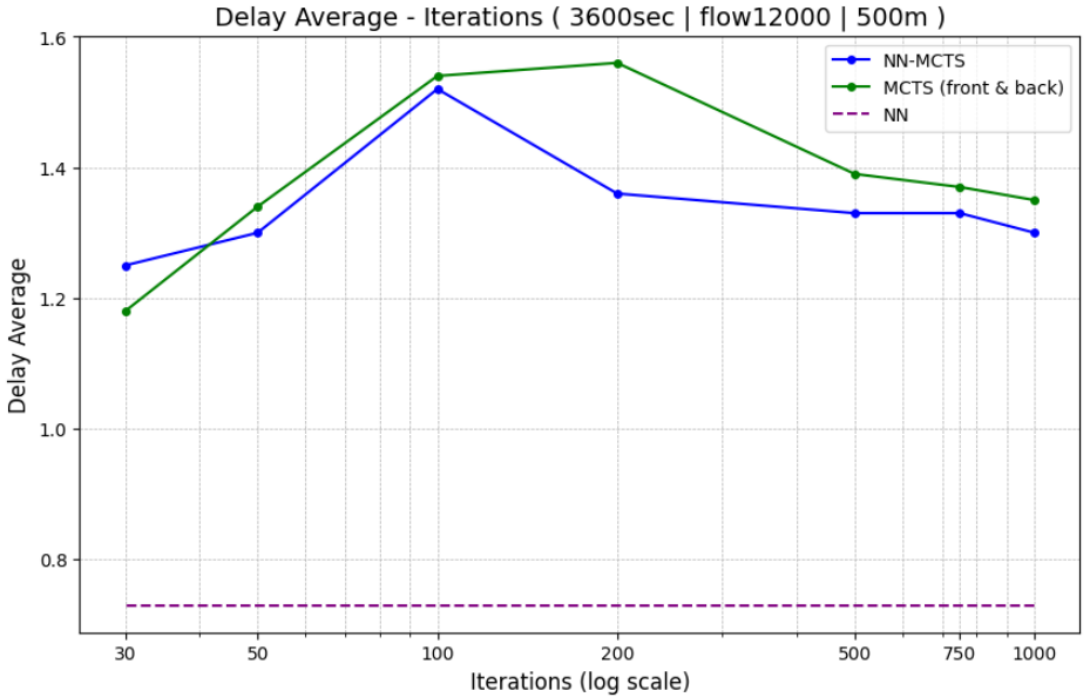


Figure 5.9: Graph of Delay Average for NN-MCTS, MCTS (front&back), MCTS (front) and NN across different Iterations for 12000 veh/h.

Veh/h	NN-Guided MCTS	MCTS (front&back)	MCTS (front)	NN
5400	0.58	0.61	0.64	0.73
9400	1.12	1.15	1.67	1.35
12000	1.3	1.35	-	1.69

Table 5.10: Delay Average in ms for NN-Guided MCTS, MCTS (front&back), MCTS (front) and NN for different Vehicle Flows.

Impact of Flow on NN-MCTS

From both the graphical data and the Tables 5.10 and 5.9, we observe that as vehicle flow increases and the traffic scenarios become more challenging, there is a worsening in both speed average and delay average. Specifically, the speed average decreases while the delay average increases. For instance, considering our top-performing algorithm, NN-MCTS, at 1000 iterations:

- **From 5400 to 9400 *veh/h*:** There is a 2% decrease in speed average from 28.75 m/s to 28 m/s, and a significant 93% increase in delay average from 0.58 seconds to 1.12 seconds.
- **From 9400 to 12000 *veh/h*:** The speed average further drops by 1.7% from 28 m/s to 27.5 m/s, and the delay average rises by an additional 16% from 1.12 seconds to 1.3 seconds.

As previously discussed in Section 5.3, these results are anticipated. The escalation in vehicle flow results in a negative impact on both speed and delay metrics. This is because, in denser traffic scenarios, vehicles must adopt more cautious and safer driving behaviors to prevent collisions. These behaviors include reducing speed and increasing waiting times before deployment to adhere to safety regulations. This cautious approach, while essential for safety, contributes to slower traffic movement and longer delays, reflecting the inherent trade-offs involved in managing increased traffic density effectively.

Comparison: MCTS (front) vs MCTS (front&back)

Continuing our analysis of the results, it becomes evident that MCTS (front) consistently underperforms in terms of speed average and delay average when compared to the enhanced plain MCTS (front&back) across all vehicle flows. This disparity is particularly noticeable, reinforcing the conclusions drawn from the collision data:

- **5400 *veh/h*:** The differences are very small but still favor MCTS (front&back). MCTS (front) records a delay average of 0.64 seconds and a speed average of 28.4 m/s. By contrast, MCTS (front&back) achieves slightly better results with a delay average of 0.61 seconds and a speed average of 28.67 m/s.
- **9400 *veh/h*:** The disadvantages of MCTS (front) become more pronounced. It achieves a speed average of only 26.94 m/s and a delay average of 1.67 seconds, significantly worse than MCTS (front&back), which posts a speed average of 27.92 m/s and a delay average of 1.15 seconds.
- **12000 *veh/h*:** As with the collision data, there are no results for MCTS (front) because it could not successfully execute in this high-density scenario.

These results further affirm the enhanced capabilities of our MCTS (front&back) algorithm. Not only does it handle collisions better, but it also improves overall traffic flow and reduces delays more effectively than MCTS (front), particularly as traffic conditions become more complex.

Comparison: MCTS (front&back) vs NN-MCTS

Continuing with the comparison between NN-MCTS and plain MCTS (front&back), we observe that across all vehicle flows and from the point where NN-MCTS stabilizes, it marginally outperforms the plain MCTS (front&back). This is evident in having slightly higher speed averages and marginally lower delay averages. These advantages are observed not only when NN-MCTS has stabilized while MCTS (front&back) has not but also when both algorithms have reached their best performance levels, indicating that NN-MCTS maintains a very slight edge over the plain MCTS. These differences provided by NN-MCTS are relatively minor, suggesting that while NN-MCTS can enhance performance, these enhancements are not substantial enough to signify a major leap over plain MCTS (front&back) in regards to the speed and delay metrics.

NN Performance and Comparisons

Turning to the plain NN, it consistently exhibits worse results than both NN-MCTS and MCTS (front&back) due to reasons previously discussed in the analysis of collisions. More specifically, as shown in Table 5.9 the plain NN records a speed average of 28.42 m/s, 27.61 m/s and 26.98 m/s at vehicle flows of 5400, 9400 and 12000 respectively. Additionally, as detailed in Table 5.10, it achieves a delay average of 0.73 seconds, 1.35 seconds and 1.69 seconds at same vehicle flows of 5400, 9400 and 12000 respectively. However, the NN still manages to surpass the performance of MCTS (front) in most scenarios, further highlighting the limitations of MCTS (front) under various traffic conditions.

Additional Comparison for Low Iterations

In closing our discussion of the results, an interesting pattern emerges at the lower iteration numbers 30 and 50. At these counts, there is a noticeable, slight improvement in both speed average and delay average across all algorithms. This is attributed to the algorithms not yet stabilizing their decision-making processes, thereby making more ”reckless” decisions that inadvertently lead to higher speeds and lower delays. These decisions, while beneficial to speed and delay metrics, are not sustainable or safe, as evidenced by the exhibited collisions. Such behaviors contain abrupt braking, lack of proper nudging and overall the vehicles show a lack of compliance to safety rules as imposed by the ellipsoid fields in the reward function, all contributing to the short-term improvement in speed and delay but at the cost of increased risk and collisions.

5.4.4 Simulation Execution Example and Vehicle Trajectories

In this section, we showcase how a simulation executes using our NN-Guided MCTS algorithm. It is important to note that this demonstration is not meant to provide insights into the algorithm's performance or detailed functionality but is primarily aimed at visualization purposes. By observing how vehicles move, accelerate, decelerate, and make

turns left and right within the simulation, we can visually confirm that our algorithm is operational and that the vehicles are following the sophisticated actions determined by our NN-Guided MCTS.

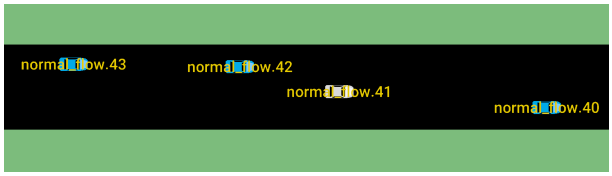


Figure 5.10: Snapshots of the simulation environment, where the marked vehicles with ID's 40, 42 and 43 have just departed in the highway-road.

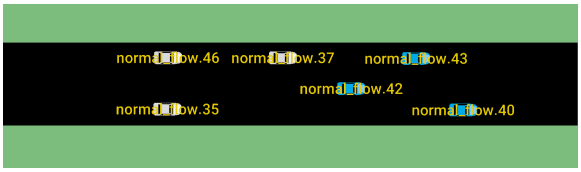


Figure 5.11: Snapshots of the simulation environment, where the marked vehicles with ID's 40, 42 and 43 are about to exit the highway-road.

As an illustrative example, we plot three different diagrams for three vehicles with different desired speeds of 29, 32 and 34 m/s. These diagrams will help clarify the trajectories that the vehicles follow, i.e. the paths along which they move and their speed in both the lateral and longitudinal axes. These speeds directly stem from the accelerations applied by the vehicles, actions which are derived from the solutions provided by the algorithm. We present these parameters as they evolve over time within the simulation, using the simulation's discrete time-steps rather than actual seconds. In each time-step, vehicles apply the updated actions, so their position and speed change in response to these executed accelerations.

Here are the diagrams that we are visualizing:

- **Lateral Position p_y (m) – Time (time-steps):** This first diagram showcase the lateral positions of our vehicles throughout the simulation. It provides a visual representation of the trajectory each of the three vehicles follows within the road confines. Imagine that the y axis represents the road width, and the x axis represents the road length. Each vehicle begins from a certain point on the road, when entering the simulation and ends up in a different point when exiting it. From this diagram, we can identify when a vehicle turns left or right, rising curves on the graph indicate a left turn, and falling curves indicate a right turn, while a flat line indicates that the vehicle is moving straight. This graph effectively maps the lateral path that each vehicle takes from its entry onto the road until it exits. We chose not to plot the longitudinal position diagram because as said in Section 4.1.1, the simulation happens in an open highway, so this diagram would just be an increasing straight line.

- **Longitudinal Speed v_x (m/s) – Time (time-steps):** This diagram showcases the longitudinal speed of the vehicles. When the graph trends upward, it indicates that the vehicle is accelerating along the x-axis, which means the vehicle is speeding up. Conversely, when the graph trends downward, it means the vehicle is decelerating, slowing down its forward movement.

- **Lateral Speed v_y (m/s) – Time (time-steps):** This diagram visualizes the lateral speeds. An upward movement in the graph indicates that the vehicle is turning

left as it accelerates laterally and a downward trajectory means the vehicle is turning right, as it decelerates laterally.

Vehicle Trajectories - Lateral Positions

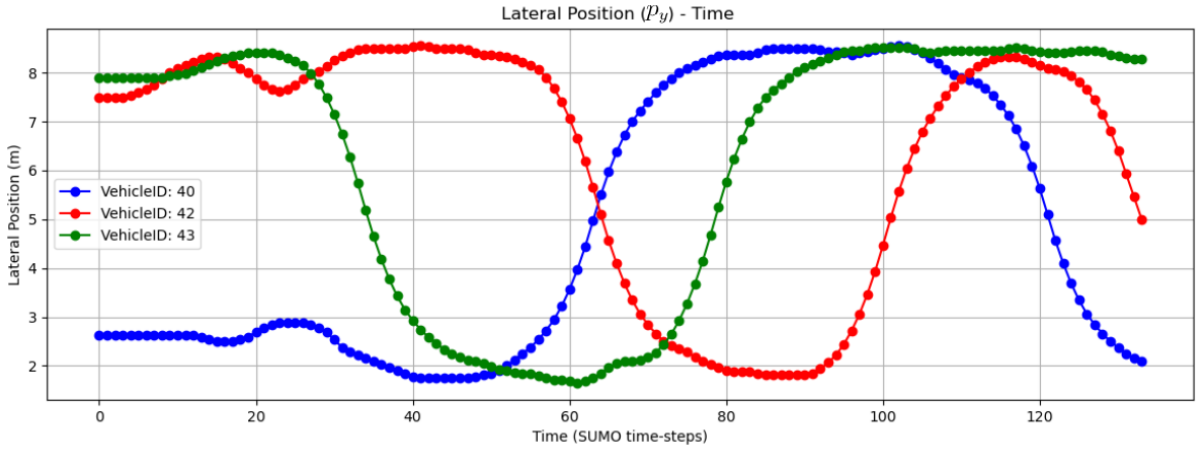


Figure 5.12: Diagram of Lateral Position p_y (m) – Time (time-steps) for vehicles with ID's 40, 42 and 43 from the SUMO simulation.

From the presented diagram, we can affirm that it corresponds with the trajectories shown in figures 5.10 and 5.11, starting and ending at the same points on the road as previously shown. The effectiveness of the NN-Guided MCTS algorithm is underscored by the evidence from the vehicles' ability to execute complex maneuvers and nudge efficiently between other vehicles and not just move forward with no actions taken. These actions are well performing also, because there are taken not just to accelerate and achieve their desired speeds but also to maintain safety, managing their distances from surrounding vehicles to avoid collisions effectively and focusing on achieving their desired speed.

Vehicle Speed - Longitudinal Speed

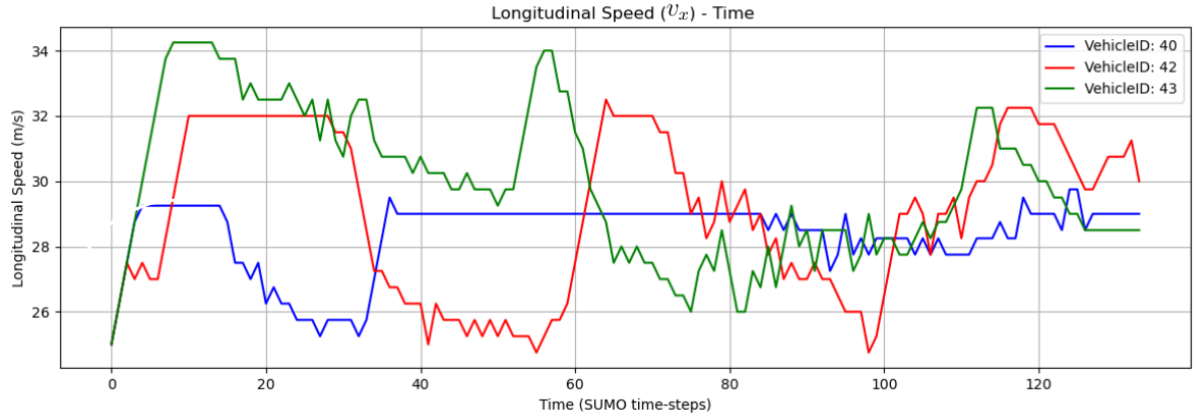


Figure 5.13: Diagram of Longitudinal Speed v_x (m/s) – Time (time-steps) for vehicles with ID's 40, 42 and 43 from the SUMO simulation.

The diagram also illustrates the variance in vehicle speeds relative to their respective desired speeds. For instance, vehicle 40, with a desired speed of 29 m/s, maintains the lowest speeds, vehicle 42 with a desired speed of 32 m/s shows balanced speeds, and vehicle 43, aiming for 34 m/s, consistently reaches the highest speeds. This indicates that each vehicle is actively trying to match or closely approximate its targeted speed. Notably, when a vehicle reaches its desired speed, it does not accelerate further, avoiding unnecessary speeds that exceed its set goal. Furthermore, there are observable shifts in speed, marked by decreases that signify braking and slowing down, either to prevent a potential collision or to execute a precise maneuver, aligning with the respective trajectories of each corresponding vehicle.

Vehicle Turing - Lateral Speed

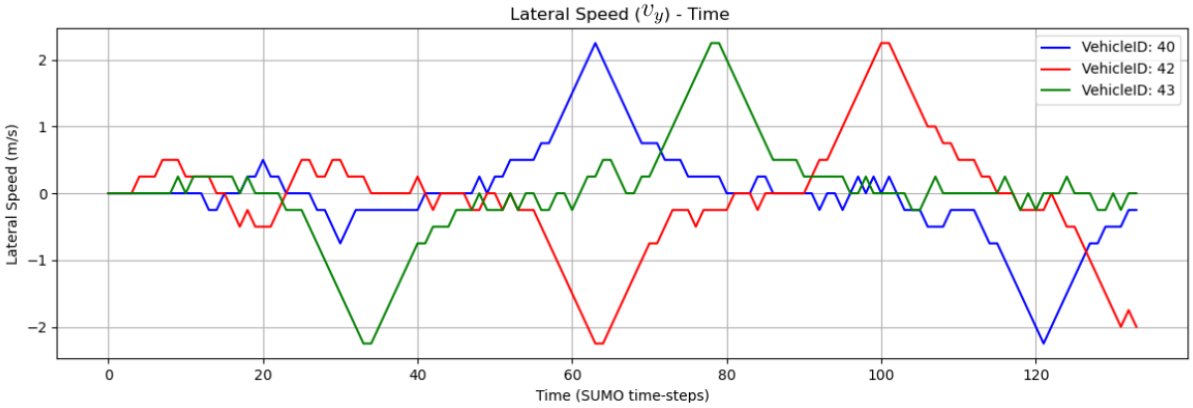


Figure 5.14: Diagram of Lateral Speed v_y (m/s) – Time (time-steps) for vehicles with ID's 40, 42 and 43 from the SUMO simulation.

Furthermore, the correlation between the lateral speed changes and the physical positioning of the vehicles during the simulation is evident. An increase in lateral speed (v_y) suggests a leftward movement, while a decrease indicates a rightward turn. This relationship is not only consistent with the expected directional changes but also with the scale of speed adjustments. Smaller variations in lateral speed lead to minor trajectory shifts, whereas larger changes in speed result in more distinct movements across the road. This detailed observation from the diagram not only validates the algorithm's operational accuracy but also its capability to adaptively respond to dynamic driving conditions, ensuring both efficiency and safety in maneuver execution.

Here we provide a link of the vehicle trajectories video that was recorded for creating the illustrations above and for capturing all the needed metrics of Figures 5.13 and 5.14: Vehicle Trajectories Video Link.

5.5 Discussion

In this section, we first summarize the impact of NN-MCTS and then discuss current limitations.

Results for Limited Iterations and Computational Resources

A fundamental point to recognize is that the primary goal of integrating NN into MCTS was not to achieve better results in an absolute sense but rather to accelerate the process of reaching optimal MCTS results. The aim was to reduce the number of iterations required, thereby saving computational time and resources. This strategic enhancement allows NN-MCTS to reach the best possible outcomes more rapidly compared to plain MCTS.

It's important to note that in scenarios where both algorithms have enough computational resources and time to exhaustively explore and build the search space tree, NN-MCTS does not produce better or substantially different solutions than plain MCTS. This equivalence arises because the NN used in NN-MCTS was trained using data from simulations run by plain MCTS. Consequently, the NN does not possess additional or superior information, rather, it facilitates quicker access to the information already available to plain MCTS, thanks to its ability to leverage prior knowledge from the training.

Thus, NN-MCTS can navigate to optimal solutions more efficiently, making it particularly advantageous in scenarios with computational constraints. In such cases, when both algorithms operate under a limited number of iterations, NN-MCTS is capable of delivering more sophisticated outcomes that plain MCTS might not achieve within the same iteration limit. This capability highlights the effectiveness of NN-MCTS in utilizing its pre-trained knowledge to expedite the decision-making process, making it a valuable approach in time-sensitive or resource-limited environments.

Simulation Execution Time and Temporal Complexity Constraints

Having analyzed the performance improvements brought about by the integration of the NN with MCTS, it is crucial to also consider the computational time each algorithm requires to execute its simulations, as this aspect holds significant practical importance.

While it has been established that NN guidance effectively reduces the number of iterations required by NN-MCTS to reach optimal results, the computational time cost associated with each algorithm must also be evaluated. Our experiments show that plain MCTS, on average, takes about 45 minutes to complete one simulation of 3600 seconds. By contrast, despite the enhancements and the increased efficiency in reaching optimal solutions more quickly as detailed in Section 5.4.2, NN-MCTS takes about 3 hours on average to execute the same simulation length, with this time overhead exclusively due to the NN prediction required for every vehicle at every time-step.

This difference in execution time means that if we compare the algorithms only based on their temporal cost per iteration, NN-MCTS does not surpass plain MCTS in terms of speed. This is a critical insight: even though NN-MCTS requires fewer iterations to reach a sophisticated decision, the longer duration of each iteration means that, from a time-cost perspective, it might be more economical to run a greater number of quicker iterations with plain MCTS than fewer, slower iterations with NN-MCTS.

This situation is mainly attributed to the technical implementation combined Python for

NN predictions with C++ for the MCTS search procedure through Pybind, thus rendering the NN integration feasible. At this point, we would need to additionally develop the proper infrastructure in order to do NN predictions natively in C++ so that to have a proper indication of the actual speed, but this is not the focus of the current study. Although the integration of NN has shown promising results in simulation efficiency, the inherent programming limitations have impacted the overall speed of execution. In the next section of our study we ponder on potential strategies and alternative approaches that could further optimize the execution time of NN-MCTS, addressing this drawback and enhancing the practical applicability of integrating NN into MCTS algorithms.

Chapter 6

Conclusions and Future Work

In this thesis, we first enhanced an existing Monte Carlo Tree Search (MCTS) algorithm, for lane-free traffic environments. Through precise fine-tuning of parameters and modifications in the state space, we crafted a more informed and efficient MCTS variant. This enhanced version has shown marked improvements over its predecessor in all key performance metrics, including safety, measured by collision frequency, average speed, and average delay. Notably, it has proven capable of operating effectively in complex traffic scenarios with dense vehicle flow, a challenge that the initial MCTS approach could not efficiently tackle.

Building upon this, the main part of our approach involved the integration of a NN into the selection phase of MCTS. The NN was trained offline through self-play simulations, using the data generated by our refined MCTS. Despite initial challenges relating to the algorithm's inherent randomness and its effect on accuracy, we addressed these issues through model calibration and the utilization of reliability diagrams.

The integration process involved employing the NN as a classifier to evaluate MCTS states and predict outcomes across all possible actions. This predictive capability, denoted as $P(s, a)$, was then applied as prior knowledge within the selection phase, substituting the traditional Upper Confidence Trees (UCT) approach with Predictive UCT. The results were remarkably positive, showcasing that the NN-guided MCTS could expedite the search for optimal solutions, reducing the need for extensive iterations and computational complexity. The NN-guided algorithm demonstrated an enhanced ability to explore the depths of the search tree space rapidly and efficiently, reaffirming the effectiveness of NN guidance in accelerating the decision-making process in MCTS.

However, our success is tempered by challenges related to the execution speed of the algorithm. Despite concerted efforts to streamline and accelerate the prediction process, including techniques like batch NN processing with batch predictions, the NN-guided MCTS remained slower in execution compared to the baseline MCTS when considering the quality of results relative to the actual execution time taken.

6.1 Future Work

As we conclude, we acknowledge that while we have taken significant progress in enhancing MCTS for lane-free autonomous driving, there remains much to be explored and refined. Future work can address this limitation in order to fully realize the potential of an NN-integrated MCTS algorithm that not only outperforms in decision-making efficacy but also in computational efficiency. All these ideas for future work that are going to follow.

Technical developments for Native NN Predictions

For the future enhancement of the NN-Guided MCTS algorithm. A fundamental step would be to homogenize the programming languages used for the algorithm. The current architecture operates with the NN written in Python and MCTS in C++. As identified in Section 4.3.6, the main bottleneck is the time-intensive process required to access Python scripts from C++ during runtime, rather than the core functionality of the NN itself. As a result, by eliminating this bottleneck the algorithm will speed up all its calculations in significant way.

NN-Guided Expansion

Looking ahead, to leverage the NN's capabilities beyond the selection phase of MCTS and into other areas, we see potential in implementing a guided expansion mechanism within the MCTS. Presently, the algorithm considers a set number of 15 potential actions from a state, which, while manageable, limits the precision and detail of the decisions. Expanding the action space could enable the algorithm to make more sophisticated decisions. However, it would proportionally increase the search tree size, making the current search strategy inefficient.

To resolve the increased action space with manageable search complexity, a guided expansion process can assist in that endeavour. Drawing inspiration from [28] that evaluates the application of neural MCTS methods in various domains beyond gaming we suggest following this guided expansion method as explained next. By employing the NN's learned policy, we could selectively expand only those child nodes that the NN identifies as promising, effectively pruning the tree of less beneficial paths and focusing computational efforts on the most valuable decisions. This process would allow the algorithm to benefit from a richer decision set while keeping the search space optimally constrained.

Such a guided expansion approach is applied by the notion of sampling possible actions at a leaf node, as per the NN's policy guidance. In the context of MCTS, this would mean a shift from expanding nodes indiscriminately to a more strategic expansion, prioritizing children with higher predicted values of success. In practice, this method would balance between the breadth and depth of exploration, and adapt the tree growth to focus mainly on high quality solutions paths. The exact impact of neurally guided expansion will vary from application to application, its general potential is demonstrated in [84]. Additionally, here are some examples where neurally guided expansion is used [85, 86, 87].

NN-Guided Rollout for Observed Vehicles

NN guidance could extend into the simulation phase of MCTS as well. In its current state, as highlighted in Section 4.2.5, the algorithm operates under the assumption that neighboring vehicles maintain a steady course without any applied acceleration to them (neither lateral or longitudinal), which does not reflect the dynamic nature of real-world driving scenarios where neighboring vehicles will indeed take various actions.

Taking further cues from the same paper [28], we recommend adopting the neural guided evaluation methods as detailed in the following text. To enhance the realism and predictive accuracy of these simulations, the NN's learned policy could be instrumental. By applying NN guidance, we can simulate neighbor vehicle actions that are not only reactive to the evolving traffic scenario but also based on informed decisions derived from the NN's extensive training. This implementation would result in simulations that more accurately represent realistic and well decided actions of neighboring vehicles, thereby producing outcomes and rewards for each state that are closer to to the actual movement of observed vehicles.

Such guided simulations will undoubtedly result in a more robust and reliable MCTS framework, where each simulated trajectory is not a product of mere assumption, but a reflection of strategic actions guided by the NN.

To achieve this, we could employ a hybrid approach to node evaluation that balances solution quality and computational time. For nodes closer to the tree's root, direct state-value predictions using the NN could be used, while deeper nodes might still utilize randomized roll-outs due to their reduced computational demand. Here we provide some examples that use neural guidance in evaluation with this hybrid approach [88, 89, 90].

Guidance Network Training and Policy Improvement by MCTS

A potential enhancement for the training methodology of the NN used in guiding the MCTS is shifting from the current offline supervised learning approach, as outlined in Section 4.3.2, to an online learning paradigm. This alternative method involves continuous policy improvement via MCTS during the actual tree search process. It is a dynamic and iterative procedure where the results from the MCTS search inform and refine the NN's training in real-time. These methods are also detailed in [28].

The principle behind this method aligns with the concept of policy improvement by MCTS, a technique that leverages the strengths of both MCTS and the NN to progressively develop a more powerful policy. This method was applied in AlphaZero [91], where the NN was used to guide MCTS and then it's learned policy was improved iteratively by MCTS search results. The iterative cycle works as follows: The NN guides the tree search by predicting potential actions, MCTS then refines these predictions through simulation, and the outcomes of the MCTS simulations serve as fresh, enriched data for further NN training. The improved NN, in turn, provides a stronger guiding hand for subsequent tree searches.

This approach transforms the NN into an active learner, continuously adapting and evolv-

ing based on the latest insights gathered from the tree search. It creates a feedback loop where the NN not only influences the decision-making process but also learns from the quality of decisions it helps to make. Over time, this can lead to the development of a deeply layered policy that captures sophisticated decision patterns which might be overlooked in a purely offline training program. Here are some example uses that combine supervised learning pre-training offline and online policy improvement by MCTS [92, 93, 94, 95].

Summing Up Future Work

By implementing the proposed enhancements and future work strategies, the NN-Guided MCTS is geared to achieve not only faster navigation through the search tree and quicker solution generation but also significantly more refined outcomes. These advancements will contribute to a more sophisticated and realistic decision-making process, enabling the algorithm to produce results that are not only efficient but also of higher quality and closer to real-world applicability. This evolution in the NN-Guided MCTS will reinforce its foundation, making it an even more powerful tool for autonomous lane-free driving systems.

References

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez Liebana, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, Mar 2012. [Online]. Available: 10.1109/TCIAIG.2012.2186810

[2] A. Choudhary. (2023, Dec) Introduction to monte carlo tree search: The game-changing algorithm behind deepmind's alphago. 11 min read. [Online]. Available: https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/

[3] P. Baheti. (2021, Oct) Supervised and unsupervised learning [differences & examples].

[4] R. Pramoditha. (2021, Dec) The concept of artificial neurons (perceptrons) in neural networks. [Online]. Available: https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc

[5] ——. (2022, Feb) Overview of a neural network's learning process. Published in Data Science 365, 5 min read. [Online]. Available: https://medium.com/data-science-365/overview-of-a-neural-networks-learning-process-61690a502fa

[6] P. Mehta. (2020, Jun) Am i sure or unsure?—uncertainty talks with a neural network. 5 min read. [Online]. Available: https://towardsdatascience.com/am-i-sure-or-unsure-talks-with-a-neural-network-fc0e14d31373

[7] D. Shulga. (2018, Mar) A (very) friendly introduction to confidence intervals. 7 min read. [Online]. Available: https://towardsdatascience.com/a-very-friendly-introduction-to-confidence-intervals-9add126e714

[8] A. Karalakou, D. Troullinos, G. Chalkiadakis, and M. Papageorgiou, "Deep reinforcement learning reward function design for autonomous driving in lane-free traffic," *Systems*, vol. 11, no. 3, 2023. [Online]. Available: https://www.mdpi.com/2079-8954/11/3/134

[9] NHTSA, "Critical reasons for crashes investigated in the national motor vehicle crash causation survey," National Highway Traffic Safety Administration, Traffic Safety Facts, Feb 2015. [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115

[10] R. Bishop, "A survey of intelligent vehicle applications worldwide," in *Proceedings of the IEEE Intelligent Vehicles Symposium*, 2000, pp. 25–30. [Online]. Available: https://www.semanticscholar.org/paper/A-survey-of-intelligent-vehicleapplications-Bishop/b80f4f07fa0503fc980a746c7d8c05d9abf9e52c

[11] M. Papageorgiou, K.-S. Mountakis, I. Karafyllis, and I. Papamichail, "Lane-free artificial-fluid concept for vehicular traffic," *arXiv preprint arXiv:1905.11642*, May 2019. [Online]. Available: https://arxiv.org/abs/1905.11642

[12] Z. Tahiri, K. Jetto, M. Bouadi, A. Benyoussef, and A. Kenz, "The effect of anisotropy on the traffic flow behavior: Investigation of the correlation created by a single node on two-lane roads," *International Journal of Modern Physics C*, vol. 31, Jan 2020. [Online]. Available: https://doi.org/10.1142/S0129183120500606

[13] I. Karafyllis, D. Theodosis, and M. Papageorgiou, "Two-dimensional cruise control of autonomous vehicles on lane-free roads," *arXiv preprint arXiv:2103.12205*, Mar 2021. [Online]. Available: https://arxiv.org/abs/2103.12205

[14] D. Troullinos, G. Chalkiadakis, I. Papamichail, and M. Papageorgiou, "Collaborative multiagent decision making for lane-free autonomous driving," in *AAMAS '21*. Virtual Event, United Kingdom: International Foundation for Autonomous Agents and Multiagent Systems, 2021, pp. 1335–1343. [Online]. Available: https://www.ifaamas.org/Proceedings/aamas2021/pdfs/p1335.pdf

[15] V. K. Yanumula, P. Typaldos, D. Troullinos, M. Malekzadeh, I. Papamichail, and M. Papageorgiou, "Optimal trajectory planning for connected and automated vehicles in lane-free traffic with vehicle nudging," *arXiv preprint arXiv:2207.09670*, Jul 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9328636

[16] I. Chrysomallis, D. Troullinos, G. Chalkiadakis, I. Papamichail, and M. Papageorgiou, "Deep reinforcement learning with implicit imitation for lane-free autonomous driving," in *FAIA*, Sep 2023. [Online]. Available: https://doi.org/10.3233/FAIA230304

[17] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, no. 1, pp. 237–285, 1996. [Online]. Available: 10.1613/jair.301

[18] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2016.

[19] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959. [Online]. Available: 10.1147/rd.33.0210

[20] Q. Liu and Y. Wu, "Supervised learning," in *Encyclopedia of the Sciences of Learning*, N. M. Seel, Ed. Boston, MA: Springer US, 2012, pp. 3243–3245. [Online]. Available: https://doi.org/10.1007/978-1-4419-1428-6_451

[21] S. C. Kleene, "Representation of events in nerve nets and finite automata," 1951.

[22] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[23] C. Xu and J. McAuley, "A survey on dynamic neural networks for natural language processing," *arXiv preprint arXiv:2202.07101*, 2022, comment: EACL 2023 Findings. [Online]. Available: https://doi.org/10.48550/arXiv.2202.07101

[24] J. Chai, H. Zeng, A. Li, and E. W. Ngai, "Deep learning in computer vision: A critical review of emerging techniques and application scenarios," *Machine Learning with Applications*, vol. 1, no. 100134, 2021, under a Creative Commons license. [Online]. Available: https://doi.org/10.1016/j.mlwa.2021.100134

[25] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, "Monte carlo tree search: a review of recent modifications and applications," *Artificial Intelligence Review*, Jul 2022, published online. [Online]. Available: https://link.springer.com/article/10.1007/s10462-022-10228-y

[26] H. Nakhost and M. Müller, "Monte-carlo exploration for deterministic planning," in *IJCAI*, vol. 9, 2009, pp. 1766–1771.

[27] M. Schoenauer, S. Gelly, L. Kocsis *et al.*, "The grand challenge of computer go: Monte carlo tree search and extensions," *Communications of the ACM*, vol. 55, no. 3, pp. 106–113, Mar 2012. [Online]. Available: https://hal.inria.fr/hal-00695370v3/document

[28] M. Kemmerling, D. Lütticke, and R. H. Schmitt, "Beyond games: A systematic review of neural monte carlo tree search applications," *arXiv*, vol. abs/2303.08060, 2023, available as arXiv preprint abs/2303.08060. [Online]. Available: https://arxiv.org/abs/2303.08060

[29] T. Ha, K. Cho, G. Cha, K. Lee, and S. Oh, "Vehicle control with prediction model based monte-carlo tree search," in *17th International Conference on Ubiquitous Robots (UR)*, 2020, pp. 303–308. [Online]. Available: https://doi.org/10.1109/UR49135.2020.9144958

[30] J. Chen, C. Zhang, J. Luo, J. Xie, and Y. Wan, "Driving maneuvers prediction based autonomous driving control by deep monte carlo tree search," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7146–7158, 2020. [Online]. Available: https://doi.org/10.1109/TVT.2020.2991584

[31] L. Lei, R. Luo, R. Zheng, J. Wang, J. Zhang, C. Qiu, L. Ma, L. Jin, P. Zhang, and J. Chen, "Kb-tree: Learnable and continuous monte-carlo tree search for autonomous driving planning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Prague, Czech Republic: IEEE Press, 2021, pp. 4493–4500. [Online]. Available: https://doi.org/10.1109/IROS51168.2021.9636442

[32] N. Catenacci Volpi, Y. Wu, and D. Ognibene, "Towards event-based mcts for autonomous cars," in *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, 2017, pp. 420–427.

[33] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, pp. 679–684, 1957.

[34] P. Giankoulidis, "Monte carlo tree search for autonomous driving in lane-free traffic settings," Diploma Work, School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece, 2023, https://doi.org/10.26233/heallink.tuc.96157.

[35] C. Gao, R. Hayward, and M. Muller, "Move prediction using deep convolutional neural networks in hex," *IEEE Transactions on Games*, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8226781

[36] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," *arXiv preprint arXiv:1706.04599*, 2017. [Online]. Available: https://arxiv.org/abs/1706.04599

[37] A. Johansen, "Monte carlo methods," in *International Encyclopedia of Education*, 3rd ed. Elsevier, 2010, accessed: date-of-access. [Online]. Available: https://www.sciencedirect.com/topics/agricultural-and-biological-sciences/monte-carlo-method

[38] I. Dimov, *Monte Carlo Methods for Applied Scientists*. World Scientific, Jan 2008. [Online]. Available: https://www.researchgate.net/publication/267115359_Monte_Carlo_Methods_for_Applied_Scientists

[39] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957. [Online]. Available: http://www.jstor.org/stable/24900506

[40] M. T. J. Spaan, "Partially observable markov decision processes," in *Reinforcement Learning: State-of-the-Art*, M. Wiering and M. van Otterlo, Eds. Springer Berlin Heidelberg, 2012, pp. 387–414. [Online]. Available: https://doi.org/10.1007/978-3-642-27645-3_12

[41] D. P. Bertsekas, *Dynamic Programming and Optimal Control*. Athena Scientific Belmont, 2005, vol. 1.

[42] C. Sammut and G. I. Webb, *Encyclopedia of Machine Learning*. Springer US, 2010. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_71

[43] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293.

[44] G. Hinton and T. J. Sejnowski, *Unsupervised Learning: Foundations of Neural Computation*. The MIT Press, May 1999. [Online]. Available: https://doi.org/10.7551/mitpress/7011.001.0001

[45] C. Castillo-Botón, D. Casillas-Pérez, C. Casanova-Mateo, S. Ghimire, E. Cerro-Prada, P. Gutierrez, R. Deo, and S. Salcedo-Sanz, "Machine learning regression and classification methods for fog events prediction," *Atmospheric Research*, vol. 272, p. 106157, Jul 2022.

[46] M. A. Nielsen, *Neural Networks and Deep Learning*. San Francisco, CA, USA: Determination Press, 2015, vol. 25.

[47] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[48] W. Mcculloch and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 127–147, 1943.

[49] G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," *IEEE Potentials*, vol. 13, no. 4, pp. 27–31, 1994. [Online]. Available: 10.1109/45.329294

[50] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *CoRR*, vol. abs/1511.08458, 2015. [Online]. Available: http://arxiv.org/abs/1511.08458

[51] L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications*. CRC Press, 1999.

[52] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *International Journal of Engineering Applied Sciences and Technology*, vol. 04, no. May, pp. 310–316, 2020.

[53] A. F. Agarap, "Deep learning using rectified linear units (relu)," *CoRR*, vol. abs/1803.08375, 2018. [Online]. Available: http://arxiv.org/abs/1803.08375

[54] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *arXiv preprint arXiv:1702.05659*, 2017. [Online]. Available: https://arxiv.org/abs/1702.05659v1

[55] T. Chai and R. Draxler, "Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature," *Geoscientific Model Development*, vol. 7, pp. 1247–1250, 2014. [Online]. Available: 10.5194/gmd-7-1247-2014

[56] Z. Zhang and M. R. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," *arXiv preprint arXiv:1805.07836*, May 2018. [Online]. Available: https://arxiv.org/abs/1805.07836

[57] J. Luo, H. Qiao, and B. Zhang, "Learning with smooth hinge losses," *arXiv preprint arXiv:2103.00233*, 2021. [Online]. Available: https://arxiv.org/abs/2103.00233

[58] S. Damadi, G. Moharrer, and M. Cham, "The backpropagation algorithm for a math student," *arXiv preprint arXiv:2301.09977*, 2023. [Online]. Available: https://arxiv.org/abs/2301.09977

[59] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[60] H. E. Robbins, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 1951, online. Available: https://api.semanticscholar.org/CorpusID:16945044.

[61] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, Dec 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[62] T. Kavzoglu, "Increasing the accuracy of neural network classification using refined training data," *ResearchGate*, 2009. [Online]. Available: https://www.researchgate.net/publication/223565557_Increasing_the_accuracy_of_neural_network_classification_using_refined_training_data

[63] J. Moon, J. Kim, Y. Shin, and S. Hwang, "Confidence-aware learning for deep neural networks," *arXiv preprint arXiv:2007.01458*, 2020. [Online]. Available: https://arxiv.org/pdf/2007.01458.pdf

[64] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, and R. Fonseca, "Alphax: exploring neural architectures with deep neural networks and monte carlo tree search," *arXiv preprint arXiv:1903.11059*, 2019. [Online]. Available: https://arxiv.org/abs/1903.11059

[65] B. Paden, M. Cáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, 2016. [Online]. Available: 10.1109/TIV.2016.2578706

[66] W. Hager, C. Dardy, and A. Rao, "An hp-adaptive pseudospectral method for solving optimal control problems," *Optimal Control Applications and Methods*, vol. 32, pp. 476–502, 2011.

[67] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in *Proceedings of the International Conference on Robotics and Automation*, vol. 3. IEEE, 1997, pp. 2719–2726.

[68] J. Ziegler, P. Bender, M. Schreiber, H. Lategahn, T. Strauss, C. Stiller, T. Dang, U. Franke, N. Appenrodt, C. G. Keller *et al.*, "Making bertha drive—an autonomous journey on a historic route," *IEEE Intelligent Transportation Systems Magazine*, vol. 6, no. 2, pp. 8–20, 2014.

[69] S. La Valle, "Rapidly-exploring random trees: a new tool for path planning," 1998, technical Presentation, CS Department, Iowa State University.

[70] X. Tang, K. Yang, H. Wang, W. Yu, X. Yang, T. Liu, and J. Li, "Driving environment uncertainty-aware motion planning for autonomous vehicles," *Chinese Journal of Mechanical Engineering*, vol. 35, Sep 2022, cite this article. [Online]. Available: https://cjme.springeropen.com/articles/10.1186/s10033-022-00790-5

[71] M.-A. Papilaris and G. Chalkiadakis, "Markov chain monte carlo for effective personalized recommendations," in *Proceedings of the 16th European Conference on Multi-Agent Systems (EUMAS-2018)*, Bergen, Norway, December 2018. [Online]. Available: https://www.intelligence.tuc.gr/~gehalk/Papers/pcRecMCMC-eumas18.pdf

[72] A. Theodoridis and G. Chalkiadakis, "Monte carlo tree search for the game of diplomacy," in *Proceedings of the 11th Hellenic Conference on Artificial Intelligence (SETN 2020)*, Athens, Greece, September 2020. [Online]. Available: https://www.intelligence.tuc.gr/~gehalk/Papers/SETN2020-atgc-MCTS-Diplomacy.pdf

[73] E. Karamalegos, "Monte carlo tree search in the "settlers of catan" strategy game," Diploma Work, School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece, 2016. [Online]. Available: http://purl.tuc.gr/dl/dias/1CCC4417-B1AA-4C2E-B33B-0F6BEFDED36E

[74] A. Castellini, G. Chalkiadakis, and A. Farinelli, "Influence of state-variable constraints on partially observable monte carlo planning," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI-2019)*, Macao, China, August 2019. [Online]. Available: https://www.intelligence.tuc.gr/~gehalk/Papers/ijcai2019_POMCP-TRP_cca.pdf

[75] D. Troullinos, G. Chalkiadakis, D. Manolis, I. Papamichail, and M. Papageorgiou, "Extending sumo for lane-free microscopic simulation of connected and automated vehicles," in *SUMO Conference Proceedings*, vol. 3, Sep 2022, pp. 95–103. [Online]. Available: https://doi.org/10.52825/scp.v3i.110

[76] F. J. J. Joseph, S. Nonsiri, and A. Monsakul, "Keras and tensorflow: A hands-on experience," in *Advanced Deep Learning for Engineers and Scientists*. Thai-Nichi Institute of Technology: Springer, Jul 2021, pp. 85–111. [Online]. Available: https://doi.org/10.1007/978-3-030-66519-7_4

[77] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, "Gpgpu processing in cuda architecture," *Advanced Computing: An International Journal*, vol. 3, no. 1, p. 105, Jan 2012. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1202/1202.4347.pdf

[78] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, the paper describes the development of a library of efficient deep learning primitives, akin to BLAS for parallel processors. Version 3. [Online]. Available: https://arxiv.org/abs/1410.0759

[79] W. McKinney, "pandas: a foundational python library for data analysis and statistics," *ResearchGate*, Jan 2011. [Online]. Available: https://www.researchgate.net/publication/265194455_pandas_a_Foundational_Python_Library_for_Data_Analysis_and_Statistics

[80] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, A. Müller, J. Nothman, G. Louppe *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011, updated authors list and URLs. Version 4. [Online]. Available: https://doi.org/10.48550/arXiv.1201.0490

[81] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," Preferred Networks, Inc., Jul 2019, preprint compiled July 26, 2019.

[82] P. E. Barrett, J. Hunter, J. Miller, and J.-C. Hsu, "matplotlib – a portable python plotting package," in *Proceedings of the Astronomical Data Analysis Software and Systems XIV ASP Conference Series*, vol. 347. George Washington University, Dec 2005. [Online]. Available: https://www.researchgate.net/publication/234238535_matplotlib_--_A_Portable_Python_Plotting_Package

[83] S. Rodriguez and P. Cardiff, "A general approach for running python codes in openfoam using an embedded pybind11 python interpreter," *arXiv preprint arXiv:2203.16394*, 2022, preprint submitted to OpenFOAM Journal. [Online]. Available: https://arxiv.org/abs/2203.16394

[84] B. Riviere, W. Honig, M. Anderson, and S.-J. Chung, "Neural tree expansion for multi-robot planning in non-cooperative environments," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 6868–6875, 2021. [Online]. Available: https://arxiv.org/abs/2104.09705

[85] D. Erikawa, N. Yasuo, and M. Sekijima, "Mermaid: An open source automated hit-to-lead method based on deep reinforcement learning," *Journal of Cheminformatics*, vol. 13, no. 1, pp. 1–10, 2021. [Online]. Available: https://jcheminf.biomedcentral.com/articles/10.1186/s13321-021-00572-6

[86] S. Genheden, A. Thakkar, V. Chadimova, J.-L. Reymond, O. Engkvist, and E. Bjerrum, "Aizynthfinder: A fast, robust and flexible open-source software for retrosynthetic planning," *Journal of Cheminformatics*, vol. 12, no. 1, pp. 1–9, 2020. [Online]. Available: https://jcheminf.biomedcentral.com/articles/10.1186/s13321-020-00472-1

[87] M. H. S. Segler, M. Preuss, and M. P. Waller, "Planning chemical syntheses with deep neural networks and symbolic ai," *Nature*, vol. 555, no. 7698, pp. 604–610, 2018. [Online]. Available: https://www.nature.com/articles/nature25978

[88] H. Deng, X. Yuan, Y. Tian, and J. Hu, "Neural-augmented two-stage monte carlo tree search with over-sampling for protein folding in hp model," *IEEJ Transactions on Electrical and Electronic Engineering*, vol. 17, no. 5, pp. 685–694, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/tee.23556

[89] S. Song, H. Chen, H. Sun, and M. Liu, "Data efficient reinforcement learning for integrated lateral planning and control in automated parking system," *Sensors*, vol. 20, no. 24, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/24/7297

[90] B. Sridharan, S. Mehta, Y. Pathak, and U. D. Priyakumar, "Deep reinforcement learning for molecular inverse problem of nuclear magnetic resonance spectra to molecular structure," *The Journal of Physical Chemistry Letters*, vol. 13, pp. 4924–4933, 2022. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/35635003/

[91] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/30523106/

[92] Y.-Q. Chen, Y. Chen, C.-K. Lee, S. Zhang, and C.-Y. Hsieh, "Optimizing quantum annealing schedules with monte carlo tree search enhanced with neural networks," *Nature Machine Intelligence*, vol. 4, no. 3, pp. 269–278, 2022. [Online]. Available: https://arxiv.org/abs/2004.02836

[93] A. Rinciog, C. Mieth, P. M. Scheikl, and A. Meyer, "Sheet-metal production scheduling using alphago zero," in *Proceedings of the Conference on Production Systems and Logistics: CPSL 2020*. Hannover, Germany: Leibniz Universität Hannover, 2020. [Online]. Available: https://www.repo.uni-hannover.de/handle/123456789/9732

[94] R. Sun and Y. Liu, "Hybrid reinforcement learning for power transmission network self-healing considering wind power," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–11, 2021. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/34968180/

[95] X. Wang, Y. Qian, H. Gao, C. W. Coley, Y. Mo, R. Barzilay, and K. F. Jensen, "Towards efficient discovery of green synthetic pathways with monte carlo tree search and reinforcement learning," *Chemical Science*, vol. 11, no. 40, pp. 10 959–10 972, 2020. [Online]. Available: https://pubs.rsc.org/en/content/articlelanding/2020/sc/d0sc04184j