# Combining Reinforcement Learning and Search for Cooperative Trajectory Planning

Master Thesis

by


Timo Klein

Degree Course: Economics Engineering M.Sc.

Matriculation Number: 2122521

Institute of Applied Informatics and Formal Description Methods (AIFB)

KIT Department of Economics and Management


| | |
|---|---|
| Advisor: | Prof. Dr.-Ing. Johann Marius Zöllner |
| Second Advisor: | Prof. Dr. Andreas Oberweis |
| Supervisor: | M.Sc. Karl Kurzer |
| Submitted: | August 14, 2021 |

# Abstract

How can the sample efficiency of decision making in autonomous driving be improved? This thesis provides an answer by combining Reinforcement Learning (RL) with Monte Carlo tree search (MCTS). The proposed algorithm plans cooperative trajectories for scenarios with a flexible number of agents through a novel neural network architecture. Agent interactions are modeled using a Transformer encoder. Exploiting the limited information from their own viewpoint, agents are able to generate predictions for other vehicles. The method builds on an AlphaZero-extension for continuous action spaces (A0C) by formulating a multi-agent training objective. Its policy enforces action bounds through a mixture model of transformed normal distributions.

The proposed algorithm is evaluated against an MCTS augmented with domain-specific heuristics. In a challenging driving scenario, the method is able to outperform the baseline in terms of sample efficiency and planning speed. In another scenario, the algorithm is able to recover the heuristic knowledge of the MCTS while learning tabula rasa. Ablation studies highlight important design choices and provide insights into the significance of the method's components. Finally, a comprehensive analysis illuminates the generalization capabilities of learned networks and highlights the potential of methods combining learning with planning and heuristics.

# Contents

# List of Abbreviations

**CDF** Cumulative Distribution Function

**CNN** Convolutional Neural Network

**dec-MDP** decentralized Markov Decision Process

**DQN** Deep Q Network

**DRL** Deep Reinforcement Learning

**DUCT** Decoupled UCT

**GMM** Gaussian Mixture Model

**MARL** Multi-Agent Reinforcement Learning

**MCTS** Monte Carlo tree search

**MDP** Markov Decision Process

**MLP** Multilayer Perceptron

**PDF** Probability Density Function

**POMDP** Partially Observable Markov Decision Process

**ReLU** Rectified Linear Unit

**RL** Reinforcement Learning

**RNN** Recurrent Neural Network

**SAC** Soft Actor-Critic

**UCT** Upper Confidence Trees

# List of Figures

# List of Tables

# 1 Introduction

Reinforcement Learning (RL) has been responsible for some of the biggest breakthroughs in the recent history of artificial intelligence. It has solved the long standing grand challenge of Go [73], achieved Grandmaster-level performance in the video game Starcraft II [81] and navigated balloons in the stratosphere [6]. Succeeding in tasks which experts deemed decades away has been made possible by a combination of classical RL with neural network function approximation called Deep Reinforcement Learning (DRL). Due to its tremendous achievements, some researchers have even hypothesized that reward maximization is sufficient for the construction of artificial general intelligence [76].

A tantalizing application domain for such an artificial intelligence is highly automated traffic. Futurists imagine meals delivered by autonomous vehicles and parcels arriving via autonomous drones. Seamless cooperation between cars and trucks on highways makes the congestion plaguing large cities a thing of the past. Along the way, the efficiency gains of fully autonomous traffic reduce humanity's carbon footprint and contribute towards overcoming the biggest challenge of our generation: climate change [67].

Why has this future not been realized yet despite DRL's fabulous success stories? As amazing of a tool as it is, DRL is still bedeviled by many issues. Particularly its deployment to the real world has been impeded by what is in robotics called the *reality gap*. Looking at the achievements presented in the introductory paragraph, a common thread emerges: DRL has been successful in domains where simulators can provide the vast amounts of data needed for trial and error learning. On the other hand, progress in real-world applications has been slow.

Delving deeper into the reality gap, a number of smaller issues emerge which together hamper the adoption of DRL for autonomous driving. A recent survey has identified nine challenges holding back RL in the real world [21]. Of these, four are of particular interest and together with a fifth question form a set of guiding research objectives for this thesis:

1. Being able to learn on live systems from limited samples.
2. Learning and acting in high-dimensional state and action spaces.
3. Reasoning about system constraints that should never or rarely be violated.
4. Being able to provide actions quickly, especially for systems requiring low latencies.
5. Dealing with other learning agents within the same environment.

In the autonomous driving setting on which this work is based, cooperative driving scenarios are given and implemented in a simulator. An Monte Carlo tree search (MCTS)

based planner is used to solve these situations [46]. The main research objective can therefore be narrowed down to:

*How can the sample efficiency of a cooperative planning algorithm be improved?*

To answer this research question, a combination of DRL and the aforementioned MCTS is employed. The method is motivated by the success of AlphaZero in Go, where intertwining learning and search has led to superior performance compared to supervised learning [73, 75]. Building on previous work, a hybrid input representation is used by a neural network to prune likely unpromising actions from the MCTS sampling [47]. The network is inspired by recent Transformer architectures [80, 19, 20] and uses a self-attention mechanism to learn proposal distributions for a flexible number of agents. Through the Transformer, an agent is able to predict the behavior of other agents with the limited data available from its own point of view. Usage of a transformed Gaussian distribution allows the enforcement of action bounds, thereby conforming to the physical limitations of real-life vehicles [29]. Lastly, the algorithm is trained by extending a recently proposed AlphaZero modification (A0C) for continuous action spaces [58] to multi-agent environments.

The structure of this thesis is given as follows: First, the theoretical basis for the proposed method is defined by an introduction to RL. Then, MCTS is explained along with some relevant modifications. Neural networks are introduced next, with a focus on important building blocks for this thesis. The third chapter constitutes a short review of related literature and provides context to the approach. Afterwards, the environment is defined, which is followed by the derivation of a transformed normal distribution. Subsequently, the A0C loss is extended to multi-agent settings before the network architecture is introduced. Last in the chapter, the guided MCTS procedure is described. This is followed by a short section on implementation details. The next chapter presents the evaluation results of the proposed algorithm and discusses its empirical performance. A discussion of the results along with their limitations is followed by a conclusion and an outlook.

To summarize, this thesis provides the following contributions to extant literature:

- Novel self-attention based network architecture suitable for applying guided MCTS to scenarios with a flexible number of agents.
- Extension of the A0C loss for continuous domains to multi-agent settings.
- Thorough empirical evaluation demonstrating the sample efficiency and generalization abilities at test time.

# 2 Preliminaries

This chapter introduces the theoretical foundations of the proposed method. First, the basics of Reinforcement Learning (RL) are described starting with Markov Decision Processes (MDPs). Following is a subsection extending MDPs to a more general multi-agent setting: decentralized Markov Decision Processes (dec-MDPs). Then relevant RL terminology is introduced. Next, the concept of *Centralized Training with Decentralized Execution* is explained, which is a very common framework for training Multi-Agent Reinforcement Learning (MARL) algorithms. The second sub-chapter describes the Monte Carlo tree search (MCTS) planning algorithm and its most common instantiation, Upper Confidence Trees (UCT). This is followed by two important extensions that allow the application of MCTS to problems with continuous action spaces and multiple agents. The last section introduces neural networks with a particular focus on concepts relevant to the approach chosen in this thesis. These are namely Convolutional Neural Networks (CNNs), the self-attention mechanism and Gaussian Mixture Models (GMMs).

## 2.1 Reinforcement Learning

The following chapter gives an introduction into the main RL formalism, namely MDPs. This problem specification is then extended to multi-agent settings with dec-MDPs. RL terminology is described next before the chapter ends with a brief description of a common MARL training paradigm.

### 2.1.1 Reinforcement Learning as a Markov Decision Process

Borrowing slightly adapted notation from [24], an RL problem can be formalized as a 5-tuple $\langle \mathcal{S}, \mathcal{A}, P, \mathcal{R}, \gamma \rangle$, where[1]:

- $\mathcal{S}$ is the *state space* of the environment.
- $\mathcal{A}$ is the *action space* of the environment.
- $P(s'|s,a) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ is the *transition function* specifying the dynamics of the environment. That is, given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, the transition function outputs a probability distribution over the next state $s' \in \mathcal{S}$.
- $\mathcal{R}(s,a,s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function* of the environment, mapping a *transition* $(s,a,s')$ to a real-valued reward $r \in \mathbb{R}$.

[1] To make the notation more concise, in the following text $s' \coloneqq s_{t+1}$, $a' \coloneqq a_{t+1}$, $s \coloneqq s_t$ and $a \coloneqq a_t$ are used as abbreviations.

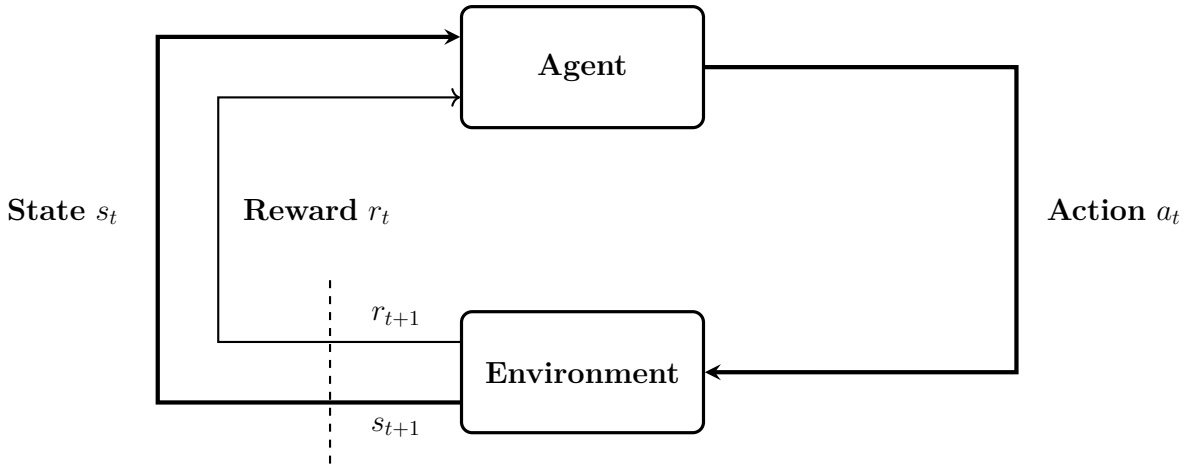



Figure 1: MDP control loop. The agent receives a state $s_t$ and a reward $r_t$. Once the action $a_t$ is selected, the environment transitions to the next state $s_{t+1}$. The transition yields a new reward $r_{t+1}$.

- $\gamma \in [0, 1]$ is a *discount factor* which weighs future rewards.

In an MDP, the agent and the environment interact with each other over a sequence of discrete time steps $t = 0, 1, 2, \ldots$. More specifically, at time step $t$ the environment is in state $s \in \mathcal{S}$. The agent perceives this representation and chooses an action $a \in \mathcal{A}$. After execution of action $a$, the environment transitions into the next state $s' \sim P(s'|s, a)$. The transition $(s, a, s')$ also yields a reward $\mathcal{R}(s, a, s')$ for the agent. The whole process is illustrated schematically in Figure 1. Resulting from the agent-environment interaction is a sequence of transitions which is called a *trajectory* and denoted as

$$T = \Big( (s_0, a_0, r_0),\ \ldots,\ (s_{t-2}, a_{t-2}, r_{t-2}),\ (s_{t-1}, a_{t-1}, r_{t-1}),\ s_t \Big) \ , \tag{2.1}$$

where $s_t$ is a terminal state [24].

There are two important characteristics of the above MDP definition meriting further explanation. First, the process is *fully observable*. This means that the agent observes the true state of the environment. A model that extends MDPs to partially observable environments is the Partially Observable Markov Decision Process (POMDP), which is not considered in this work. The second key characteristic of the MDP framework is that the transition function exhibits the *Markov property*. It requires that $P(s'|s, a)$ solely depends on the previous state-action pair $s, a$ instead of the whole sequence of previous states and actions [77]. The Markov property can be stated formally as

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) \ . \tag{2.2}$$

On one hand, this assumption simplifies modeling the transition dynamics significantly. On the other hand, the Markov property imposes restrictions on the state $s \in \mathcal{S}$, as it must now encode the knowledge of all previous agent-environment interactions [77].

The rule by which an agent selects actions is called a *policy* and denoted as

$$\pi(a|s) : \mathcal{S} \times \mathcal{A} \to [0, 1] \ . \tag{2.3}$$

It is a mapping from an action $a$ to the probability of selecting that action given a state $s$.

Finally, the goal of the agent within the MDP framework is to find a policy that maximizes the discounted future reward $\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ [77]. This can be expressed in terms of the so called *value function* [77]

$$V^{\pi}(s) = \mathbb{E}_{\pi}\Big[ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\Big|\, s_t = s \Big], \quad \forall s \in \mathcal{S} \ , \tag{2.4}$$

where $V^{\pi}(s)$ is the expected reward when starting in state $s$ and following policy $\pi$.

The agent's optimization goal can now be expressed in terms of the value function as

$$V^{*}(s) = \max_{\pi \in \Pi} V^{\pi}(s) \ , \tag{2.5}$$

where $V^{*}(s)$ is the optimal value function. Another function playing an important role in RL is the action-value function or *Q function*. It is similarly defined to the value function with the key difference being that it additionally conditions on the action [77]:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\Big[ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\Big|\, s_t = s, a_t = a \Big] \ . \tag{2.6}$$

The optimal Q function is defined analogously [24]:

$$Q^{*}(s, a) = \max_{\pi \in \Pi} Q^{\pi}(s, a) \ . \tag{2.7}$$

A key aspect in which the Q function differs from the value function is that it allows a direct expression for the optimal policy $\pi^{*}(a|s)$. If in each state $s \in \mathcal{S}$ the agent selects the action $a \in \mathcal{A}$ with the highest Q-value, it automatically follows the optimal policy [24]. Equation 2.8 shows this relationship

$$\pi^{*}(s) = \arg\max_{a \in \mathcal{A}} Q^{*}(s, a) \ . \tag{2.8}$$

The formalism described above covers single-agent settings. Since the goal of this thesis is to learn behavior in environments with multiple agents, it must be extended. A general extension of MDPs for multiple agents is called a dec-MDP and described in the next section.

### 2.1.2 Decentralized Markov Decision Processes

So far only solutions single-agent MDPs have been considered. In the real world however, agents often compete with each other for resources or cooperate in teams to achieve a common goal. Incorporating the set of agents into the problem specification, a dec-MDP can be described as the 6-tuple $\langle \Upsilon, \mathcal{S}, \mathcal{A}, P, \mathcal{R}, \gamma \rangle$[62, 46], where:

- $\Upsilon$ is the set of available agents with indices $i \in \{1, 2, \ldots\}$.
- $\mathcal{S} = \times \mathcal{S}_i$ represents the *joint state space*, where $\mathcal{S}_i$ denotes the state space of agent $i$.
- $\mathcal{A} = \times \mathcal{A}_i$ formalizes the *joint action space*, where $\mathcal{A}_i$ denotes the action space of agent $i$.
- $P(s'|s, \mathbf{a}) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition function specifying the dynamics of the environment. Note that the transition function now conditions on the *joint action* $\mathbf{a}$ of all agents.
- $\mathcal{R}(s, \mathbf{a}, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function of the environment, mapping a *joint transition* $(s, \mathbf{a}, s')$ to a real-valued reward $r \in \mathbb{R}$.
- $\gamma \in [0, 1]$ is a *discount factor*, describing the influence of future rewards.

In the following work, a variable denoted with the subscript $_i$ refers to the agent $i$.

As with MDPs, a dec-MDP makes two important assumptions which require further examination. The first assumption is that agents select their actions independently without knowing about other agent's decisions. While the solution to the dec-MDP is a joint policy $\Pi = \langle \pi_i, \ldots, \pi^n \rangle$, each agent only optimizes its individual policy $\pi_i : \mathcal{S}_i \times \mathcal{A}_i \to [0, 1]$ [62].

The second assumption is that the dec-MDP is *jointly observable* [62], meaning that combining the observations of all agents allows recovering the exact state of the environment. To give a concrete example relevant to this work, consider an environment where multiple cars are on a road and each car observes its own location and dynamics (e.g. velocity, steering angle, etc.) perfectly. Then combining the observations of all vehicles would yield the complete state of the environment.

### 2.1.3 Reinforcement Learning Terminology

Since its inception in the late 1970's [77], RL as a field has developed its own terminology for a variety of concepts. Since some of them are relevant to understand content in later chapters of this work, they are briefly explained here.

Recall that the goal of RL from Section 2.1.1 is to find a policy $\pi(a|s)$ that maximizes future reward. The question then becomes: How can such a policy be learned? In general, RL algorithms can be categorized into two broad classes: *policy-based* and *value-based*. Policy-based methods try to directly learn a policy which maximizes Objective 2.4. A very common class of algorithms within this framework are policy gradients, among which REINFORCE is known best [86]. Value-based algorithms try to instead learn a value

function or Q-function from which the optimal policy can be derived. Q-Learning and an extension called Deep Q Network (DQN) [57] are its most prominent representatives.

DQNs belong to a major area of focus for current research in RL called Deep Reinforcement Learning (DRL). Consider a simple environment like a $9 \times 9$ grid maze: In such an environment it is possible to simply store all relevant quantities in a table. As state or action spaces grow large or even infinite, the approach quickly becomes intractable. A solution to this problem is to approximate the Q-function or policy using deep neural networks, which are introduced in section 2.3. The combination of RL and function approximation is called DRL and responsible for many recent breakthroughs in artificial intelligence [57, 73].

A learned policy can be either stochastic $\pi(a|s)$ or deterministic $\pi(s)$. Given a state $s$, a deterministic policy will always select the best known action $a$. This inhibits exploration of actions which are currently deemed suboptimal but could provide better returns in the long run. To alleviate the issue, deterministic policies usually inject randomness into the action selection to encourage exploration. Examples are $\varepsilon$-greedy action selection [57] or adding action noise via some stochastic process [54, 75]. Another option is to learn the parameters of a distribution from which actions can then be sampled. Particularly in continuous control settings this is a common strategy, where often the parameters of a normal distribution are learned. An example of an algorithm learning a stochastic policy is Soft Actor-Critic (SAC), which is a very competitive model-free method for continuous control [28].

The term *model-free* in the last sentence refers to systems that are only learning from trial-and-error [77]. *Model-based* methods on the other hand rely on a model of the environment to plan future actions. This model can either be given, for instance through the rules of the game like in chess, or it can be learned by the algorithm [77]. The system described in Section 4 is an example of a model-based algorithm.

Lastly, RL systems can be differentiated into *off-policy* and *on-policy* algorithms. To give a succinct explanation: "*On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data*" [77]. To illustrate the difference, consider again the example of DQN. Just like this thesis (see Algorithm 1), it uses a replay buffer to store past experiences and learn from them. These experiences however are all gathered from different policies, since between environment interactions training occurs. It is therefore an off-policy method. On-policy methods on the other hand are trained using only data gathered from the current policy. The reuse of previous experiences in what is called experience replay allows off-policy algorithms to be more sample efficient [24].

### 2.1.4 Centralized Training with Decentralized Execution

In the dec-MDP framework, each agent selects its own actions independently of the other agent's choices. This paradigm is called *decentralized execution*. During training however, it is desirable that agents have access to additional information to accelerate the training process [56]. Due to the nature of training RL agents in a simulator, conditioning on extra knowledge is a commonly used approach. This gives rise to a paradigm called *Centralized Training with Decentralized Execution*, where the agent may utilize additional data during training which it cannot access at inference time [23].

## 2.2 Monte Carlo Tree Search

MCTS is a decision-time planning algorithm which uses multiple sampled trajectories to approximate an action-value function [77]. At each step, simulations are run until a computational budget is exhausted, building a tree rooted at the current state. The simulation statistics accumulated at the root node are then used to make a final decision. As a model-based method it relies on knowledge of the environment's transition dynamics, which are usually known beforehand (e.g. in games like Go) but could also be learned[2] [24]. MCTS exhibits a number of desirable characteristics [12]:

- **Aheuristic**: While the algorithm's performance can often be increased by including domain-specific knowledge into the search[3], the base version already works on a wide variety of problems without any modifications.
- **Anytime**: MCTS can be terminated at any time during the search and immediately yield up-to-date results.
- **Asymmetric**: The algorithm's search favors more promising nodes in the search tree, thus providing more accurate results for important regions of the tree.

The following chapter provides an overview over the MCTS algorithm, starting with the most popular single-agent variant called Upper Confidence Trees (UCT). This algorithm is then extended to be applicable to dec-MDPs with a continuous action space in the subsequent sections. The MCTS description mostly follows [12] using notation adopted from [58] and [73, 75] to keep consistency with Chapter 4.

[2] An example of an algorithm that learns the transition dynamics is *MuZero* [70].

[3] In Section 6.2 the effect of heuristics on the MCTS used as baseline for this work is evaluated.

### 2.2.1 Upper Confidence Trees

To describe the basics of MCTS, it makes sense to start with a simple MDP first. It is simple in the sense that the action space $\mathcal{A}$ is discrete and finite. An additional assumption is that $|\mathcal{A}|$ is "small", where small in this case means that the cardinality of the set $\mathcal{A}$ is smaller than the number of MCTS iterations. While the algorithm is still applicable if this condition is not met, it requires modifications to perform well. One such modification is introduced in Section 2.2.2.

At each state $s \in \mathcal{S}$ of the environment, the UCT algorithm builds a new tree starting from the current state. As this state constitutes the root of the search tree it is denoted as $s_0$. Each node within the tree stores an environment state $s \in \mathcal{S}$[4]. Each edge consists of a state-action pair $(s, a)$ and stores a set of statistics $\{n(s,a), W(s,a), Q(s,a)\}$[5], where $n(s,a)$ is the *visitation count* and $W(s,a)$ is the cumulative sum of returns for each action. $Q(s,a) = W(s,a)/n(s,a)$ represents the approximated state-action value [58].

The MCTS algorithm then grows the tree with each simulation run according to the four steps illustrated graphically in Figure 2 and described in the following. The name Upper Confidence Trees (UCT) comes from applying a selection policy based on upper confidence bounds [43, 42]. Through this, the algorithm is able to find a solution to the exploration-exploitation dilemma.

1. **Selection**
   A so called *tree policy* $\pi_{tree}$ descends down the tree starting from the root node $s_0$, selecting actions according to the UCB formula

   $$\mathrm{UCT}(a) = Q(s,a) + C_{uct}\sqrt{\frac{\log n(s)}{n(s,a)}} \tag{2.9}$$

   until a leaf node is reached. Here $n(s) = \sum_a n(s,a)$ denotes the total visitation count of state $s$ and $C_{uct} > 0$ is a constant controlling the weight of the exploration term. A node is defined to be a leaf node if it is both non-terminal and expandable (i.e. has unvisited actions) [12]. Note that the correct exploration-exploitation trade-off is only achieved if $Q(s,a)$ is properly scaled between 0 and 1 to not diminish the exploration term [12].

2. **Expansion**
   Once a leaf node is reached, the tree is expanded by selecting a previously unvisited action $\bar{a}$, leading to a new leaf state $s_L$. $L$ denotes the depth of node $s_L$ in the tree and indicates that it is a leaf node. This is usually achieved by setting $\mathrm{UCT}(\bar{a}) = \infty$

[4] Usually environment states are vectors or matrices and are denoted in bold. To improve readability, the bold font is omitted in the MCTS section.

[5] To keep it simple, subscripts for node depth are used only when they are needed for disambiguation.

for all actions $\bar{a}$ that have not been visited yet. If there are multiple unexplored actions to choose from, the tie can be broken by selecting an action at random [12].

3. **Simulation**
   Starting from the newly selected node $s_L$, a *simulation policy* $\pi_{simulation}$ is used to perform a Monte Carlo rollout until a predetermined depth or a terminal stage is reached. The simplest possible policy $\pi_{simulation}$ is to select an action uniformly at random from the set of available actions $\mathcal{A}$. Once the ending conditions are met, the simulation policy halts with a stored trajectory $T$

$$T = \Big((s_{L+1}, a_{L+1}, r_{L+1}),\ \ldots,\ (s_{L+D}, a_{L+D}, r_{L+D})\Big)\ . \tag{2.10}$$

   The accumulated and discounted rewards $\Delta = \sum_{d=0}^{D-1} \gamma^d r_{L+d+1}$ can now be used as a Monte Carlo estimate of the leaf node's value function $V(s_L)$ [58].

4. **Backup**
   In the last step, the nodes that have been visited on the descent down the tree must be updated with the estimated value of the new node. Given the trace

$$T = \Big((s_0, a_0, r_0),\ \ldots,\ (s_{L-1}, a_{L-1}, r_{L-1})\Big) \tag{2.11}$$

   within the tree, the total action values of the trace can be computed recursively as

$$R(s_d, a_d) = r(s_d, a_d) + \gamma R(s_{d+1}, a_{d+1}), \quad 0 \leq d < L\ . \tag{2.12}$$

   The recursion is initialized with the Monte Carlo value estimate of the expanded node $R(s_L, a_L) = \Delta = V(s_L)$. Now the cumulative sum of returns for all actions visited during the iteration is updated for each edge $(s_d, a_d)$ with $W(s_d, a_d) \leftarrow W(s_d, a_d) + R(s_d, a_d)$. The corresponding visitation counts are incremented with $n(s_d, a_d) \leftarrow n(s_d, a_d) + 1$. Finally, the update can be completed by re-computing the approximate state-action values $Q(s_d, a_d) = W(s_d, a_d)/n(s_d, a_d)$. This backup step is then recursively applied to all nodes up the tree until the root node $s_0$ is reached [12, 58].

Once a fixed number of iterations $N$ is completed, the algorithm terminates with a tree of exactly $N$ nodes, since with each iteration one leaf node $s_L$ has been added. The root node now holds the visitation counts for each available action $n(s, a)$, $\forall a \in \mathcal{A}$ together with their action-value estimates $Q(s, a)$. There are multiple criteria for selecting the action to be executed within the environment, among which two are commonly used [12]:

1. $a_{\text{final}} = \max_a Q(s_0, a)$: selection of the action with the highest action-value estimate.
2. $a_{\text{final}} = \max_a n(s_0, a)$: selection of the action that has been visited the most.

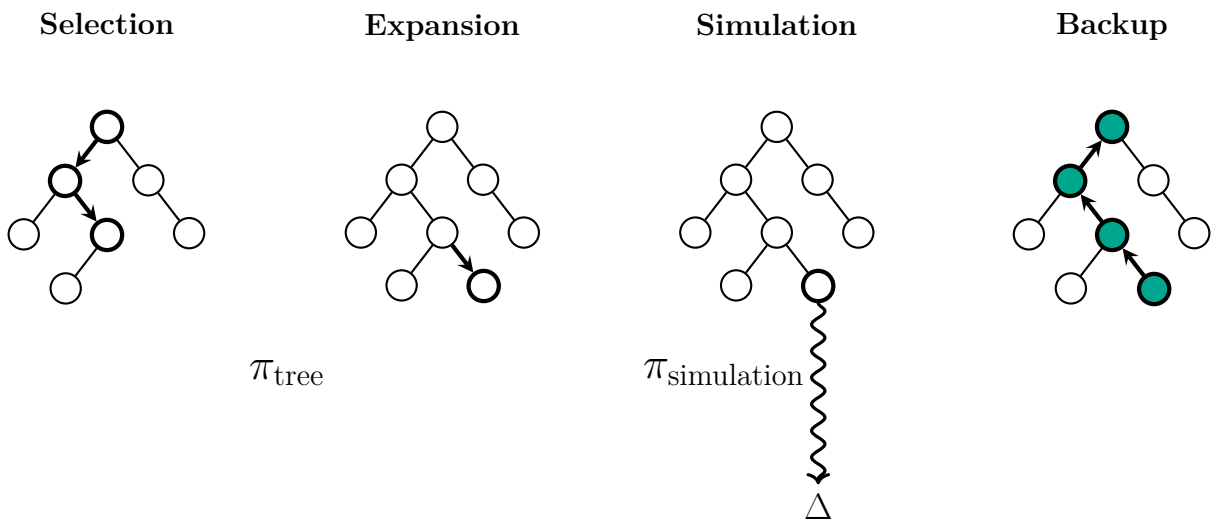



Figure 2: Illustration of the four phases in each MCTS iteration. The tree policy $\pi_{\text{tree}}$ traverses the tree until an expandable node is reached. A simulation policy $\pi_{\text{simulation}}$ then performs a rollout, resulting in an outcome $\Delta$. Finally, the tree statistics are updated in the backup phase. Graphic is based on [12].

While the first selection rule seems intuitive at first, it suffers from the high variance of $Q(s, a)$ estimates for actions with low visitation counts. A more common choice is therefore the selection of $\max_a n(s_0, a)$ since the visitation counts are more robust to outliers. Usually though both rules are tried and the decision is made based on empirical performance in the application domain.

The plain UCT algorithm described above can neither select actions for multiple agents nor can it deal with continuous action spaces. Since the goal of this thesis is to plan cooperative driving trajectories, it must be extended to function in such settings. These necessary extensions are the topic of the next two sections.

### 2.2.2 Monte Carlo Tree Search in Continuous Action Spaces

UCT as discussed so far cannot handle continuous action spaces. To see why, it is worth remembering that any interval on the real line contains infinitely many numbers. Therefore $|\mathcal{A}| = \infty$ and the MCTS never stops exploring actions at the root node, growing a tree without any depth. A technique that deals with this dilemma and allows for exploitation in continuous action spaces is called *Progressive Widening* [18]. It proceeds as follows: Each time the tree policy visits a node in the tree (including the root node), the criterion in Equation 2.13 is evaluated and the number of available actions $m(s)$ is determined [58].

$$m(s) = C_{pw} \cdot n(s)^{\alpha_{pw}} \ . \tag{2.13}$$

Here $C_{pw} > 0$ and $\alpha_{pw} \in [0, 1]$ are constants and control whether a flat tree (exploration) or a deep tree (exploitation) is built [18]. If the number of already chosen actions $|A^s|$ in state $s$ is smaller than $m(s)$, the conditions for progressive widening are met. Then one or more new actions are sampled from the action space and added as edges. Through this procedure, the tree slowly grows larger in areas that are visited more often, gradually filling out the interval on which the continuous action space is defined [18]. The most basic strategy for sampling new actions is uniform sampling. More sophisticated approaches are however also possible as will be explored in this thesis (see Section 4.4).

### 2.2.3 Monte Carlo Tree Search in Multi-Agent Settings

UCT with progressive widening as introduced in the previous section has the ability to deal with continuous action spaces. It is however not applicable to dec-MDPs out of the box. In the following, an extension to UCT called Decoupled UCT (DUCT) is described which has been shown to work well empirically in settings with multiple agents [78, 50].

The core idea behind DUCT is that each agent $i \in \Upsilon$ maintains its own set of statistics $\{n_i(s, a), W_i(s, a), Q_i(s, a)\}$. All rewards and visitation counts are treated as if there were no dependency between them which is called *decoupling*. As a result, each agent retains its own tree policy $\pi_{tree,i}$ by independent application of Formula 2.9.

During the selection phase, iterative action selection for all agents without knowledge of each other's choices generates a joint action $\mathbf{a} \in \times\mathcal{A}_i$ [78]. Since the UCT value of an unexplored action is infinite, it is ensured that each agent tries each action at least once. This does not mean however that all combinations of all actions are selected at least once. As a consequence, the full joint action space $\times\mathcal{A}_i$ is not completely explored [78]. If the joint action $\mathbf{a}$ has already been chosen previously, the search progresses to the corresponding node and recursively applies the selection policy again.

If at least one agent chooses an unexplored action or the combination of independent best actions $a_i$ has not been selected yet, the search tree is expanded and a leaf node is added. Compared to Section 2.2.1, the edge leading to the new node now comprises a joint action and a joint state $(\mathbf{s}, \mathbf{a})$, where the joint state $\mathbf{s}$ is the concatenation of the individual agents' states.

In the simulation phase, each agent now has its own rollout policy $\pi_{simulation,i}$ to sample actions from (analogous to the plain UCT algorithm). Together they form a joint action for all agents and allow performing Monte Carlo estimates of the leaf node in a multi-agent setting. Similarly to the single-agent case, uniform random sampling for each agent is a natural choice as rollout policy. The simulation results in a reward vector $\mathbf{\Delta} = (\Delta_1, \ldots, \Delta_{|\Upsilon|})$ that has to be backed up through the tree. Since all actions chosen during the selection phase have been joint actions, each agent increments its own, independently

stored values and visitation counts and updates its action-value estimates $Q_i(s, a)$ [78].

In the last step, each agent individually selects the action with the highest $Q_i(s, a)$, forming a joint action that is then executed in the environment. To what does the algorithm described above converge? This can be analyzed using tools from game theory, where each step of the dec-MDP can be seen as a simultaneous move game. The solution to such a game is called a Nash equilibrium, which is defined by mutual best responses. This means that no player has a reason to switch strategies given the other player's strategy. The DUCT algorithm does not converge to such an optimum [78]. However, its strong empirical performance compared to other approximate solution methods still makes it a good choice for a dec-MDP [78, 50].

The extension of DUCT to continuous action spaces is straightforward: As each agent has its own tree policy $\pi_{tree,i}$, it is able to evaluate criterion 2.13 on its own and decide whether a new action needs to be sampled or not. Consequently, the number of actions chosen may be different for each agent in each node of the tree. While this poses no problem for the DUCT algorithm, it makes training a network to guide the search more challenging, as will be discussed in Section 4.3.3.

## 2.3 Neural Networks

In simple Reinforcement Learning (RL) environments, it is often possible to store the value function or $Q(s, a)$ in a table and compute the optimal policy. Most interesting applications however have either large state spaces (e.g. images) or large action spaces, for instance in continuous control. In these settings, an optimal policy or optimal value function can only be approximated.

One commonly used technique for function approximation — originally intended for supervised learning — are neural networks, which are the topic of this chapter. The first section introduces artificial neurons and how they can be combined to construct more expressive, *deep* models. Then Convolutional Neural Networks (CNNs) are briefly explained as they form the backbone of most current advances in image processing. Next, an attention mechanism is introduced which allows a neural network to learn interactions between elements of a sequence. The last section discusses a class of models that can represent an arbitrary conditional probability distribution called Gaussian Mixture Models (GMMs).

### 2.3.1 Deep Learning

The basic model of an artificial neuron has been introduced in 1958 as the "Perceptron" and has not changed much since then [68]. Figure 3 depicts its structure. The goal of such a model is to approximate some function $y = f^*(\mathbf{x})$, for instance in a regression problem

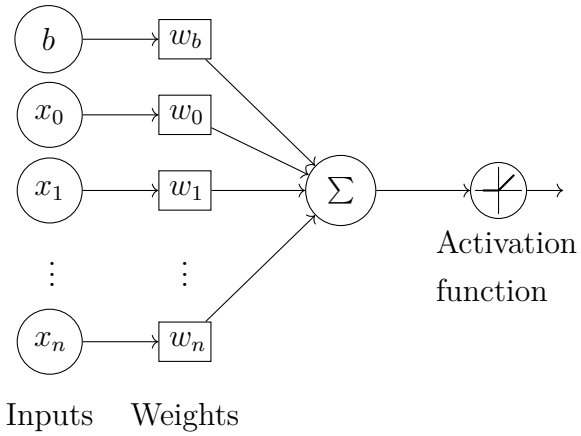



Figure 3: The Perceptron. First inputs are weighted and summed together with a bias term. Then they are transformed by a nonlinear activation function.

[26]. Inputs are fed into the neuron and weighted by a corresponding weight, hence the term "*feedforward network*" for networks composed of such neurons. Then the weighted inputs are summed together with a bias term and transformed by a nonlinear activation function. Mathematically, the process can be described through Equation 2.14 [26]

$$h = g(\mathbf{w}^T\mathbf{x} + b) = \sum_{a=0}^{A} w_a x_a + b \ , \tag{2.14}$$

where the inner function $\mathbf{w}^T\mathbf{x} + b$ defines an affine-linear transformation controlled by a weight vector $\mathbf{w}$ and a scalar bias term $b$. $g$ denotes the activation function. Some important options for $g$ are introduced in later paragraphs of this section.

The simple model outlined above cannot learn complex relationships between inputs and outputs. A common example is the XOR function or "exclusive or" [26]. To be able to solve these kinds of problems, compositions of different functions are needed such that $y = f^D(\dots f^2(f^1(\mathbf{x})))$. Here $D$ specifies the depth of the network. Such a composition of neurons is called Multilayer Perceptron (MLP). A single layer can be represented by Equation 2.15:

$$\mathbf{h} = g(\mathbf{W}^T\mathbf{x} + \mathbf{b}) \ . \tag{2.15}$$

It uses a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$ to produce a vector-valued output $\mathbf{h}$. The activation function $g$ is now applied element-wise. Figure 4 depicts a possible network topology for an MLP with one hidden layer.

Now that the basic concepts of deep learning have been introduced, the choice of an activation function can be discussed. Clearly it must be a nonlinear function, because if $g$ is chosen to be linear, the whole network collapses into a linear model [26]. What activation function should be used then? One commonly used function is called the

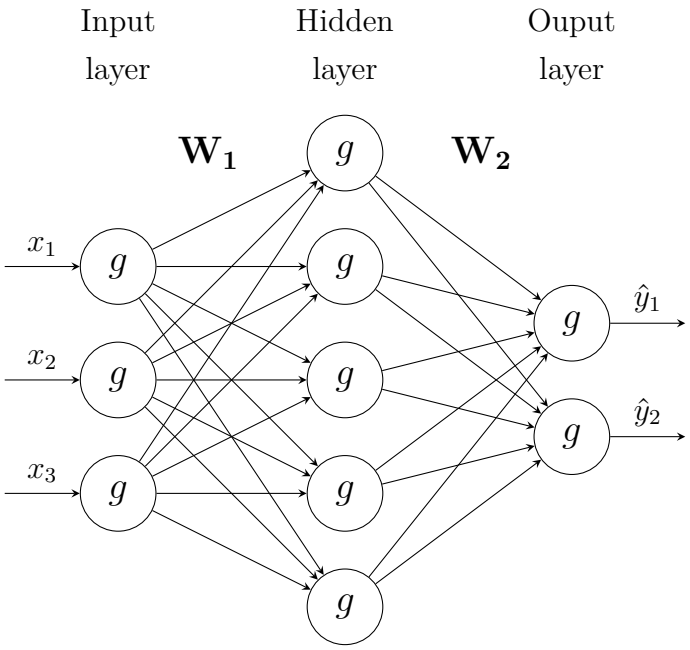



Figure 4: Visualization of an MLP model. Inputs are passed into a first layer and transformed by a weight matrix $\mathbf{W}_1$ and a nonlinear activation function $g$. The process is repeated to produce estimates $\hat{y}_1, \hat{y}_2$. The bias term is not visualized.

*sigmoid function* and defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \ . \tag{2.16}$$

However, it has multiple drawbacks. First, the exponential in the denominator is expensive to compute. While this is not an issue for small neural networks it can quickly add up to a noticeable performance overhead in large models with billions of neurons. Second, for large positive or negative values of $x$ the gradient converges towards 0, making training difficult [26].

An alternative that fixes the first problem of the sigmoid and partially addresses its second shortcoming is the Rectified Linear Unit (ReLU) [26]. It is simple to compute mathematically and can be expressed as

$$\text{ReLU}(x) = \max(0, x) \ . \tag{2.17}$$

It is a linear function for positive input $x$ and 0 else. While it is easy to compute, it is still nonlinear and allows the network to function [26].

The last import activation function that needs to be discussed is the Softmax. It is commonly used as an output function for classification problems since it can produce a valid probability distribution over output classes. The probability for class $i$ is defined as the exponential $e^{x_i}$ of the activation $x_i$, which is then normalized by the sum of exponentiated

activations over all classes $k = 1, \ldots, K$ [26].

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \ . \tag{2.18}$$

MLPs are a good choice for processing a wide array of numerical inputs. They do however lack the inductive biases needed to succeed in domains where the data has a certain structure like images or time series. A very common architecture with such a bias are CNNs [51]. Since they are particularly successful at processing images and the concept described in this work relies on visual input, they are introduced next.

### 2.3.2 Convolutional Neural Networks

CNNs [51] are a widely-used architecture of neural network for all kinds of tasks but excel at processing visual input. In image classification, they have been responsible for arguably the biggest breakthrough in modern machine learning [44] by introducing three key concepts into the architecture [26]:

1. **Sparsity**. In an MLP, every input unit is connected to every output unit. Interactions in convolutional layers are limited to local neurons.
2. **Parameter sharing**. The localized interactions of a CNN share weights within a layer.
3. **Equivariance to translation**. If a pattern in an image is moved to another location, its feature map will be moved by the same amount.

Properties one and two result in large efficiency gains as a convolutional layer only has to store a fraction of the weights compared to an MLP [26]. Property three is desirable for image processing since detecting an edge should succeed irrelevant of its place within the image.

Using notation from the Squeeze-and-Excitation authors [36] , a convolutional layer transforms an input $\mathbf{X} \in \mathbb{R}^{C' \times H' \times W'}$ to a feature map $\mathbf{U} \in \mathbb{R}^{C \times H \times W}$ using a set of $C$ filter weights $\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_C]$. The transformation can be written mathematically (without bias terms) as

$$\mathbf{u}_c = \mathbf{v}_c * \mathbf{X} = \sum_{s=1}^{C'} \mathbf{v}_c^s * \mathbf{x}^s \ , \tag{2.19}$$

where $*$ denotes the convolution operation. In Equation 2.19, a single output feature map $\mathbf{u}_c \in R^{H \times W}$ for channel $c$ is constructed by applying the corresponding filter $\mathbf{v}_c = [\mathbf{v}_c^1, \ldots, \mathbf{v}_c^{C'}]$ to each channel of the input $\mathbf{X} = [\mathbf{x}^1, \ldots, \mathbf{x}^{C'}]$ [36]. Note that the filter $\mathbf{v}_c$ for a single output channel $\mathbf{u}_c$ has a set of weights for each channel $1, \ldots, C'$ of the

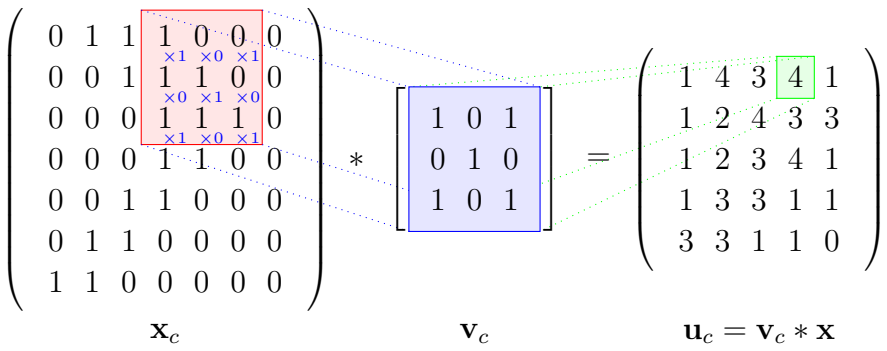



Figure 5: Visualization of a single computation in a $2D$ convolution operation. Local values of the input feature map $\mathbf{x}_c$ are processed by a $3 \times 3$ kernel and produce a single output value. The process is simultaneously performed in all locations to produce the output feature map $\mathbf{u}_c$.

input image. These weights are shared for all local interactions however, indicated by the constant subscript $_c$. One such interaction is visualized in Figure 5, where a filter processes local input values to produce a single value in the output feature map.

CNNs model channel dependencies implicitly through the convolutional filter being applied to all input channels [36]. Since the filter only captures local dependencies, the learned features lack global context. Squeeze-and-Excitation networks [36] have popularized the concept of *channel attention*[6], which is a mechanism that explicitly allows the network to learn channel dependencies. More recently, ECA-Net has been proposed and includes a simplified and more efficient method to add global channel context to the learning [83]. It describes a two-step mechanism, where in the first step channel statistics are aggregated via $2D$ global average pooling:

$$z_c = \frac{1}{H \times W} \sum_{h=1}^{H} \sum_{w=1}^{W} u_c(h, w) \ . \tag{2.20}$$

The channel statistic $z_c$ can be interpreted as a summary representation of the local features in channel $c$. Performing global pooling over all channels of the feature map $\mathbf{U}$ results in a vector $\mathbf{z} \in \mathbb{R}^C$ containing the aggregated representations.

To learn interactions between the channel statistics $z_c$, one could now employ a fully connected neural network [36]. As has been discussed in this section however, convolutional layers are computationally more efficient. In ECA-Net, the channel dependencies are therefore learned using a $1D$ convolution with kernel size $k$:

$$\mathbf{s} = \sigma(\text{C1D}_k(\mathbf{z}_c)) \ . \tag{2.21}$$

The $1D$ convolution is denoted as C1D while $\sigma$ indicates the sigmoid activation function. The resulting feature vector $\mathbf{s}$ is now a representation of local channel interactions scaled

[6] The more general concept of neural attention is introduced in the next section.

between zero and one. It can be used to obtain a recalibrated output feature map $\tilde{\mathbf{X}}$

$$\tilde{\mathbf{x}}_c = s_c \mathbf{u}_c \tag{2.22}$$

by scaling each channel of the output feature map $\mathbf{u}_c$ with the corresponding channel activation weight $s_c$ obtained from the vector $\mathbf{s}$.

The last question remaining now is how the kernel size $k$ of the ECA-module is determined. The authors give a formula to automatically set the kernel size defined in Equation 2.23

$$k = \phi(C) = \left| \frac{\log_2(C)}{\gamma} + \frac{b}{\gamma} \right|_{odd} , \tag{2.23}$$

where $C$ is the channel dimension and $|\cdot|_{odd}$ indicates the nearest odd number. $\gamma = 2$ and $b = 1$ are constants given by the authors. The formula is derived in Appendix A.

Channel-attention is a successful mechanism for modeling global context in CNNs. It is however limited to applications in computer vision. To learn interactions between arbitrary sequences of inputs, a more general model of neural attention is needed. The *self-attention* mechanism described in the next section introduces one such concept.

### 2.3.3 Attention and Self-Attention

Neural attention architectures originate from natural language processing, where they have been proposed to solve the alignment problem in machine translation [5]. The attention mechanism in Sequence-to-Sequence models can be seen as a scoring function: It learns relative weights for each element of an input sequence used to decode a specific output element. As established in Section 2.3.1, such a function can be parameterized by a neural network. By conditioning on all elements of the input sequence with learned weights, the network is able to model long-range dependencies between the elements.

The Transformer proposed in [80] goes one step further and removes the recurrent encoder and decoder in Sequence-to-Sequence, solely relying on attention. The resulting architecture is able to learn complex dependencies between sequence elements in parallel. While the model was designed to learn interactions between word tokens in sentences, it has since then been applied to image data [20], pedestrian trajectory prediction [22] as well as RL [87]. This section introduces the basics of multi-head attention before it is applied to learn agent interactions in Chapter 4.5.

Multi-head attention is computed from three matrices: the queries $\mathbf{Q}$, the keys $\mathbf{K}$ and the values $\mathbf{V}$. Given the dimensionality of the queries and keys $d_k$, it can be described mathematically as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} . \tag{2.24}$$

Where do queries, keys and values come from? They are all computed from the same input sequence $(x_1, \ldots, x_N)$ and the representations generated from that input sequence $\mathbf{Z} = (\mathbf{z}_1, \ldots, \mathbf{z}_N)$[7] [80]. The concatenated representation $\mathbf{Z}$ of the input sequence is an $N \times d_{model}$ matrix. $\mathbf{Q}, \mathbf{K}$ and $\mathbf{V}$ can now be obtained by multiplying $\mathbf{Z}$ with three distinct weight matrices $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d_{model} \times d_k}$ and $\mathbf{W}^V \in \mathbb{R}^{d_{model} \times d_v}$. The resulting products are given below [80]:

$$\mathbf{Z}\mathbf{W}^Q = \mathbf{Q} \in \mathbb{R}^{N \times d_k} \tag{2.25}$$

$$\mathbf{Z}\mathbf{W}^K = \mathbf{K} \in \mathbb{R}^{N \times d_k} \tag{2.26}$$

$$\mathbf{Z}\mathbf{W}^V = \mathbf{V} \in \mathbb{R}^{N \times d_v} \ . \tag{2.27}$$

In the original model as well as this work, $d_k = d_v$ for simplicity. The matrix product $\mathbf{Q}\mathbf{K}^T$ now produces an $N \times N$ matrix, where each row contains unnormalized attention scores. After applying the Softmax function row-wise, multiplication with the values matrix $\mathbf{V}$ results in a representation $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \in \mathbb{R}^{N \times d_v}$. Each row of the self-attention representation is a convex combination of the rows of $\mathbf{V}$, where the summation weights are determined by the corresponding row of $\text{Softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})$[8]. To give an example: The second row of $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ is computed as linear combination of all the rows of $\mathbf{V}$, where the weights for the linear combination are given by the second row of $\text{Softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})$. The term $\sqrt{d_k}$ increases numerical stability by preventing the dot-product inside the Softmax from growing too large. Summing up: The self-attention mechanism computes a representation $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$, which is a weighted sum of a projected input representation $\mathbf{Z}\mathbf{W}^V = \mathbf{V}$. Since the weights are computed from the same input sequence (using different projections $\mathbf{Q}$ and $\mathbf{K}$), the whole architecture is called *self-attention*. Figure 6 visualizes the process as computational graph[9].

The last important component of the Transformer model is the concept of *multi-head attention*. It is defined in the original work through Equation 2.28 [80]:

$$\begin{aligned} \text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_H)\mathbf{W}^O \\ \text{where head}_h &= \text{Attention}(\mathbf{Z}\mathbf{W}_h^Q, \mathbf{Z}\mathbf{W}_h^K, \mathbf{Z}\mathbf{W}_h^V) \ . \end{aligned} \tag{2.28}$$

The second line in the formula immediately elucidates the core idea of multi-head attention: Instead of using just one matrix $\mathbf{Q}, \mathbf{K}$ and $\mathbf{V}$ for each self-attention layer of the

[7] In natural language processing, the input sequence is a sentence and the generated representation $\mathbf{z}$ are the embedded words.

[8] A blog post illustrating the row perspective of matrix multiplication instead of the commonly used dot-product perspective can be found at https://ghenshaw-work.medium.com/3-ways-to-understand-matrix-multiplication-fe8a007d7b26.

[9] Several visual guides can be recommended. The "Illustrated Transformer" (https://jalammar.github.io/illustrated-transformer) and "Transformers from scratch" (http://peterbloem.nl/blog/transformers) are particularly worth mentioning.

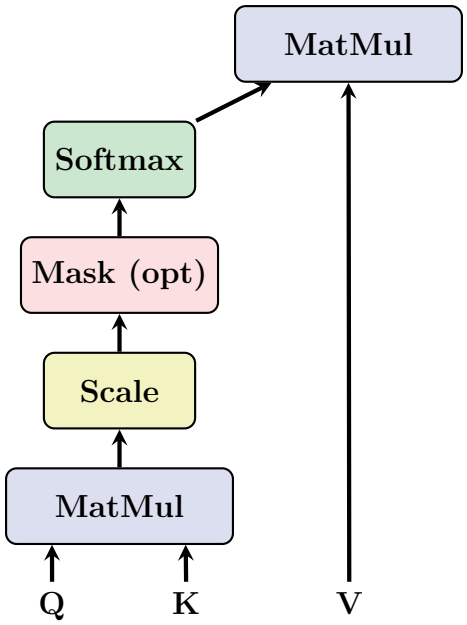



Figure 6: Computational graph of self-attention. Queries $\mathbf{Q}$ are multiplied with keys $\mathbf{K}$ and scaled. Optional masking prevents the model from attending to specific positions. After applying the Softmax function, values $\mathbf{V}$ are multiplied together with the learned weights. Graphic is based on [80].

model, $H$ different such matrices are used in parallel. This allows the Transformer to attend to different projection subspaces [80]. Apart from producing higher quality representations, multiple heads also increase computational efficiency by using $d_k = d_v < d_{model}$. This reduces the memory needed for the weight matrices $\mathbf{W}_h^Q$, $\mathbf{W}_h^K$ and $\mathbf{W}_h^V$. For example, the values are now computed on $d_v$ columns of the input representation $\mathbf{Z}\mathbf{W}^V_{:,:d_v} = \mathbf{V}$. The notation $:,:d_v$ is borrowed from array indexing in Python and denotes selecting all rows and the first $d_v$ columns. Thus the queries, keys and values for each head all focus on different parts of the input representation $\mathbf{Z}$ [80]. The only requirement imposed by splitting up the weights for each head like this is that $d_{model}$ must be divisible by $H$ to produce valid dimensions for $d_v$ and $d_k$ [80].

Once attention is computed for each head in Equation 2.28, the resulting representations $\text{head}_h$ are concatenated and multiplied by a weight matrix $\mathbf{W}^O$. This enables learning interactions between the outputs of each head before $\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ is fed as input into the next layer of the Transformer. After $N$ layers of self-attention, the output is used to decode a target representation in the original model [80].

### 2.3.4 Gaussian Mixture Models

While the normal distribution is the most common probability distribution, it has one drawback: It is unimodal. This severely limits its capabilities when modeling multimodal

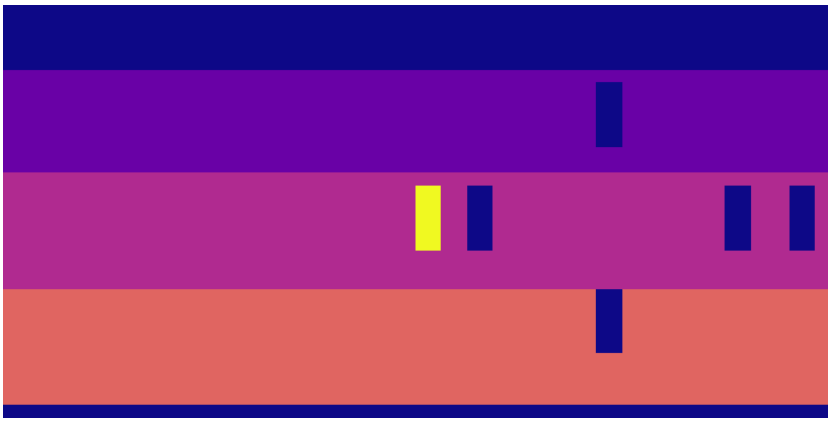


Figure 7: View of the agent facing an obstacle. Ideally, the agent would learn that evading the obstacle is possible on both sides through a multimodal distribution.

data. One solution to this problem is to mix multiple normal distributions in a superposition [8]. The resulting *mixture distribution* is a linear combination of normal distributions with the desirable property of being able to approximate arbitrary probability distributions [7].

Coming back to the case of RL, it can often be advantageous to have a stochastic policy $\pi(a|s)$ with multiple modes. Consider for instance the case in which a vehicle is positioned directly in front of an obstacle (see Figure 7 for example): It can avoid the obstacle to the left or to the right, which would ideally be modeled by a multimodal distribution. While a categorical distribution in discrete action spaces is inherently able to do so, a normally distributed policy would have to choose either side. This example provides the motivation for an introduction of GMMs in the following section, which are able to represent multimodal stochastic policies in continuous control settings.

Mathematically, a mixture of normal distributions is defined by the linear combination in Equation 2.29 [8]:

$$p(\mathbf{a}) = \sum_{k=1}^{K} \alpha_k \mathcal{N}(\mathbf{a}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \ . \tag{2.29}$$

Here the mixture coefficients $\alpha_k$ are scalar weights for the individual Gaussians making up the mixture. Each normal density $\mathcal{N}(\mathbf{a}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is called a component and has its own parameters. $\boldsymbol{\mu}_k$ is the mean vector of the distribution and $\boldsymbol{\Sigma}_k$ is the covariance matrix. The covariance matrix must be symmetric and positive semidefinite to specify a defined normal distribution. To form a valid mixture model, the coefficients have to be

non-negative and must sum to one as stated in conditions 2.30 [8]:

$$\sum_{k=1}^{K} \alpha_k = 1, \quad \alpha_k \geq 0 \ . \tag{2.30}$$

A *mixture density network* is a mixture model where the coefficients $\alpha_k(\mathbf{s})$ and the component parameters $\boldsymbol{\mu}_k(\mathbf{s})$ and $\boldsymbol{\Sigma}_k(\mathbf{s})$ are the output of a neural network given an input $\mathbf{s}$. It is thus a conditional probability distribution given $\mathbf{s}$ [7]. Note how with the right notation, the left-hand side of Equation 2.31 looks similar to the notation used for a policy $\pi_\theta(\mathbf{a}|\mathbf{s})$. Indeed, a mixture density network is able to parameterize a stochastic policy.

$$p(\mathbf{a}|\mathbf{s}) = \sum_{k=1}^{K} \alpha_k(\mathbf{s}) \mathcal{N}(\mathbf{a}|\boldsymbol{\mu}_k(\mathbf{s}),\, \boldsymbol{\Sigma}_k(\mathbf{s})) \ . \tag{2.31}$$

How does a neural network learn outputs that conform to the restrictions imposed by a GMM? A closer look at the requirements for the mixing coefficients $\alpha_k(\mathbf{s})$ in Equation 2.30 reveals that they are the same as for a probability distribution. It is thus possible to learn $\alpha_k(\mathbf{s})$ by a network with $k$ output neurons over which the Softmax activation function from Definition 2.18 is applied [8].

Learning the covariance matrix $\boldsymbol{\Sigma}_k(\mathbf{s})$ is usually done with a trick to improve numerical stability. It has two steps: First the log standard deviation is learned by the network. This learned value is then exponentiated to produce the real standard deviation value. Through learning the log standard deviation, the neural network can output both positive and negative values, increasing numerical stability. The following exponentiation returns a value greater than zero, making it a valid choice as standard deviation [71].

The generation of observations from a GMM follows a two-step procedure:

1. An index $k$ is drawn from $1, \ldots, K$. The probability of choosing index $k$ is determined by the corresponding mixing coefficient $\alpha_k(\mathbf{s})$.

2. An observation is sampled from the $k$-th mixture component. It is parameterized by the mean vector $\boldsymbol{\mu}_k(\mathbf{s})$ and the standard deviation $\boldsymbol{\Sigma}_k(\mathbf{s})$.

# 3 Related work

| | | **Action space** | |
|---|---|---|---|
| | | Discrete action space | Continuous action space |
| **Number of agents** | Single agent | **A** Example algorithms: *AlphaGo, AlphaZero, MuZero, SAVE* Example studies: [73, 75, 74, 70, 30] | **B** Example algorithms: *A0C, Continuous MuZero, Sampled MuZero* Example studies: [58, 89, 38] |
| | Multi agent | **C** Example algorithms: *Multiplayer AlphaZero* Example studies: [64] | **D** **Example algorithms: *This work*** **Example studies: This study** |

Table 1: Overview over prior research combining Reinforcement Learning (RL) and Monte Carlo tree search (MCTS). While the AlphaZero family of algorithms utilizes self-play during training, they only plan for a single agent at a time. Therefore they are considered single-agent algorithms in this overview.

AlphaGo and its monumental win in the show match versus Lee Sedol have received a tremendous amount of attention [73]. Naturally, a plethora of follow-up research continues to improve the original algorithm or extends it for use in other domains. Work by the same group for instance removes the need for human expert knowledge with AlphaGo Zero [75] and applies the combination of RL and MCTS to other games [74]. This results in the *AlphaZero* algorithm. Expert iteration is another extension to AlphaGo which has been developed independently of AlphaGo Zero [2]. In discrete action spaces, using Q-Learning instead of policy gradients shows improved performance [30]. Among follow-up works, the *MuZero* algorithm stands out since it removes the need for knowing the environment's transition dynamics. Instead, MuZero performs tree search in a latent-space model [70]. As the exact knowledge of the transition function is a very restrictive assumption, using an approximate model allows application to domains such as Atari [70]. An orthogonal direction for extensions is provided with Multiplayer AlphaZero, which successfully applies a modified algorithm to three-player games [64]. Multiplayer AlphaZero differs from this thesis in two ways: It considers a competitive setting instead of a cooperative one and does not learn interactions between agents.

A further stream of research focuses on developing combinations of RL and MCTS for continuous action spaces. A0C is the first of such works, extending AlphaZero to simple continuous control scenarios provided by OpenAI Gym [10]. It forms the basis for this thesis. Similar extensions have then been applied to MuZero, resulting in successful applications to classical control and simulated robot manipulation/locomotion tasks [89,

38].

[33] apply an AlphaZero-inspired algorithm to tactical decision making in autonomous driving. Compared to this work, they do not learn in a continuous action space. Five high-level actions are used instead which are subsequently mapped to continuous actions by a physics model. [33] also consider single-agent settings only and do not plan actions for other agents in a scenario[10]. A very similar approach is presented by [14]. Other works use model-free RL [53], combine RL with $A^*$ planning [91] or use RL together with game-theoretic reasoning [9]. A Multi-Agent Reinforcement Learning (MARL) approach to learning driving behavior is proposed by [4], from which the hybrid input representation in this thesis is derived.

When a multi-agent setting is considered, most works rely on learning a single policy that is used by all agents via parameter-sharing [4, 64]. Some algorithms do consider scenarios with a flexible number of agents. Deep sets [40] or graph networks [39] are able to learn interactions between different entities in an environment. They do however only learn behavior for a single decision-making agent. Another approach is to use Recurrent Neural Networks (RNNs) to process trajectories from multiple agents [22]. While an RNN is able to aggregate information from different entities, its inherent sequentiality implies that some agents have more information at their disposal than others.

Transformers [80] have emerged as the dominant architecture for processing sequential input, with applications in natural language [19] and trajectory forecasting [25]. [87] have proposed an attention-based network architecture which is conceptually similar to this thesis. However, they do not use a planner to improve the policy and instead rely on model-free RL.

Lastly, this thesis builds on an MCTS approach for cooperative decision making in autonomous driving [46]. Note that [46] plan in a continuous action space compared to previous approaches which just learn high-level primitives [52]. Using a neural network to guide the MCTS has already been investigated [47]. In contrast to this thesis, they rely on supervised learning from expert trajectories instead of RL to train the network. Their architecture also uses a Multilayer Perceptron (MLP) and thus produces a fixed-size output that does not scale with the number of agents within a scenario.

[10] While other agents exist in their scenarios, they do not learn and are controlled by a driver model.

# 4 Concept

Chapter 2 has laid the theoretical foundations needed to describe the concept of this thesis. In the first section, the environment is defined with a focus on three aspects: The state space $\mathcal{S}$, which is defined by a hybrid input representation using numerical and image input. Then the reward function is described, before the action space $\mathcal{A}$ is defined. Next is a derivation of the tanh-squashed Normal distribution, which is motivated by the need to constrain actions within the physical boundaries of a vehicle. In the following section, an objective function for multi-agent scenarios is obtained by extending a formulation of the AlphaZero loss for continuous action spaces. How a Transformer-based network architecture can be used to handle scenarios with a flexible number of agents comes next. The penultimate section describes a guided Monte Carlo tree search (MCTS) procedure, where rollouts are truncated by network evaluations. Finishing up the chapter is the concept of a training algorithm for the system.

## 4.1 Environment

Section 2.1.2 has introduced the theoretical framework of a decentralized Markov Decision Process (dec-MDP) as a tuple $\langle \Upsilon, \mathcal{S}, \mathcal{A}, P, \mathcal{R}, \gamma \rangle$. Now it is time to elaborate further on how some of its components are implemented concretely. Of particular interest are the state space $\mathcal{S}$, the action space $\mathcal{A}$ and the reward function $\mathcal{R}(\cdot)$. Section 4.1.1 details a hybrid input representation consisting of a local visual map of an agent together with numerical information about all other vehicles in a scenario. It can be processed efficiently by a neural network. This characterizes the state space $\mathcal{S}$. Next is the definition of the reward function $\mathcal{R}(\cdot)$, before a brief description of the action space $\mathcal{A}$ follows. How the chosen actions are mapped to vehicle trajectories finishes the chapter.

### 4.1.1 Input Representation

In general, a state $s \in \mathcal{S}$ in the dec-MDP is given by a scenario. It in turn is defined as a road with multiple lanes, a set of active agents executing actions and optionally a set of stationary vehicles (obstacles) on the road. Surrounding the road is non-drivable area. The number of lanes, active agents and obstacles as well as their position is flexible and different for each scenario. Figure 8 shows the map for such a scenario, where two agents have to drive through a bottleneck. In order to avoid a crash, one vehicle will have to slow down and the other must accelerate while both merge into the center. This maneuver requires cooperation between both agents [46].

How can such a scenario be encoded in a way that is both efficient to process for a neural network as well as close to real-world autonomous driving? To answer this question,

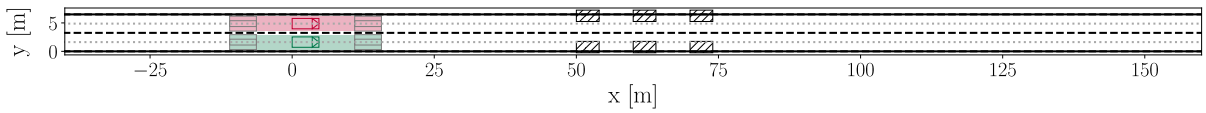



Figure 8: Example of a cooperative driving scenario where two agents have to avoid a crash while driving through a narrow bottleneck.

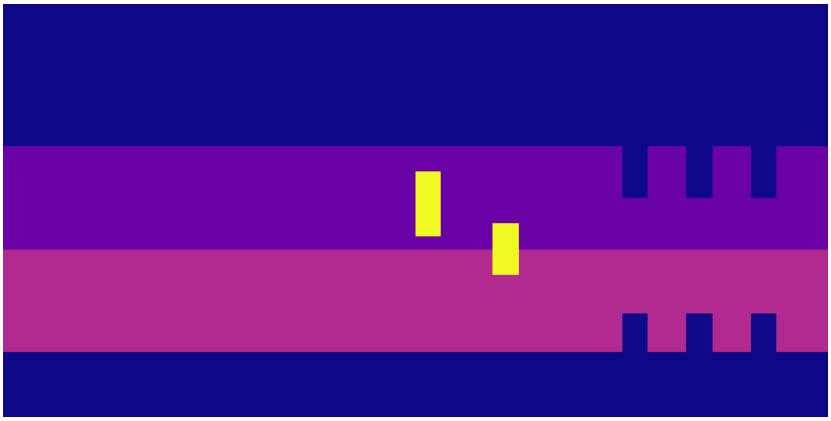


Figure 9: Example of an agent's view from scenario 08 (Figure 8). Active agents are depicted in yellow, whereas obstacles (stationary vehicles) are represented as non-drivable area on the lane. The lanes are colored in different shades of purple.

a hybrid input representation consisting of both a visual map and a set of numerical inputs is chosen [4, 47]. The map is a top down view of the agent's region of interest in the scenario that is defined by its sensor range. It covers an area of $64m$ forwards and backwards as well as $6.4m$ to each side from the center of a vehicle's back axle. The total viewable area is thus $128m \times 12.8m$. This is encoded into a two channel image of $2 \times 32 \times 64$ pixels, where the first channel is an agent map that solely encodes the active agents in the scenario. The second channel represents lanes, obstacles and non-drivable area. Obstacles are using the same numerical values as non-drivable area. This allows for a more efficient representation using only two channels instead of three. Since an agent receives the same reward for crashing into an obstacle as it does for driving off the road, no distortion between its perception and its immediate reward occurs. This is discussed further in Section 4.1.2 (also see Appendix F). Figure 9 shows an example of an agent's view in scenario 08.

Lastly, it is important to note that the resolution of the visual map is coarse due to

performance constraints[11]. Particularly for longitudinal maneuvers, 2 meters per pixel might not seem granular enough. A consideration however is that lateral maneuvers usually require more precision, for instance when avoiding an obstacle. This is still allowed by a finer lateral resolution of 0.4 meters per pixel.

Visual input representations have the advantage that they are flexible and environment-agnostic. As described above, there is however a trade-off between representational capacity and computational efficiency. Numerical inputs on the other hand are less computationally expensive and allow for detailed information but need to be tailored to a specific task. Since the goal of this thesis is learning cooperative driving behavior, it is sensible to additionally encode agent information numerically to foster interaction. The state of a single agent $i$ at time step $t$ is composed of a static state vector $\mathbf{n}_i^{\text{static}}$ and dynamic state vectors $\mathbf{n}_i^{\text{dynamic}}(t)$ [47]. The static component of the state refers to agent information that does not change over the course of a scenario whereas the dynamic component changes with each time step $t$. Concatenation of the current dynamic state vector with the past seven dynamic state vectors and the static state vector yields the full numerical state of an agent. It is represented in Equation 4.1, where $\oplus$ represents the concatenation operation. While combining past information into a Markovian state might seem redundant at first, it produces little computational overhead due to the efficiency with which numerical values are processed. At the same time it ensures that agents have access to all needed information [47]. Similar techniques like "*frame-stacking*", where the past $m$ images are combined into a single stacked image, are also common in other Reinforcement Learning (RL) applications [57].

$$\begin{aligned}
\mathbf{n}_i^{\text{dynamic}}(t) &= \Big(x_i(t),\, y_i(t),\, \dot{x}_i(t),\, \dot{y}_i(t),\, \ddot{x}_i(t),\, \ddot{y}_i(t),\, \phi_i(t)\Big) \\
\mathbf{n}_i^{\text{static}} &= \Big(\dot{x}_i^{\text{desire}},\, l_i^{\text{desire}}, v_i^{\text{width}}, v_i^{\text{length}}\Big) \\
\mathbf{n}_i(t) &= \mathbf{n}_i^{\text{dynamic}}(t-7) \oplus \mathbf{n}_i^{\text{dynamic}}(t-6) \oplus \ldots \oplus \mathbf{n}_i^{\text{dynamic}}(t) \oplus \mathbf{n}_i^{\text{static}} \ . \qquad (4.1)
\end{aligned}$$

More concretely:

- $(x_i(t),\, y_i(t))$ is the normalized relative position of agent $i$.
- $(\dot{x}_i(t),\, \dot{y}_i(t))$ is the normalized relative velocity of agent $i$ in lateral and longitudinal direction.
- $(\ddot{x}_i(t),\, \ddot{y}_i(t))$ is the normalized relative acceleration of agent $i$ in each direction.
- $\phi_i(t)$ is agent $i$'s normalized relative heading in radian.
- $\dot{x}_i^{\text{desire}}$ is the desired velocity of agent $i$ in longitudinal direction.

[11] Profiling the executable reveals that the program spends around 9.5% of its runtime generating the maps. This can quickly jump to over 40% for higher resolutions.

- $l_i^{\text{desire}}$ is the desired lane of agent $i$.
- $(v_i^{\text{width}}, v_i^{\text{length}})$ are the vehicle dimensions of agent $i$.

It is important to highlight that dynamic state information is given *relative to the ego agent*. That is, relative to the agent currently making the decision. During each scenario run all agents sense each other with limited sensor precision outside of the $128m \times 12.8m$ region of interest. The limited precision is modeled by bounding relative values on the interval $[-1, 1]$. To give an example, suppose $x_1 = 10$ is the longitudinal position of agent one while $x_2 = 138$ is the longitudinal position of agent two. Then the relative longitudinal position of agent two with respect to agent one is $x_2^{\text{rel}} = 128$. Normalizing by the maximum frontal sensor range of agent one yields $x_2^{\text{rel, norm}} = 128/64 = 2$. However, since agent two has been outside of agent one's sensor range, $x_2^{\text{rel, norm}}$ is capped at 1. This reflects the limited precision of sensors over longer ranges. Intuitively, agent one can sense another vehicle approaching but its distance is too far to take exact measurements.

The full numerical state of an agent $i$ within a scenario is given as the matrix produced by stacking the numerical states $\mathbf{n}_i(t)$ of all agents $\Upsilon$ within the scenario:

$$\mathbf{s}_i^{\text{num}}(t) = \begin{pmatrix} \mathbf{n}_0(t) \\ \mathbf{n}_1(t) \\ \vdots \\ \mathbf{n}_\Upsilon(t) \end{pmatrix} . \tag{4.2}$$

It is worth repeating that all numerical states in $\mathbf{s}_i^{\text{num}}(t)$ are relative to the current agent $i$. Now that the state space $\mathcal{S}$ of the environment is defined, the next section can introduce the reward function $\mathcal{R}$.

### 4.1.2 Reward function

The reward function of an RL environment is key in eliciting the desired behavior from an agent. Actions that are desired should be reinforced while actions leading to undesired or even harmful behavior should be punished. Ideally, the reinforcement comes immediately after a specific action, making the reward function *dense*. The opposite of dense rewards are *sparse* rewards. An example of a sparse reward environment is the game of chess: Here an agent only receives a positive or negative reward upon completion of the game but not for any intermediate actions. Therefore the following paragraphs specify a dense reward function composed of three parts (see Equation 4.3) [46]:

1. A component which punishes actions that are too jerky, thereby encouraging smooth driving maneuvers.

2. A term that rewards an agent depending on its distance to a desired target state , which is specified by a target velocity $\dot{x}_i^{\text{desire}}$ and a target lane $l_i^{\text{desire}}$.

3. A validity component punishing actions that lead to either collisions or driving off the road.

The components outlined above define the ego reward for each agent. It is a sum of three elements [46]:

$$r_i = r_i^{\text{action}} + r_i^{\text{state}} + r_i^{\text{valid}} \ . \tag{4.3}$$

Its first term is the *action reward* $r_i^{\text{action}}$. It is always negative and can be interpreted as the cost of performing a driving maneuver between two time steps $t_0$ and $t_1$ [46]. The action reward consists of the following components

$$r_i^{\text{action}} = w_{LC}(l_i(t_1) - l_i(t_0))^2 + w_{AX}\int_{t_0}^{t_1}(\ddot{x}_i(t))^2 dt + w_{AY}\int_{t_0}^{t_1}(\ddot{y}_i(t))^2 dt + w_{IA} \ . \tag{4.4}$$

The first term $(l_i(t_1) - l_i(t_0))^2$ punishes lane changes while the second $\int_{t_0}^{t_1}(\ddot{x}_i(t))^2 dt$ and third term $\int_{t_0}^{t_1}(\ddot{y}_i(t))^2 dt$ aim at minimizing the acceleration in longitudinal and lateral direction. The relative importance of each component can be adjusted via the weights $w_{LC}, w_{AX}$ and $w_{AY}$. $w_{IA}$ is a constant that is added if the generated action is invalid, for instance when the vehicle's steering angle is past its maximum or the agent exceeds the maximum speed limit [46].

Next is the *state reward* $r_i^{\text{state}}$ aimed at minimizing the distance between the current state and a desired target state of the agent [46]:

$$\begin{aligned} r_i^{\text{state}} = & \underbrace{2w_{VD} \cdot \exp\Big(-0.00745 \cdot (\dot{x}_i^{\text{desire}} - \dot{x}_i(t))^2\Big) - w_{VD}}_{\text{Velocity component}} \\ & + \underbrace{w_{LD} - w_{LCD}\Big|l_i(t) - l_i^{\text{desire}}\Big|}_{\text{Lane deviation component}} \\ & + \underbrace{w_{LCD} \cdot \exp\left(-5.0\frac{c_i^l(t) - y_i(t)}{2 \cdot W_l}\right)}_{\text{Lane center deviation component}} \ . \end{aligned} \tag{4.5}$$

Subscripts for the time step $t$ in Equation 4.5 are dropped as quantities are only evaluated at the current step $t$. The weight $w_{VD}$ is used to scale the velocity deviation term. It is defined using an exponential function $\exp\Big(-0.00745 \cdot (\dot{x}_i^{\text{desire}} - \dot{x}_i(t))^2\Big)$ with a quadratic exponent [46]. If the agent's current velocity $\dot{x}_i(t)$ reaches its desired velocity $\dot{x}_i^{\text{desire}}$, the whole exponential reduces to the factor 1, thus maximizing the velocity deviation reward as $w_{VD}$.

The lane deviation component is composed of a positive weight $w_{LD}$, from which the weighted absolute lane deviation $w_{LCD}\Big|l_i(t) - l_i^{\text{desire}}\Big|$ is subtracted. Here $w_{LCD}$ is the lane

center deviation weight. If for instance the agent's target lane is $l_i^{\text{desire}} = 3$ and its current lane $l_i(t) = 1$, then $w_{LCD} \cdot 2$ is subtracted [46].

Lastly, the deviation from the lane center is rewarded using an exponential term similar to the velocity deviation term. It is subsequently weighted by the lane center deviation weight $w_{LCD}$. If the vehicle is exactly on the center line of the lane it is currently driving on (denoted as $c_i^l(t)$), the denominator reduces to 0 and the whole center deviation reward becomes $w_{LCD}$. The term in the numerator is defined as two times the lane width $W_l$ in the current scenario[12] [46].

Combining everything above, the maximum state reward achievable is the sum of the three weights $w_{VD} + w_{LD} + w_{LCD}$. This is accomplished if the agent drives exactly center on its target lane and with the desired velocity. The exact values of the weights $w_{VD}, w_{LD}$ and $w_{LCD}$ may be adjusted individually for each scenario.

The last component of the reward function in Equation 4.3 is the *validation reward* [46]:

$$r_i^{\text{valid}} = \begin{cases} w_{IS}, & \text{if invalid state} \\ w_C, & \text{if collision} \\ 0, & \text{else} \end{cases} . \tag{4.6}$$

$w_{IS}$ is a constant added if the action leads to an invalid state. That is, if the vehicle leaves the road and is in non-drivable area. The reward $w_C$ is given if the action results in a collision with another vehicle or obstacle. When neither of the above occurs, the chosen action is deemed valid and nothing is added. Usually the weights for invalid states and collisions are set to large negative constants (e.g. $w_{IS} = w_C = -1000$) as both states are highly undesirable [46].

The three components described above fully define the ego reward for an agent. While the validation reward is straightforward, there is a push-pull dynamic between the action reward and the state reward: On one hand, an agent wants to reach its target state as quickly as possible through the state reward. One the other hand, abrupt maneuvers are punished more severely by the action reward. Together they form a balance in which an agent tries to achieve its goal state quickly but also economically [46].

To achieve cooperation and fully specify the reward function $\mathcal{R}$ of the environment, one last component is missing. It is the cooperative reward. Given a cooperation factor $\lambda_i$, the cooperative reward $r_i^{\text{coop}}$ of an agent $i$ is defined as [46]:

$$r_i^{\text{coop}} = r_i + \lambda_i \sum_{j=0, j\neq i}^{\Upsilon} r_j \tag{4.7}$$

Here $\sum_{j=0,j\neq i}^{\Upsilon} r_j$ are the summed ego rewards of all other agents. The cooperation factor

[12] Lanes are assumed to have equal width in all scenarios.

$\lambda_i \in [0, 1]$ determines the disposition of agent $i$ to cooperate[13]. This scaled sum of other agent's rewards added to the own ego reward $r_i$ finally specifies the reward function of the environment $\mathcal{R}$.

Exact values for all constants from this section are given in Appendix F. The last critical component to be defined in the environment is the action space $\mathcal{A}$, which is outlined in the next section.

### 4.1.3 Action space

At each time step $t$, an agent $i$ can choose to change its longitudinal velocity $\Delta v_i^{\text{longitudinal}}$ as well as its lateral velocity $\Delta v_i^{\text{lateral}}$ [46]. This results in a 2-dimensional, continuous action space $\mathcal{A}$. To interpolate between two discrete time steps $t_0$ and $t_1$, a jerk-minimizing trajectory is generated by solving for the coefficients $a_p, b_p$ of fifth order polynomials [46]:

$$x_i(t) = \sum_{p=0}^{5} a_p t^p, \quad y_i(t) = \sum_{p=0}^{5} b_p t^p \tag{4.8}$$

using the following boundary conditions

$$\ddot{x}_i(t_1) = 0 \tag{4.9}$$

$$\ddot{y}_i(t_1) = 0 \tag{4.10}$$

$$\dot{y}_i(t_1) = 0 \tag{4.11}$$

$$\dot{x}_i(t_1) = \dot{x}_i(t_0) + \Delta v_i^{\text{longitudinal}} \tag{4.12}$$

$$y_i(t_1) = y_i(t_0) + \Delta v_i^{\text{lateral}} \tag{4.13}$$

$$x_i(t_1) = \frac{\dot{x}_i(t_0) + \dot{x}_i(t_1)}{2}(t_1 - t_0) \ . \tag{4.14}$$

Constraints 4.9, 4.10 and 4.11 specify that the vehicle must not accelerate in its target state and that there is no lateral acceleration. The target state is defined by the desired velocity (Equation 4.12) in longitudinal direction and the desired vehicle position in lateral direction (Equation 4.13). Now only the distance covered in longitudinal direction is missing, which is described by Boundary 4.14 [46].

Lastly, the produced actions must conform to the physical limitations of the controlled vehicle. This is ensured by bounding $\Delta v_i^{\text{longitudinal}}$ and $\Delta v_i^{\text{lateral}}$ on the interval $[-5, 5]$. How these action bounds are enforced for actions sampled from a Gaussian Mixture Model (GMM) is described in the next section.

[13] It is possible to specify different cooperation factors for the agents. In practice, the cooperation factor is the same. Refer to Appendix F for details.

## 4.2 Enforcing action bounds

Stochastic policies in continuous control settings are commonly represented by a diagonal normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Each dimension in the action space is represented by an independent normal distribution, resulting in a diagonal covariance matrix $\boldsymbol{\Sigma}$. The network is then trained by backpropagating through the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ of the distribution. Prominent algorithms using this technique to specify a stochastic $\pi_\theta(\mathbf{a}|\mathbf{s})$ are for example Trust Region Policy Optimization (TRPO) [71] and Proximal Policy Optimization (PPO) [72].

In most RL tasks, the action space is bounded in some form. Common benchmarks like OpenAI Gym [10] usually restrict actions to be in $[-1, 1]$. In real-world tasks like autonomous driving or robotics the action space is limited by the capabilities of the physical system. In such cases a normal distribution has the undesirable characteristic that its support is unbounded.

The most crude solution to this problem is cutting the distribution at the desired bounds. This may however introduce biases that negatively affect performance [15]. Another solution is to select a distribution with finite support. [15] and [58] propose to use a transformed Beta distribution. While the transformed Beta distribution has theoretical appeal due to its flexibility and finite support, training a policy for this work has not succeeded using it.

A more elegant and stable solution has been proposed in the Soft Actor-Critic (SAC) algorithm [29]. Here $\pi_\theta(\mathbf{a}|\mathbf{s})$ is parameterized by a normal distribution which is squashed by the tanh function to be between $(-1, 1)$. Subsequently, it is re-scaled by a constant factor to fit the action bounds. Figure 10 shows the flexibility of such a distribution for different standard deviations. Since the tanh-squashed normal distribution is used for the approach outlined by this thesis, it is derived in the following paragraph.

Let $\mathbf{u} \in \mathbb{R}^D$ be a normally distributed random variable and $\mu(\mathbf{u}|\mathbf{s})$ be the density from which $\mathbf{u}$ is drawn. Note that the support of $\mu(\mathbf{u}|\mathbf{s})$ is unbounded. The goal is to find the density of the transformed random variable $\mathbf{a} = c\tanh(\mathbf{u})$ with $c \in \mathbb{R}^+ \setminus \{0\}$ and express it in terms of $\mu(\mathbf{u}|\mathbf{s})$. The Cumulative Distribution Function (CDF) of the transformed random variable is given as

$$\int_{-c}^{c} \pi(\mathbf{a}|\mathbf{s})\, d\mathbf{a} \ , \tag{4.15}$$

which is desired to be expressed in terms of $c\tanh(\mathbf{u})$. The following proposition expresses the Probability Density Function (PDF) of the transformed random variable in terms of the untransformed density $\mu(\mathbf{u}|\mathbf{s})$.

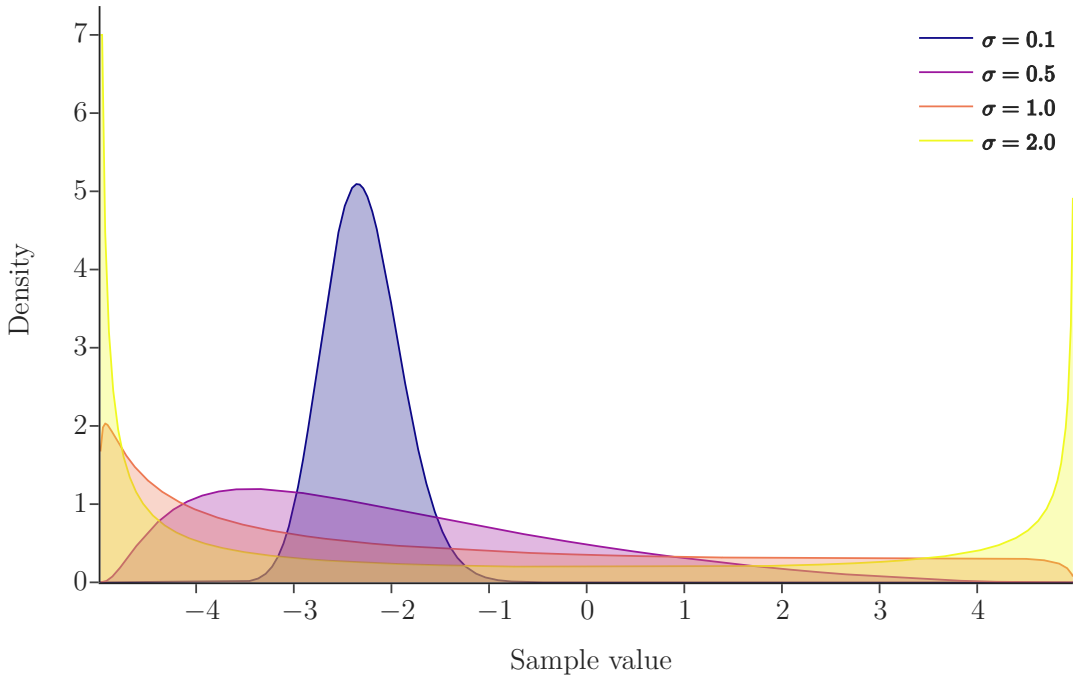



Figure 10: Density function of the squashed and rescaled normal distribution for $\mu = -0.5$ and different standard deviations. Samples are bounded in $[-5, 5]$.

**Proposition 1.** *The PDF of the transformed density can be expressed as*

$$\pi(\mathbf{a}|\mathbf{s}) = \mu(\mathbf{u}|\mathbf{s}) \left| \det \frac{d\mathbf{a}}{d\mathbf{u}} \right|^{-1} , \tag{4.16}$$

*with*

$$\left| \det \frac{d\mathbf{a}}{d\mathbf{u}} \right| = c^D \prod_{i=1}^{D} 1 - \tanh^2(u_i) \ . \tag{4.17}$$

*Proof.* See Appendix B. □

Now that the transformed density $\pi(\mathbf{a}|\mathbf{s})$ is known, the next step is to derive its log-probabilities since they are needed to calculate the gradient of the objective function in Sections 4.3.2 and 4.3.3. They can be obtained by plugging in the squashed normal PDF

and iteratively applying logarithm rules until Expression 4.18 is reached[14]:

$$\begin{aligned}\log \pi(\mathbf{a}|\mathbf{s}) &= \log \left( \mu(\mathbf{u}|\mathbf{s}) \left| \det \frac{d\mathbf{a}}{d\mathbf{u}} \right|^{-1} \right) \\ &= \log \left( \frac{\mu(\mathbf{u}|\mathbf{s})}{c^D \prod_{i=1}^{D} 1 - \tanh^2(u_i)} \right) \\ &= \log \mu(\mathbf{u}|\mathbf{s}) - D \log c - \sum_{i=1}^{D} \log \left( 1 - \tanh^2(u_i) \right) . \end{aligned} \tag{4.18}$$

A closer look at the term $D \log c$ reveals that it is only dependent on the dimensionality of the action space $D$ and the scaling term $c$. Therefore it does not contribute to gradients with respect to $\mathbf{u}$. Dropping it yields the log-likelihood formula from the SAC paper [28]:

$$\log \pi(\mathbf{a}|\mathbf{s}) = \log \mu(\mathbf{u}|\mathbf{s}) - \sum_{i=1}^{D} \log \left( 1 - \tanh^2(u_i) \right) . \tag{4.19}$$

Expression 4.19 is mathematically correct, but numerically unstable. To see why, note that tanh is already very close to one for values $u_i \geq 3$. Due to floating point imprecision, it can easily occur that $\tanh(u_i) = 1$ for larger values of $u_i$. This results in taking the log of zero and NaNs during training. Luckily, Formula 4.19 can be re-expressed into a mathematically equivalent, but more stable version. Recall the definition of the hyperbolic secant as $\operatorname{sech}(u_i) = \frac{2e^{-u_i}}{e^{-2u_i}+1}$. Also note that $\operatorname{sech}^2(u_i) = 1 - \tanh^2(u_i)$. Some algebra and application of the logarithm rules yield[15]:

$$\begin{aligned} -\sum_{i=1}^{D} \log \left( 1 - \tanh^2(u_i) \right) &= -\sum_{i=1}^{D} \log \left( \operatorname{sech}^2(u_i) \right) \\ &= -2 \cdot \sum_{i=1}^{D} \Big( \log 2 - u_i - \log(e^{-2u_i} + 1) \Big) \\ &= -2 \cdot \sum_{i=1}^{D} \Big( \log 2 - u_i - \operatorname{Softplus}(-2u_i) \Big) , \end{aligned} \tag{4.20}$$

where Softplus is a common activation function in machine learning defined as $\operatorname{Softplus}(x) = \log(1 + e^x)$.

Extending the squashed normal distribution defined above to a squashed GMM is conceptually easy. Generating observations follows the same procedure as sampling from a regular GMM (see Section 2.3.4). First a component is sampled before subsequently an observation is drawn from that component. As all components in a squashed GMM are squashed normal distributions, the observation conforms to the specified action bounds.

So far this thesis has discussed the environment and how an algorithm can comply with its restrictions. The next section explains how such an algorithm may be trained.

[14] The detailed steps can be found in Appendix C.

[15] For a derivation see Appendix D.

## 4.3 Objective Function

This chapter details the procedure for policy improvement in continuous action spaces. It is guided by finding answers to two questions:

1. How can a continuous policy specified by a tanh-squashed normal distribution be improved via MCTS?
2. How can the derived objective function be extended to a multi-agent setting?

The first two sections then proceed to derive a loss function based on the MCTS visitation count distribution for improving a continuous policy in the single-agent case. Their contents follow the approach in the A0C paper [58]. The last section provides a simple — yet effective — method for extending the objective function to a multi-agent setting.

### 4.3.1 Training target

The MCTS outputs a set of actions $A^0$ and visitation counts $N^0$ at the root node. Unlike the training in discrete settings, where the normalized visitation counts can be used as training target [74], the count distribution needs to be transformed into a continuous target first. The key assumption is that the density at the root actions $\mathbf{a}_j \in A^0$ should be proportional to the visitation counts $n(\mathbf{s}, \mathbf{a}_j) \in N^0$. This allows the definition of the following training target [58]:

$$\hat{\pi}(\mathbf{a}|\mathbf{s}) = \frac{n(\mathbf{s}, \mathbf{a}_j)^\tau}{Z(\mathbf{s}, \tau)} \ , \tag{4.21}$$

where $\tau \in \mathbb{R}^+$ is a positive temperature parameter that scales the importance of the visitation counts. $Z(\mathbf{s}, \tau)$ is an *action-independent* normalization term. It is noteworthy that Equation 4.21 does not specify a proper density as it is not defined between its support points. However, the following section shows that this fact becomes irrelevant as the objective function only considers the loss at the support points [58].

### 4.3.2 Single-agent objective

The single-agent objective function for training the network in a continuous setting consists of three components [58]:

$$\mathcal{L}_\theta(\mathbf{a}, \mathbf{s}) = \mathcal{L}_\theta^{\text{policy}}(\mathbf{a}, \mathbf{s}) - \alpha \mathcal{L}_\theta^{\text{H}}(\mathbf{a}, \mathbf{s}) + \mathcal{L}_\theta^{V}(\mathbf{s}) \ , \tag{4.22}$$

where $\mathcal{L}_\theta^{\text{policy}}$ is the *policy loss*, $\mathcal{L}_\theta^{\text{H}}$ is an *entropy regularization* term and $\mathcal{L}_\theta^{V}$ is the *value loss*. In the following, each of these three terms is examined in detail.

The aim of the policy loss $\mathcal{L}_\theta^{\text{policy}}$ is to move the network distribution $\pi_\theta(\mathbf{a}|\mathbf{s})$ closer to the improved MCTS distribution $\hat{\pi}(\mathbf{a}|\mathbf{s})$. Intuitively, it would make sense to use a measure of how different $\pi_\theta(\mathbf{a}|\mathbf{s})$ is from $\hat{\pi}(\mathbf{a}|\mathbf{s})$ to compute the loss. The first choice among such measures in RL is the Kullback-Leibler divergence [45]. This is indeed how the policy loss is specified [58]:

$$\mathcal{L}_\theta^{\text{policy}}(\mathbf{a},\mathbf{s}) = \mathrm{D}_{KL}\Big(\pi_\theta(\mathbf{a}|\mathbf{s})\Big|\Big|\hat{\pi}(\mathbf{a}|\mathbf{s})\Big) \tag{4.23}$$

$$= \mathbb{E}_{\mathbf{a}\sim\pi_\theta(\mathbf{a}|\mathbf{s})}\Big[\log\pi_\theta(\mathbf{a}|\mathbf{s}) - \log\hat{\pi}(\mathbf{a}|\mathbf{s})\Big]\ . \tag{4.24}$$

Plugging in $\hat{\pi}(\mathbf{a}|\mathbf{s}) = \frac{n(\mathbf{s},\mathbf{a})^\tau}{Z(\mathbf{s},\tau)}$ from Equation 4.21 and simplifying using logarithm rules, the gradient of the policy loss can be expressed as[16] [58]:

$$\begin{aligned}\nabla_\theta\mathcal{L}_\theta^{\text{policy}}(\mathbf{a},\mathbf{s}) &= \nabla_\theta\mathbb{E}_{\mathbf{a}\sim\pi_\theta(\mathbf{a}|\mathbf{s})}\Big[\log\pi_\theta(\mathbf{a}|\mathbf{s}) - \log\hat{\pi}(\mathbf{a}|\mathbf{s})\Big] \\ &= \nabla_\theta\mathbb{E}_{\mathbf{a}\sim\pi_\theta(\mathbf{a}|\mathbf{s})}\Big[\log\pi_\theta(\mathbf{a}|\mathbf{s}) - \log\Big(\frac{n(\mathbf{s},\mathbf{a})^\tau}{Z(\mathbf{s},\tau)}\Big)\Big] \\ &= \nabla_\theta\mathbb{E}_{\mathbf{a}\sim\pi_\theta(\mathbf{a}|\mathbf{s})}\Big[\log\pi_\theta(\mathbf{a}|\mathbf{s}) - \tau\log n(\mathbf{a},\mathbf{s}) + \log Z(\mathbf{s},\tau)\Big]\ .\end{aligned} \tag{4.25}$$

At this point it is time for a closer look at Formula 4.25. First note that the goal is to calculate the gradient with respect to $\theta$ of an expectation over the policy distribution $\pi_\theta(\mathbf{a}|\mathbf{s})$. However, the term inside the expectation also contains the log-visitation counts $\log n(\mathbf{a},\mathbf{s})$. They implicitly depend on the policy via action selection, but also on the environment state $\mathbf{s}$. The states $\mathbf{s}$ follow an unknown distribution which cannot be differentiated [77]. Luckily, the REINFORCE-trick allows the transformation of Equation 4.25 into one from which gradients can be taken [86, 58]. It does so by interpreting the term $\log\pi_\theta(\mathbf{a}|\mathbf{s}) - \tau\log n(\mathbf{a},\mathbf{s}) + \log Z(\mathbf{s},\tau)$ as a score function. This allows transforming the gradient of an expectation to an *expected gradient*[17]. Equation 4.26 can now be estimated via sampling.

$$\nabla_\theta\mathcal{L}_\theta^{\text{policy}}(\mathbf{a},\mathbf{s}) = \mathbb{E}_{\mathbf{a}\sim\pi_\theta(\mathbf{a}|\mathbf{s})}\Big[\Big(\log\pi_\theta(\mathbf{a}|\mathbf{s}) - \tau\log n(\mathbf{a},\mathbf{s}) + \log Z(\mathbf{s},\tau)\Big)\nabla_\theta\log\pi_\theta(\mathbf{a}|\mathbf{s})\Big]\ . \tag{4.26}$$

To further simplify Expression 4.26, the term $\log Z(\mathbf{s},\tau)$ can be dropped since it is action-independent (see previous section) and thus does not depend on $\theta$ [58]. Additionally, the expectation over the whole policy distribution $\mathbf{a}\sim\pi_\theta(\mathbf{a}|\mathbf{s})$ can now be replaced by an expectation over the empirical distribution of the actually chosen actions $\mathbf{a}_c\sim\mathcal{D}_\mathbf{s}$ in

[16] The subscript $j$ from $\mathbf{a}_j$ has been omitted to improve readability.

[17] More generally, the REINFORCE-trick states that $\nabla_\theta\mathbb{E}_{x\sim p(x|\theta)}[f(x)] = \mathbb{E}_{x\sim p(x|\theta)}[f(x)\nabla_\theta\log p(x|\theta)]$, where $f(x)$ is a black box score function. $p(x|\theta)$ is a known distribution parameterized by $\theta$, which is the parameter for which the gradient should be taken [86, 58].

each state $\mathbf{s} \sim \mathcal{D}$ currently stored in the replay buffer $\mathcal{D}$ [58]. This now yields the final expression for the gradient of the policy loss with respect to the network parameters $\theta$:

$$\nabla_\theta \mathcal{L}_\theta^{\text{policy}}(\mathbf{a}, \mathbf{s}) = \mathbb{E}_{\mathbf{s}\sim\mathcal{D}, \mathbf{a}_c \sim \mathcal{D}_\mathbf{s}} \Big[ \underbrace{\Big( \log \pi_\theta(\mathbf{a}|\mathbf{s}) - \tau \log n(\mathbf{a}_c, \mathbf{s}) \Big)}_{\text{Scaling factor for the gradient}} \nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) \Big] \ . \tag{4.27}$$

The second term $-\alpha\mathcal{L}_\theta^{\text{H}}$ in the objective function is an entropy maximization term. Augmenting the policy loss with this regularization term has several benefits. First, it prevents the policy from collapsing if all sampled actions are very close to each other [58]. Second, it improves exploration while at the same time still reducing the likelihood of obviously bad actions, leading to faster training speed [28]. Lastly, the policy is able to learn behavior with multiple modes more easily [28]. The entropy objective is defined as:

$$\mathcal{L}_\theta^{\text{H}}(\mathbf{a}, \mathbf{s}) = H(\pi_\theta(\mathbf{a}|\mathbf{s})) = -\int \pi_\theta(\mathbf{a}|\mathbf{s}) \log \pi_\theta(\mathbf{a}|\mathbf{s}) \, da \ . \tag{4.28}$$

However, since there is no closed form solution for calculating the entropy of GMMs [37], it has to be estimated during training through the log-probabilities of the policy distribution.

The last part of the objective function is the value loss $\mathcal{L}_\theta^V$. It is defined as the mean-squared error between the value estimate produced by the network $V_\theta(\mathbf{s})$ and a value target $\hat{V}(\mathbf{s})$ [58]:

$$\mathcal{L}_\theta^V(\mathbf{s}) = \mathbb{E}_{\mathbf{s}\sim D} \left[ \left( V_\theta(\mathbf{s}) - \hat{V}(\mathbf{s}) \right)^2 \right] . \tag{4.29}$$

Clearly the value target is an improved value estimate obtained through the MCTS. It is however not obvious at first how it should be calculated. Using the state-action values at the root node $Q(\mathbf{s}_0, \mathbf{a})$, one could weigh the value of each action by its normalized visitation count and use the sum as target [58, 85]. However, since the number of MCTS iterations is finite, this way of calculating $\hat{V}(\mathbf{s})$ would introduce a bias induced by exploratory moves [85]. This work therefore uses the same value targets as A0C [58], defined as:

$$\hat{V}(\mathbf{s}_0) = \max_{\mathbf{a}} Q(\mathbf{s}_0, \mathbf{a}) \ . \tag{4.30}$$

Selecting the value of the best action at the root node $\mathbf{s}$ has several desirable characteristics. First, it disregards exploratory moves. This makes it consistent with the final action selection in the MCTS if the action with the highest value is chosen. Second, it is conceptually easy as well as efficient to calculate.

### 4.3.3 Multi-agent objective

So far, the objective function in Expression 4.22 only covers the single-agent setting. To increase clarity in the following section, it can be rewritten as

$$\mathcal{L} = \frac{1}{|A|} \sum_{a=1}^{A} \mathcal{L}_a^{\text{policy}} - \alpha\mathcal{L}_a^{\text{H}} + \mathcal{L}_a^V \ , \tag{4.31}$$

where an average over the number of actions $A$ the agent selected in the batch is taken to reduce the loss to a scalar. Note that for the sake of readability and succinctness, the subscript denoting the network parameters $\theta$ has been dropped and the function arguments $(\mathbf{a}, \mathbf{s})$ are omitted.

The most straightforward extension to multi-agent scenarios is to average the components over the number of agents $|\Upsilon|$ within a scenario, producing

$$\mathcal{L} = \frac{1}{|\Upsilon|} \sum_{i=1}^{\Upsilon} \Big( \frac{1}{|A_i|} \sum_{a=1}^{A_i} \mathcal{L}_{a,i}^{\text{policy}} - \alpha \mathcal{L}_{a,i}^{\text{H}} + \mathcal{L}_{a,i}^{V} \Big) \, . \tag{4.32}$$

The key part in Equation 4.32 is that $A_i$ is now dependent on the agent $i$. To see why that is the case, it is worth remembering that progressive widening is done on a per-agent basis. It is thus not only possible but highly likely that each agent $i \in \Upsilon$ has chosen a different total number of actions compared to the other agents. If the mean were taken only over all the actions for all agents at the same time, it would be biased towards agents that have done more progressive widening than others and thus produced more actions. First calculating the mean over all the actions for each agent $A_i$ and only then averaging over all agents ensures equal contribution of all agents to the scalar objective value.

In the final step, the multi-agent objective in Expression 4.32 has to be extended to allow for training on multiple scenarios at the same time:

$$\mathcal{L} = \frac{1}{|S|} \sum_{s=1}^{S} \left( \frac{1}{|\Upsilon_s|} \sum_{i=1}^{\Upsilon_s} \Big( \frac{1}{|A_{i,s}|} \sum_{a=1}^{A_{i,s}} \mathcal{L}_{a,i,s}^{\text{policy}} - \alpha \mathcal{L}_{a,i,s}^{\text{H}} + \mathcal{L}_{a,i,s}^{V} \Big) \right) \, . \tag{4.33}$$

The extension to multiple scenarios follows the same logic as in the previous step. $S$ represents the set of scenarios in the training run. The number of agents $|\Upsilon_s|$ is now dependent on the scenario $s$ being used to generate a particular training sample, as is the action count $A_{i,s}$ for an agent $i$. Again, simply taking the average loss over all agents as proposed in Equation 4.32 would lead to bias when using multiple scenarios. Only this time the objective value would be biased towards scenarios with more agents as compared to bias towards agents that have sampled more actions. Therefore the average over all agents within a scenario $\Upsilon_s$ has to be taken first.

## 4.4 Guiding the Search

When using MCTS in continuous action spaces as described in Section 2.2.2, one question remaining is how newly added actions are sampled. A simple solution is to select actions uniformly at random. This does however not take into account the specifics of the current state. In autonomous driving for instance a vehicle might drive right at the left edge of the road. When sampling new actions uniformly at random from the complete action

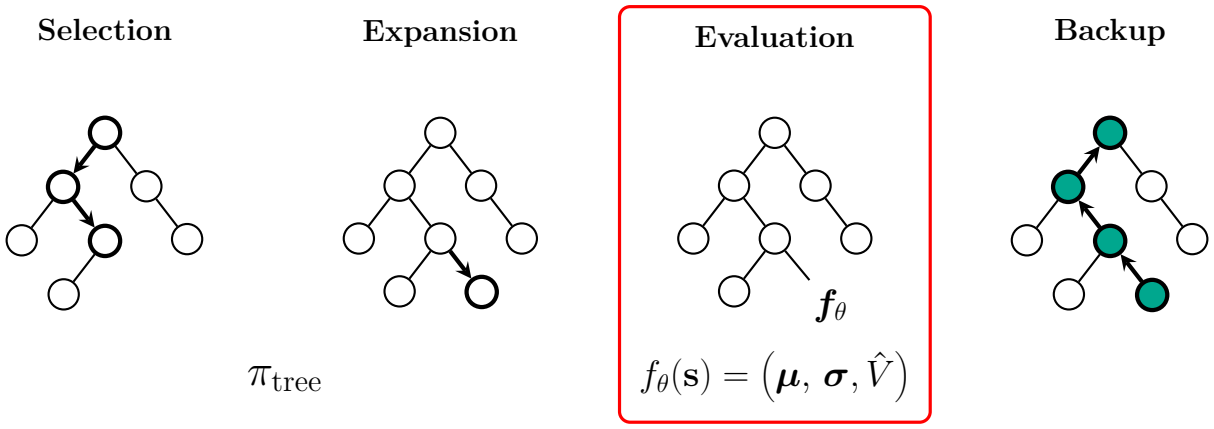



Figure 11: Iteration of the guided MCTS. The algorithm behaves like regular Upper Confidence Trees (UCT) (see Section 2.2.1) until the evaluation phase. Instead of a simulation, a network evaluation $f_\theta(\mathbf{s})$ of the expanded node is performed. A Value estimate $\hat{V}$ and distribution parameters $(\boldsymbol{\mu}, \boldsymbol{\sigma})$ are added to the node before the backup phase proceeds unmodified.

space, approximately half of the actions will lead to driving off the road. These search iterations are wasted, since the vehicle should not leave the road under any circumstance.

The goal of this work is improved sample efficiency of an MCTS-based planning algorithm. It is accomplished by pruning the search space such that undesirable actions are not being sampled. Guiding the search with state-dependent probability distributions has led to tremendous success in the game of Go [73, 75, 74]. Subsequently, it has been extended to continuous action spaces with the A0C algorithm [58]. A0C and AlphaZero form the basis for the following approach of using a neural network to guide the MCTS planning.

The procedure is visualized in Figure 11. Comparing the guided MCTS to the description of the basic algorithm from Section 2.2.1, it can be seen that the first two phases of the search are left unchanged. The key difference is in the evaluation stage: Once a new node has been expanded, a neural network evaluation is performed instead of a rollout. Note that the leaf node from which a new action has been sampled must have been expanded in a previous iteration of the search. Therefore the expansion step is already utilizing the guided MCTS[18]. In the evaluation stage, the network $f_\theta$ takes as input a state $\mathbf{s}$ and produces two outputs. First, a probability distribution $(\boldsymbol{\mu}, \boldsymbol{\sigma})$ conditioned on the current state $\mathbf{s}$[19]. This probability distribution is appended to the expanded node, allowing for repeated sampling without the need for additional network evaluations[20]. Second, the node is evaluated with a scalar value estimate $\hat{V}$. The use of a network estimate instead

[18] The root node has to be evaluated before initializing the search.

[19] If a GMM is used, the network also outputs the mixing coefficients $\alpha_k(\mathbf{s})$.

[20] Due to no training occurring during MCTS execution, the distribution parameters are deterministic given a state $\mathbf{s}$.

of a Monte Carlo estimate truncates the search at the expanded node and removes the need for a rollout. After the evaluation, the backup phase of the MCTS proceeds as described in Section 2.2.1. Instead of a rollout result however the value estimate obtained from the network $f_\theta$ is backed up.

Extending the guided search algorithm outlined above to multiple agents is straightforward: Rather than regular UCT, Decoupled UCT (DUCT) as described in Section 2.2.3 is used. Instead of a single distribution $(\boldsymbol{\mu}, \boldsymbol{\sigma})$ and a scalar value estimate $\hat{V}$, the network has to output a probability distribution for each agent $i$ as well as a value vector of length $|\Upsilon|$. Ideally, this model should account for interactions between the agents to achieve cooperative behavior. While a Multilayer Perceptron (MLP) as described in Section 2.3.1 can model dependencies between agents, it is also restricted to produce a fixed size output. As described in Chapter 4.1 however, driving scenarios may have a flexible number of vehicles, requiring a more sophisticated architecture. A model which is able to produce a flexible number of outputs is therefore described in the next section.

## 4.5 Network architecture

The previous chapter has established the need for a neural network which is able to learn interactions between a flexible number of inputs. The following paragraphs combine Section 2.3.2 and 2.3.3 with the hybrid input representation described in Section 4.1.1 into an architecture that fulfills all requirements of the environment. It is inspired by the works of [47] and [19], while being conceptually similar to the model of [87]. Its complete specification can be found in Appendix G.

Figure 12 shows a schematic visualization of the neural network proposed by this thesis. The architecture consists of two main components: First is a convolutional tower processing the visual map of the ego agent restricted to its own point of view. Note that all operations in the following description assume $3 \times 3$ kernels unless explicitly stated otherwise. After an initial convolutional layer, the image is downsampled using a pooling layer with stride 2. This initial stage is followed by three basic convolutional blocks [31], where the first convolution has stride 2 and increases the number of filters by a factor of $2\times$ over the output of the previous block. The second convolutional layer of each block uses stride 1 and does not change the depth of the feature maps. All blocks use skip connections to alleviate the degradation problem [31], which add the input of a block to its output. Before the block's input is added through the skip connection, an ECA-module learns global channel context [83]. Adding a channel-attention mechanism to the convolutional network is motivated by significant performance gains in applications to Go [88]. The final component of the image processing pipeline consists of a fully convolutional head inspired by [35]. It uses two convolutional layers with $1 \times 1$ kernel size after applying an

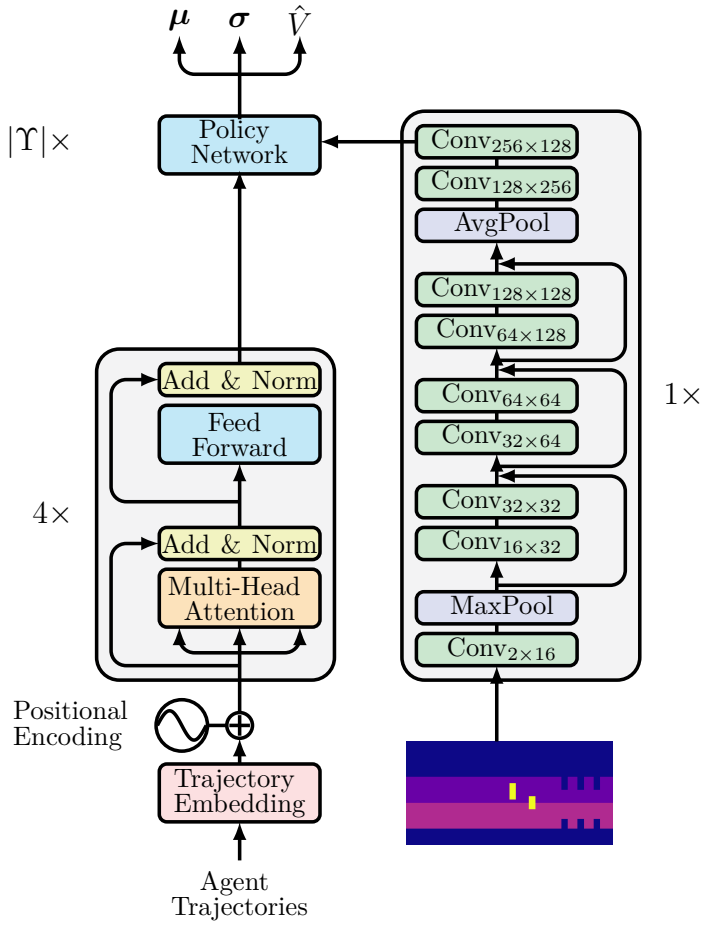



Figure 12: The Transformer model used in this thesis. On the right side, an ECA-ResNet processes visual input maps and outputs a vector representation. On the left side, the states of all agents $\Upsilon$ relative to the ego-agent are embedded and then fed into a Transformer encoder. The resulting $|\Upsilon|$ vectors are concatenated with the output of the convolutional pipeline. A parameter-shared MLP policy decodes the concatenated representations, producing a distribution $(\boldsymbol{\mu}, \boldsymbol{\sigma})$ and a value estimate $\hat{V}$ for each agent.

average pooling operation. The network output of the convolutional tower is a compact vector representation of the input image, denoted as $\mathbf{m}$.

The core component which lets the model learn interactions between a flexible number of agents is a Transformer encoder. The difference between the Transformer encoder and a decoder is that the encoder allows interactions in both directions of the sequence. The decoder on the other hand uses masking to ensure that only left-to-right dependencies are modeled[21]. Recall that an agent's numerical state $\mathbf{s}_i^{\text{num}}(t)$ consists of its own trajectory data as well as the relative trajectories of the other agents in the scenario (see

[21] In natural language processing, this means that the encoder produces a bidirectional representation. BERT is a prominent example of such a model [19]. The decoder is instead used for classical language modeling, e.g. in the GPT architecture. Here the model cannot condition on future tokens, i.e. right to left dependencies [19].

Section 4.1.1). This can be interpreted as a sequence of objects between all of which interactions are learned by utilizing the multi-head attention mechanism. Following the steps outlined in Section 2.3.3, the agent trajectories are first normalized and then embedded using a single fully connected layer. This produces an $|\Upsilon| \times d_{model}$ representation matrix $\mathbf{Z}$, where $|\Upsilon|$ is the number of agents in the scenario. Then positional encodings are added, allowing the network to identify each agent in the sequence. In this work, learned positional embeddings are used compared to fixed sinusoidal ones [19]. The embedded and augmented agent trajectories are next fed into a Transformer encoder with four layers to learn bidirectional interactions between the agents. Finally, the output of the Transformer is a representation matrix $\mathbf{T}$ of size $|\Upsilon| \times d_{model}$.

To obtain a distribution for each agent in the scenario, each row of the matrix $\mathbf{T}$ is concatenated with the vector output $\mathbf{m}$ of the convolutional tower. This can be interpreted as a sequence of combined representations for each agent. The elements of the sequence are then fed into an MLP policy network utilizing parameter-sharing. Reusing the network weights helps to keep the model as small as possible. In the final step, the policy outputs distribution parameters and value estimates for each agent.

The multi-agent Transformer architecture described in the previous paragraphs fulfills the requirement of being able to handle a flexible number of traffic participants. What is the intuition behind the model? Clearly an agent only receives information from its own sensors. Using itself as a model though, it can exploit the incomplete information obtained from other agents to approximate their action distributions. Of course this kind of modeling becomes coarser and coarser the further other agents are away. It is however not unlike humans make predictions about the immediate future of traffic situations. Drivers also have to plan using imperfect information, as they only perceive the road from their perspective.

The last part missing from the concept now that a network architecture has been determined is a training procedure. It is outlined in the next section.

## 4.6 Training algorithm

The high-level training procedure for the guided MCTS is described in Algorithm 1. Before the first execution of the search, hyperparameters are loaded from a configuration file and used to set the seeds. Then an empty replay buffer $\mathcal{D}$ is constructed. The neural network parameters $\theta$ are initialized randomly at the start of the training. The algorithm then performs a data-collection and training loop for $E$ episodes.

During training, experiences are generated by executing the guided MCTS with the current network parameters $\theta$. At each stage of the search, the root node states $\mathbf{s}$, actions

$\mathbf{a}$ and corresponding visitation counts $\mathbf{n}$ are exported[22]. Value targets $\hat{V}(\mathbf{s})$ are obtained using Equation 4.30 and saved as well. Note that the training data for all agents in the scenario is exported. This provides a regularization component by presenting the network with different ego perspectives for each scenario. More importantly, it speeds up data collection by a factor of $|\Upsilon|\times$, where $|\Upsilon|$ is the number of agents in a scenario. Since the generation of experiences is by far the most expensive computational component of the algorithm, training speed is improved significantly.

To further augment the training data, Gaussian noise is added to vehicle starting positions and scenario lane width. Given the noise standard deviations as $\sigma_{\text{lane}}, \sigma_x$ and $\sigma_y$ respectively, this results in:

$$w_{\text{lane}} = \bar{w}_{\text{lane}} + \epsilon_{\text{lane}}, \qquad \epsilon_{\text{lane}} \sim \mathcal{N}(0, \sigma_{\text{lane}}) \tag{4.34}$$

$$x_i(0) = \bar{x}_i(0) + \epsilon_x, \qquad \epsilon_x \sim \mathcal{N}(0, \sigma_x) \tag{4.35}$$

$$y_i(0) = \bar{y}_i(0) + \epsilon_y, \qquad \epsilon_y \sim \mathcal{N}(0, \sigma_y)\ , \tag{4.36}$$

where $w_{\text{lane}}$ is the lane width and $(x_i(0), y_i(0))$ is the starting position of agent $i$. The same quantities denoted with a bar correspond to the initial locations without noise.

Once the MCTS has generated $S$ samples, the data collection stops and all experiences are stored in the replay buffer $\mathcal{D}$. The buffer is using a fixed-size FIFO-queue, which is indicated in Algorithm 1 by the if-clause removing the oldest samples once the maximum size $D$ is reached. More sophisticated schemes of determining the buffer size exist, e.g. using a sublinear window function [88] or an exponentially growing buffer [2]. However, a fixed size queue keeps it simple and is sufficient to show the efficacy of the proposed approach.

After the generated experiences have been added to the dataset $\mathcal{D}$, a training loop iterates over the shuffled samples in the replay buffer $P$ times. First a batch of $B$ experiences is drawn from the buffer. The states $\mathbf{s}_b$ and actions $\mathbf{a}_b$ of the batch are used to generate action log-probabilities $\log \pi_\theta(\mathbf{a}_b|\mathbf{s}_b)$ and value estimates $V_\theta(\mathbf{s}_b)$ from the network. Criterion 4.33 can now be evaluated by using the value targets $\hat{V}(\mathbf{s}_b)$ and visitation counts $\mathbf{n}_b$. The final part of a training step consists of performing a gradient descent update for the neural network parameters $\theta$.

Once the network training has concluded, a new episode $e$ is started with the updated parameters $\theta$. The training proceeds for a fixed number of steps. A high level description such as the one given above of course omits a number of implementation details. The next chapter therefore highlights the most crucial choices.

[22] Subscripts $_0$ are omitted as only root node data is exported.

**Algorithm 1:** Guided MCTS training

Set Python, Numpy, PyTorch seeds to $S$.
Initialize network parameters $\theta$ randomly.
Initialize empty buffer $\mathcal{D} = \emptyset$.
**for** $e = 0$ **to** $E$ **do**

**while** $t < T$ **do**
  Load new network parameters $\theta$.
  Execute guided MCTS.
  Generate samples
  $(\mathbf{s}, \mathbf{a}, \mathbf{n}, \hat{V}(\mathbf{s})) \sim \hat{\pi}(\mathbf{a}|\mathbf{s})$.
**end**

Store samples:
$\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}, \mathbf{a}, \mathbf{n}, \hat{V}(\mathbf{s}))\}_{t=1}^{T}$.
**if** $|\mathcal{D}| > D$ **then**
  Remove $|\mathcal{D}| - D$ oldest samples.
**end**
**for** $p = 0$ **to** $P$ **do**
  Draw $B$ samples
  $\{(\mathbf{s}, \mathbf{a}, \mathbf{n}, \hat{V}(\mathbf{s}))\}_{b=1}^{B} \sim \mathcal{D}$.
  Generate action log-probabilities
  $\log \pi_\theta(\mathbf{a}_b|\mathbf{s}_b)$.
  Generate value estimates $V_\theta(\mathbf{s}_b)$.
  Update $\theta \leftarrow \theta - \lambda \mathcal{L}_\theta(\mathbf{a}_b, \mathbf{s}_b)$.
**end**

**end**

**Algorithm components**
Guided MCTS policy $\hat{\pi}(\mathbf{a}|\mathbf{s})$
Network policy $\pi_\theta(\mathbf{a}|\mathbf{s})$
Fixed size replay buffer $\mathcal{D}$

**Hyperparameters**
Seed $S$
Replay buffer size $D$
Number of Episodes $E$
Number of samples $T$
Number of training epochs $P$
Batch size $B$
Learning rate $\lambda$

**Languages**
C++
**Python**

# 5 Implementation

The goal of this chapter is to highlight some important details in the concept outlined previously. First, the general implementation of the system is discussed with an emphasis on the utilized software frameworks. This is followed by highlighting a specific code issue observed during network training. As it turns out, it is of crucial importance how the policy standard deviation is restricted.

## 5.1 System implementation

The overall training loop of the system described in this thesis is implemented in Python, while the Monte Carlo tree search (MCTS) itself is a C++ library. Once a run is started, a neural network is initialized from random parameters and saved to disk. PyTorch is the deep learning framework chosen for this thesis because it allows seamless interoperability between Python and C++ thanks to torchscript and the libtorch C++ library [63]. As soon as network initialization is completed, a data collector instance is constructed from a configuration file. It employs the distributed computing framework ray [60] to start multiple instances of an MCTS ROS node [65].

The MCTS loads the saved network parameters from the disk and runs the guided search until the scenario reaches a terminal condition. Care must be taken when training data is generated from different scenarios with possibly different numbers of agents. If all scenarios were chosen with equal probability for instance, the dataset would be biased towards scenarios with more agents. This is due to the trajectories of all agents being exported according to the training procedure in Section 4.6. The problem is resolved by sampling scenarios with probabilities that are inversely proportional to the number of agents $|\Upsilon_n|$ in the scenario. The probabilities can be obtained by first determining an unnormalized weight for a scenario $n$:

$$\begin{aligned} |\Upsilon_n| \cdot w_n &= \frac{1}{|S|} \\ \Leftrightarrow \qquad w_n &= \frac{1}{|\Upsilon_n| \cdot |S|} \ . \end{aligned} \tag{5.1}$$

The intuition behind Formula 5.1 is to have scenarios weights $w_n$ which result in a uniform distribution $\frac{1}{|S|}$ when multiplied by the number agents in a scenario $|\Upsilon_n|$. A valid probability distribution can then be generated through normalization of all individual weights $w_n$ by the sum of all weights $\sum_{s=1}^{S} w_s$:

$$P(S_n) = \frac{w_n}{\sum_{s=1}^{S} w_s} \ , \tag{5.2}$$

where $P(S_n)$ specifies the probability of sampling scenario $n$ for data generation.

During MCTS execution, the rollouts are replaced by a simulation policy which uses network evaluations to add both distribution parameters and value estimates to a node. Ego rewards obtained during execution of the search are normalized between $-1$ and 1 using Formula 5.3

$$\hat{r} = -1 + \frac{(r - \max(r) + d) \cdot 2}{d} \ , \tag{5.3}$$

where $r$ is the unnormalized reward given by the reward function in Section 4.1.2 and $d$ is the diameter of the reward interval. $\max(r)$ is the maximum obtainable reward. The used values can be found in Appendix H.

Unlike the Python framework, the libtorch C++ library does not possess a frontend for the PyTorch distributions. Therefore all policy distributions used in this work are implemented manually using Eigen3 [27] to keep them as lightweight as possible. After a stage of the search has finished, the root node statistics for each agent are exported in a PyTorch archive and saved on an SSD. They constitute the training data. Once enough data for an episode has been generated, the data collector terminates the MCTS runs. Subsequently, a PyTorch dataset is constructed from the $D$ newest samples, where $D$ is the size of the replay buffer. The network is then trained with stochastic gradient descent. Because storing data and training the network comprises only a fraction of the total training time these operations are implemented in Python.

If a rolling average computed from the success rate of the last three training episodes is higher than that of the previous best model, the network is saved to generate data in the next episode. When the current success rate average does not improve on the highest value, the gating test has been failed. In this case, the next batch of training experiences is still generated from the older model with the highest success rate. Nonetheless, the training proceeds with the current network and without reloading the weights.

Section 2.2.2 has established that progressive widening is done on a per-agent basis. Since the data collection phase might use scenarios with different numbers of agents, special attention has to be paid to padding the data batches. Value targets are padded using a large negative constant $-1 \cdot 10^{10}$. On one hand, this provides direct feedback through exploding loss values in the case of bugs. On the other hand, since rewards are normalized, it is easy to generate a mask for padded values.

Agent actions and visitation counts are padded using zero values. To generate a mask for the policy and entropy loss (refer to Section 4.3), note that an exported action cannot have a visitation count of zero. Therefore generating a mask that removes all zero visitation counts can also be used to discard log-probabilities generated from padded actions. The last implementation detail worth discussing is significant enough to warrant its own section, which follows.

## 5.2 Restricting the policy standard deviation

When learning the standard deviation $\sigma$ of a normal distribution in Reinforcement Learning (RL), one usually proceeds as follows [71, 66]:

1. Learn the logarithm of the standard deviation $\log \sigma$ instead of restricting the learned parameter to positive ranges only.
2. Restrain $\log \sigma$ to be in some range, e.g. an interval between 2 and $-20$.
3. Calculate $e^{\log \sigma}$ as standard deviation of the distribution.

In this process, step one helps to increase training stability by allowing the learned parameter to take positive and negative values. The second step adds numerical stability to the learning by capping outliers and restricting $\log \sigma$ to prudent values. Lastly, the actual standard deviation is computed by exponentiating the learned parameter. This ensures $\sigma \geq 0$, which is needed to generate a valid normal distribution.

The technique by which $\log \sigma$ is restricted is rarely specified but of critical importance. One common approach taken is to use "*clamping*" to cap the network output between interval boundaries [66]. The code snippet below implements this in PyTorch:

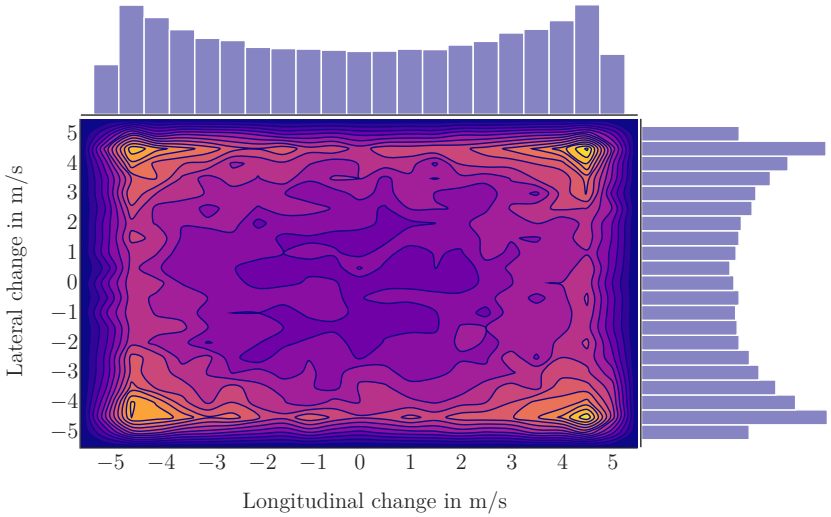
```
log_param_min, log_param_max = self.log_param_bounds
log_std = torch.clamp(log_std, min=log_param_min, max=log_param_max)
```

`torch.clamp` simply cuts values outside the target interval off and replaces them with the minimum or maximum possible values. Another option is to use a tanh transformation to restrict the standard deviation [90]:

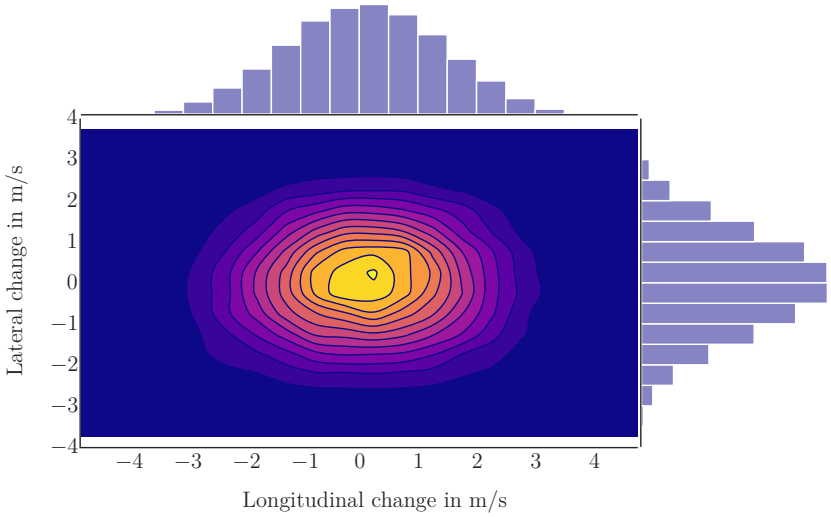
```
log_std = torch.tanh(log_std)
log_std_min, log_std_max = self.log_std_bounds
log_std = log_std_min + 0.5*(log_std_max - log_std_min) * (log_std + 1)
```

Here the learned parameter is first constrained between $-1$ and 1. Then $\log \sigma$ is rescaled and a bias term is added to achieve the desired interval boundaries. Now that both methods have been described, the question becomes: Which one to choose? This thesis uses the clamping approach motivated by its empirical performance. Figure 13 provides an explanation for the observed superiority of clamping. In the experiment, an identical neural network with random parameters is initialized. Then the distribution generated by these parameters is visualized by drawing 50000 samples and plotting an approximate $2D$ density over the action space.

Analyzing the results, it becomes immediately clear why the tanh method struggles to learn: The distribution generated is centered around the origin, but with a very narrow support. The histograms in Figure 13b further illustrate that values outside of the interval




(a) Density obtained through $\log \sigma$ clamping.




(b) Density obtained through $\log \sigma$ transformation.

Figure 13: Contour histograms using different techniques to restrict the standard deviation. To generate the plots, 50000 samples were drawn from the distribution parameters produced by the same random neural network. a) uses clamping to restrict the learned log standard deviation. As a result, the random initialization leads to exploration of the whole sample space. b) uses the tanh transformation. The resulting distribution resembles a $2D$ standard normal distribution and narrowly restricts sampling around the origin.

$[-3, 3]$ are sampled with very small probability. This is problematic, as it leaves a large amount of the action space unexplored at the start of training. Using a scaled tanh transformation to cap the log standard deviation therefore inhibits learning.

Comparing the `torch.clamp` approach, it can be seen that the distribution has four modes, one in each corner of Figure 13a. There is however also a reasonable likelihood of generating samples from all areas of the action space. This is desirable for a random network at the start of training: It allows the algorithm to fully explore all available actions before narrowing the distribution, thereby preventing premature convergence to local optima.

For the reasons elaborated on above, all experiments in the following section use clamping instead of the tanh transformation to restrict the log standard deviation.

# 6 Evaluation

To evaluate the proposed approach, two scenarios are used to train models: Scenario 06 (Figure 36), where two agents have to merge, and scenario 08 (Figure 38), where two agents have to pass a bottleneck. Scenario 06 is easier and allows multiple ablation studies whereas scenario 08 can be considered more challenging[23]. It should therefore provide more insights into the potential of the guided search algorithm. After introducing a baseline used for comparison, the section headers serve as guiding questions for the studies conducted in the body.

## 6.1 Evaluation Metrics

The algorithm described in the previous sections is empirically analyzed according to one main metric: the success rate. A scenario run is deemed successful if no collisions or invalid states occur. They are denoted with indicator variables $I_{\text{collision}}$ for a collision and $I_{\text{invalid}}$ for an invalid state. Recall that a state is invalid if the vehicle is not on the drivable area (refer to Section 4.1.2). Additionally, if the algorithm was unable to generate actions for each agent, the run is stopped with an indicator $I_{\text{unable\_continue}}$. Therefore the success of a single run can be stated in Definition 6.1:

$$I_{\text{success}} = \max(1 - I_{\text{collision}} - I_{\text{invalid}} - I_{\text{unable\_continue}}, 0) \ . \tag{6.1}$$

The indicator variable success is thus one if neither of the aforementioned negative events occurs. If a collision, invalid state or failed planning attempt has taken place, it is zero. Aggregating the individual scenario successes into the success rate over $N$ evaluations can be written as

$$P_{\text{success}} = \frac{1}{N} \sum_{n=1}^{N} I^n_{\text{success}} \ , \tag{6.2}$$

where $P_{\text{success}}$ is the success rate and $I^n_{\text{success}}$ is the indicator of success for the $n$-th run.

In this place it is worth mentioning that in Reinforcement Learning (RL) usually the episode reward is used to determine the learning progress. For the multi-agent driving scenarios evaluated in this thesis, the normalized cooperative reward $R$ is given as

$$R = \frac{1}{T} \sum_{t=1}^{T} \frac{1}{|\Upsilon|} \sum_{i=1}^{\Upsilon} r_{i,t} \ , \tag{6.3}$$

where $r_{i,t}$ is the reward of agent $i$ at time step $t$. $r_{i,t}$ is summed over all agents and time steps in the scenario before being averaged over the number of steps $T$ in the episode

[23] Why that is the case is disclosed in Sections 6.2 and 6.5.

as well as the agent count $|\Upsilon|$. As will be discussed in Sections 6.5 and 7.1, using the normalized cooperative reward as measure of success is problematic.

Another metric used in the subsequent evaluation is the percentage of times an agent's desire has been fulfilled. Recall from Section 4.1 that each agent has a goal state consisting of a target lane and a target velocity in longitudinal direction. How often this goal state is reached for a total number of $N$ runs can then be used as an alternative measure of success:

$$P_{\text{desire\_fulfilled}} = \frac{1}{N} \sum_{n=1}^{N} I^n_{\text{desire\_fulfilled}} \ , \tag{6.4}$$

where $I^n_{\text{desire\_fulfilled}}$ indicates that an agent's desire has been reached in the $n$-th run.

To facilitate reproducibility of the results shown in the following sections, the hyperparameters for the evaluation and training runs are stated in Appendix H. They are kept fixed across all runs unless explicitly stated otherwise. All training runs use the same network architecture which is defined in Appendix G. The seeds to initialize the random number generators for training and evaluation are specified in Appendix J and K, respectively. When a trained model is evaluated, no learning occurs and its weights are kept fixed.

Now that the metric for evaluation and the training settings have been discussed, the following sections empirically analyze the capabilities of the proposed algorithm.

## 6.2 The baseline

When evaluating any method, it is important to select a competitive baseline as comparison. As such, the Monte Carlo tree search (MCTS) developed by [46] is chosen when evaluating the proposed approach. It uses several heuristics to improve its performance in low iteration settings. This is in contrast to the network-guided search which learns from tabula rasa. In the following, these heuristics are explained.

To generate new actions for progressive widening, Section 2.2.2 refers to uniform sampling as the simplest strategy. The baseline however uses *Blind Values* as an orthogonal approach to produce guided actions [17, 46]. Action generation proceeds as follows: For each actually chosen action, first a set of candidate actions is sampled uniformly from the action space. Their blind value is calculated subsequently. It can be seen as a scoring function utilizing the statistics of already explored actions. Together with a distance measure the statistics provide an attractiveness score [46]. The value is high for actions far from already selected ones at the beginning of the search. With higher visitation counts, actions with higher UCT values are preferred. Blind values thus weigh exploration versus exploitation for each actually expanded action [46].

As a second heuristic that is particularly useful in low iteration settings, the baseline adds a number of pre-calculated maneuvers to each newly expanded node. These maneuvers are generated by calculating actions which conform to previously determined semantic action groups [46]. An example action from these groups is for instance a lane change to the left while decelerating. To calculate values for the chosen maneuvers, the MCTS relies on the scenario specification [46]. The number of actions added this way is dependent on the search depth: If the newly expanded node is close to the root of the tree, nine pre-calculated maneuvers are added. For nodes deeper in the tree, only five basic actions are generated [46].

What happens if the predetermined driving maneuvers in the last paragraph do not conform to the action bounds? In this case, the baseline MCTS *ignores the constraints*. As a result, it is able to select actions which are not accessible to the RL algorithm. Through the tanh squashed normal distribution introduced in Section 4.2, the approach proposed by this thesis cannot violate the action bounds.

To make results comparable, actions are bounded for the baseline in the following evaluation. Additionally, ego reward normalization according to Equation 5.3 is added to produce equally scaled values. Table 2 shows how these modifications alter the baseline results. Enforcing action bounds has a negligible effect on the success rate. Interestingly, the ego reward normalization improves the baseline compared to using unnormalized rewards. A variant of the MCTS without pre-calculated maneuvers is also evaluated. Without them, the baseline is not able to solve scenarios 02, 05, 06, and 07 at all. This highlights its reliance on heuristics in low iteration settings.

Due to its higher success rate and comparable reward values over the unmodified baseline, the **Bound & Norm** variant from Table 2 is chosen for all further evaluations. As scenarios, 06 is selected as it is solvable easily by the MCTS. This allows an evaluation of whether the guided search is able to recover the pre-calculated heuristic maneuvers. As another scenario, 08 is chosen. Here the heuristic actions are insufficient and cannot solve the task, as evidenced by the MCTS needing more iterations.

With the baseline being introduced, the focus of this work now shifts towards evaluation of the guided search approach. A starting point is a qualitative overview assessing whether the network is able to successfully prune the sample space.

| | **Base** | | **Norm** | | **Bound** | | **Bound & Norm** | | **No pre-selection** | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Iterations** | 100 | 200 | 100 | 200 | 100 | 200 | 100 | 200 | 100 | 200 |
| **Scenario** | | | | | | | | | | |
| Scenario 01 | 0.99 | 0.98 | 1.00 | 0.98 | 0.99 | 0.98 | 1.00 | 0.98 | 1.00 | 1.00 |
| Scenario 02 | 0.80 | 0.82 | 0.97 | 0.96 | 0.80 | 0.82 | 0.97 | 0.96 | 0.00 | 0.00 |
| Scenario 03 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Scenario 04 | 0.98 | 1.00 | 0.99 | 0.99 | 0.98 | 1.00 | 0.99 | 0.99 | 1.00 | 1.00 |
| Scenario 05 | 0.91 | 0.95 | 0.99 | 0.98 | 0.92 | 0.93 | 1.00 | 0.99 | 0.00 | 0.00 |
| Scenario 06 | 0.97 | 0.97 | 0.99 | 1.00 | 0.97 | 0.99 | 0.99 | 1.00 | 0.00 | 0.00 |
| Scenario 07 | 0.08 | 0.12 | 0.07 | 0.11 | 0.08 | 0.12 | 0.07 | 0.11 | 0.00 | 0.00 |
| Scenario 08 | 0.17 | 0.69 | 0.20 | 0.79 | 0.17 | 0.69 | 0.20 | 0.79 | 0.39 | 0.49 |
| **Mean** | 0.7375 | 0.8163 | 0.7763 | 0.8513 | 0.7388 | 0.8163 | 0.7775 | 0.8525 | 0.4238 | 0.4363 |

Table 2: Performance of the baseline MCTS for 100 and 200 iterations. **Norm** corresponds to normalized ego rewards according to Equation 5.3. **Bound** enforces action bounds in $[-5, 5]$. The combination is denoted as **Bound & Norm**. **No pre-selection** shows performance without expanding pre-calculated maneuvers. The best performing algorithms are shown in cyan (100 iterations) and brown (200 iterations). Ego reward normalization improves the baseline results. Not using pre-selected actions yields a performance drop-off. For all following comparisons, the **Bound & Norm** baseline version is chosen as it performs best and produces comparable reward values.

## 6.3 Does the model work as intended?

The goal of this thesis as stated in the introduction is to improve the MCTS sample efficiency through guiding the search with learned knowledge. A simple, yet effective way to qualitatively verify whether the desired phenomenon occurs or not is to visualize the network distributions.

Figure 14 contrasts an agent's view in scenario 06 with the output distribution of the neural network. The visual maps in Subplot 14a depict the situation: Agent zero approaches a number of obstacles on its current lane and must merge into the middle lane to avoid a crash. At the same time, agent one can be seen entering the images from above. It also desires to merge into the center.

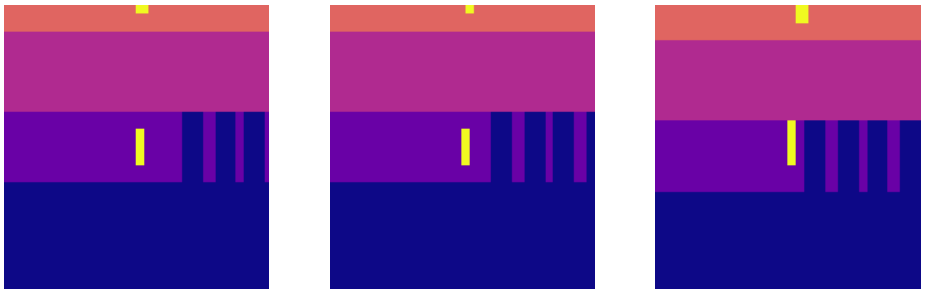
The corresponding network distributions are shown below in Figure 14b. It can be seen how the GMM effectively prunes all actions in the sample space which correspond to driving to the right. As this would lead to either driving off the road or colliding with an obstacle, the network guides the MCTS as intended. Additionally, the network shows a preference for slowing down when approaching the obstacle in the first plot of 14b. After that, actions are sampled over the whole longitudinal range. Through this, the agent is able to avoid the other vehicle by either accelerating or decelerating. The latter two visualizations in Graphic 14b indicate that the network still prefers deceleration.

A less dramatic example of the model's prediction of the action space is shown in Figure 15. Here agent zero has to avoid obstacles on both lanes as well as another vehicle while driving into the bottleneck. The corresponding maps are depicted in Illustration 15a. Note that scenario 08 is more narrow compared to scenario 06: It only has two lanes instead of three.

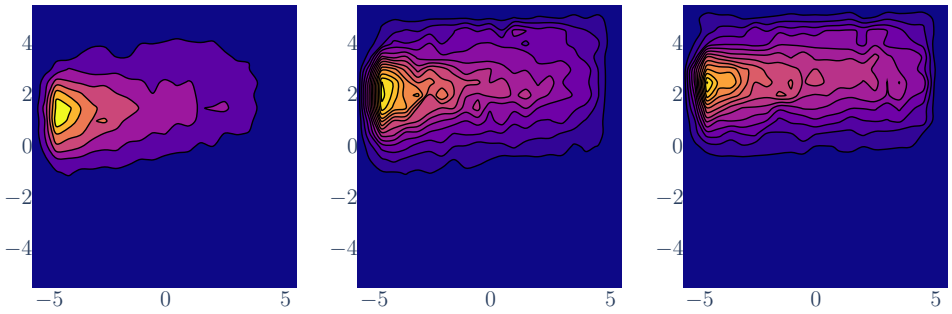
The network distributions visualized in Figure 15b show a slight preference towards driving left. This is needed to avoid the obstacles. However, both acceleration and deceleration are permitted with no obvious inclination visible. The agent therefore stays flexible and can avoid the other vehicle by either slowing down or accelerating.

To further evaluate the proposed approach qualitatively, Figure 16 visualizes 20 trajectories for each agent in scenario 08. 100 MCTS iterations are used for both the baseline as well as the guided search. The seeds are kept fixed such that the starting positions are the same for both approaches. They are marked with red dots. Obstacle positions and lane width are not randomized for visualization purposes.

Both plots show immediately how the learning based approach completes more runs than the baseline MCTS. The pair of agents passes through the middle while at the same time avoiding a collision. Their trajectories are both close to the center line between the lanes, indicating that they pass the bottleneck consecutively.

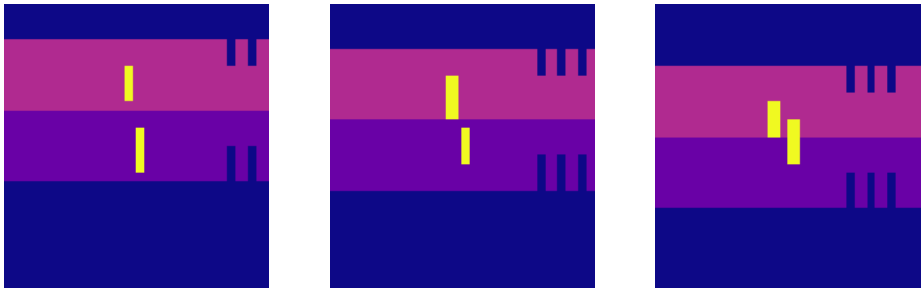


(a) Agent zero visual map.

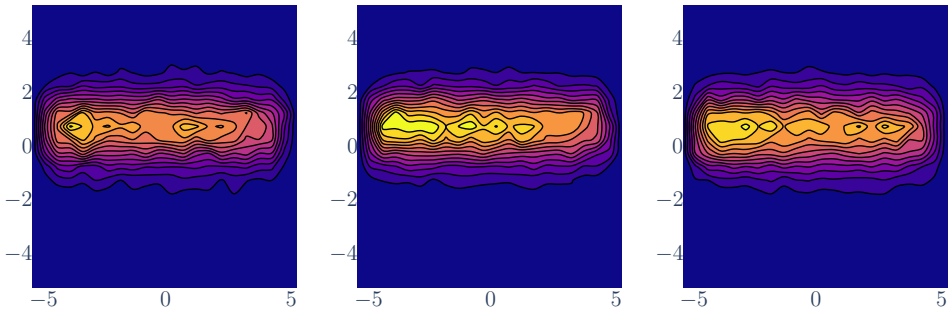



(b) Agent zero output distribution. $x$-axis refers to longitudinal velocity change in $m/s$. $y$-axis is lateral velocity change in $m/s$.

Figure 14: First three steps of scenario 06 for agent zero. In (a), the perceived visual maps are shown. It can be seen how the agent is approaching obstacles on its lane. (b) shows the distributions produced by the network for a Gaussian Mixture Model (GMM) with 3 components. 20000 samples are drawn to visualize the distributions. The model effectively biases the action space. The actions preferred by the network can be interpreted as decelerated lane change to the left.

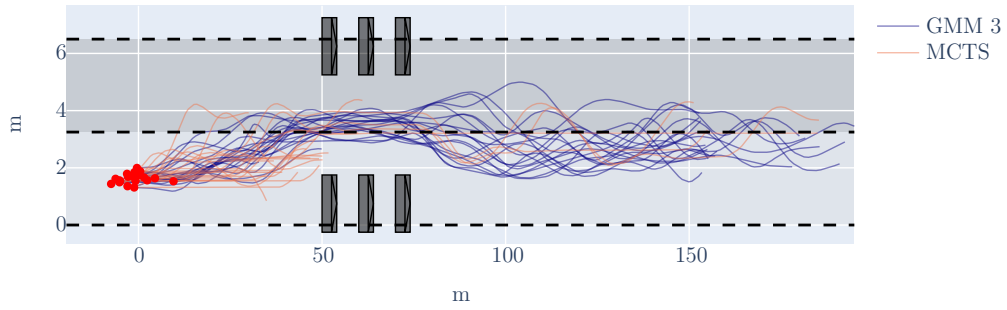


(a) Agent zero visual map.

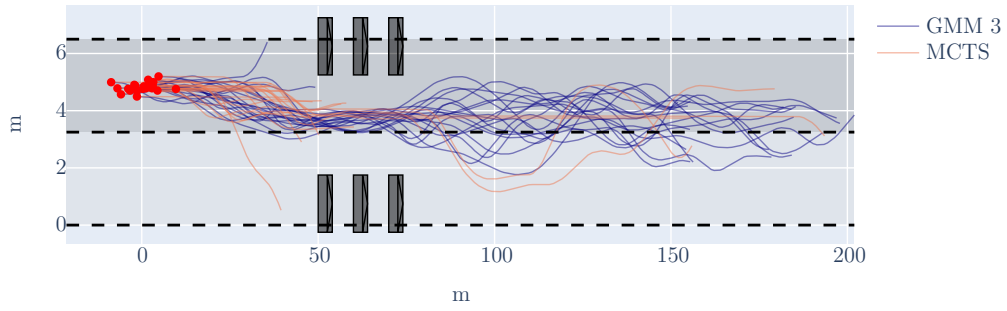



(b) Agent zero output distribution. $x$-axis refers to longitudinal velocity change in $m/s$. $y$-axis is lateral velocity change in $m/s$.

Figure 15: First three steps of scenario 08 for agent zero. In (a), the agent is approaching the bottleneck while noticing the other vehicle. (b) The network distribution biases sampling towards a slight left turn to avoid the obstacles. To avoid a collision with the simultaneously merging other agent, all longitudinal actions are still allowed. 20000 samples are drawn to visualize the distributions.




(a) Agent zero trajectories.




(b) Agent one trajectories.

Figure 16: 20 trajectories using 100 iterations for each agent in scenario 08. Seeds are fixed, resulting in the same starting positions (denoted as red dots). Guided search is using a GMM with three components. The plots show how both agents are able to navigate through the bottleneck without colliding more often when using the guided search. This contrasts with the pure MCTS approach, where the vehicles crash when driving into the center more regularly.

On the other hand, Figure 16a already shows clearly how agent zero is not able to avoid the obstacles and the other agent at the same time. Almost half of its trajectories end in the area to the left of the first obstacle. A pass of the bottleneck only succeeds four times for the baseline.

The visualizations in this section have shown that the network is able to successfully guide the search. Next, the choice of MCTS selection policy is discussed.

## 6.4 Which selection policy to choose?

In Section 2.2.1, two strategies for selecting the action to be executed in the environment are introduced: selection of the action with the highest visitation count and selection of the action with the highest action value. The choice in the Upper Confidence Trees (UCT) algorithm is usually the former [12].

Which strategy is best for cooperative autonomous driving is not immediately obvious. Therefore a short empirical evaluation is conducted to find the best approach. Figure 17

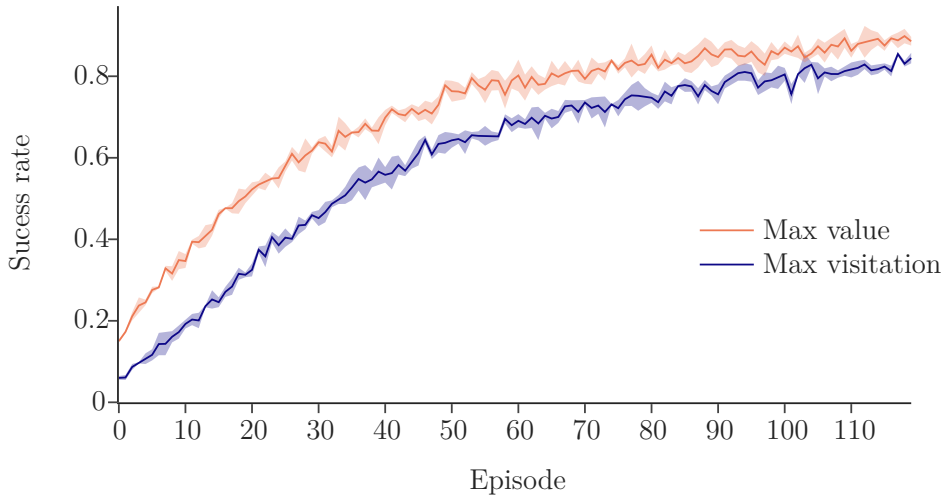



Figure 17: Training success in scenario 06 using different MCTS selection policies. Both runs use 50 iterations and the same three seeds. Selecting the action with the highest action value yields a higher success rate.

shows the training progress of both selection strategies in scenario 06 for 50 iterations. In contrast to the literature [12], selecting the action with the highest action value seems to outperform selecting the most visited action.

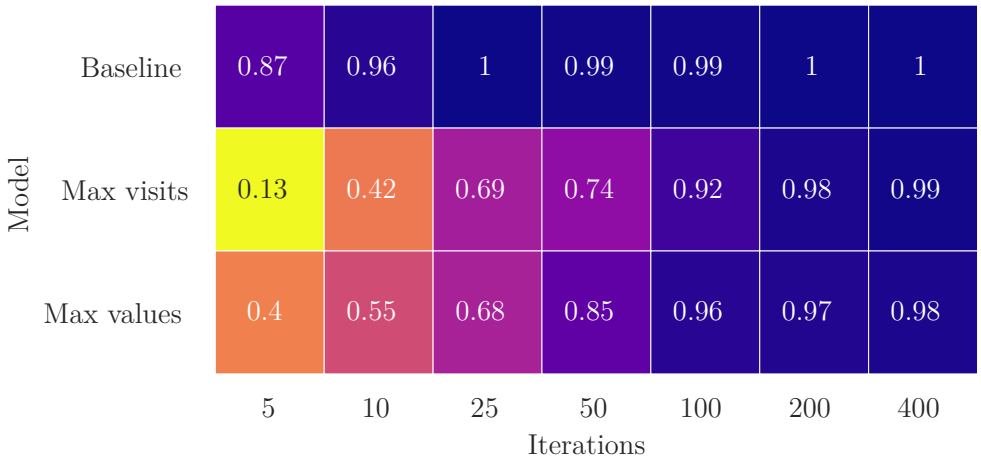



Figure 18: Training success in scenario 06 using different MCTS selection policies. Both models are trained using 50 iterations. Selecting the action with the highest action value yields a higher success rate for lower iterations. Results are averaged for three models.

Matrix 18 further examines the learned models in an evaluation setting. Overall, selecting the action with the maximum action value outperforms selection based on visitation counts with a mean success rate of 0.7686 to 0.6971. The difference mainly stems from low iteration settings, particularly for 5, 10 and 50 iterations. For higher iteration values as well as 25 iterations, the maximum visitation selection has a marginally higher success rate. Interestingly, the baseline is stronger than both learned models. Its high success percentage using just 5 iterations confirms that the heuristic maneuvers are good enough to solve some scenarios outright.

| **Selection** | **Iterations** / **Metric** | 5 | 10 | 25 | 50 | 100 | 200 | 400 | **Mean** |
|---|---|---|---|---|---|---|---|---|---|
| **Max value** | **Success** | 0.3967 | 0.5533 | 0.6767 | 0.8467 | 0.9567 | 0.9700 | 0.9800 | **0.7686** |
| | **Reward** | 0.2213 | 0.2366 | 0.2554 | 0.2738 | 0.2905 | 0.2951 | 0.3008 | **0.2676** |
| | **Desire** | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0033 | 0.0033 | 0.0033 | **0.0014** |
| **Max visits** | **Success** | 0.1300 | 0.4200 | 0.6900 | 0.7433 | 0.9233 | 0.9833 | 0.9900 | **0.6971** |
| | **Reward** | 0.1514 | 0.2104 | 0.2511 | 0.2629 | 0.2865 | 0.2942 | 0.3001 | **0.2509** |
| | **Desire** | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0200 | **0.0029** |

Table 3: Performance of different selection policies in scenario 06. Using the maximum action value as criterion outperforms selecting the most visited action in four out of seven settings. Final selection based on visitation counts has a slightly higher success rate for 25, 200 and 400 iterations.

The difference between both guided search models is expounded by Table 3. In addition to the higher average success rate, maximum value selection also has the highest cooperative reward values for each iteration setting. As the number of iterations increases, the

difference between both models narrows. For the rate of fulfilled desires, both maximum action value and maximum visitation count are close to zero for all settings.

Now that the choice of final selection policy has been evaluated, the next section discusses the performance of guided search versus the baseline in detail. Due to its higher mean success rate, the maximum action value policy is chosen for all following evaluations.

## 6.5 How well does the learned model perform versus pure MCTS?

The success rate visualization of different selection policies in the last section (Figure 18) shows the baseline outperforming the learned models. This raises an interesting question: Are there scenarios where the guided search exceeds the success rate of the baseline and vice versa?

Looking at the training success in Plot 19 first, one can see how for scenario 06 the model is able to approximate the baseline performance only late during the run. This is not

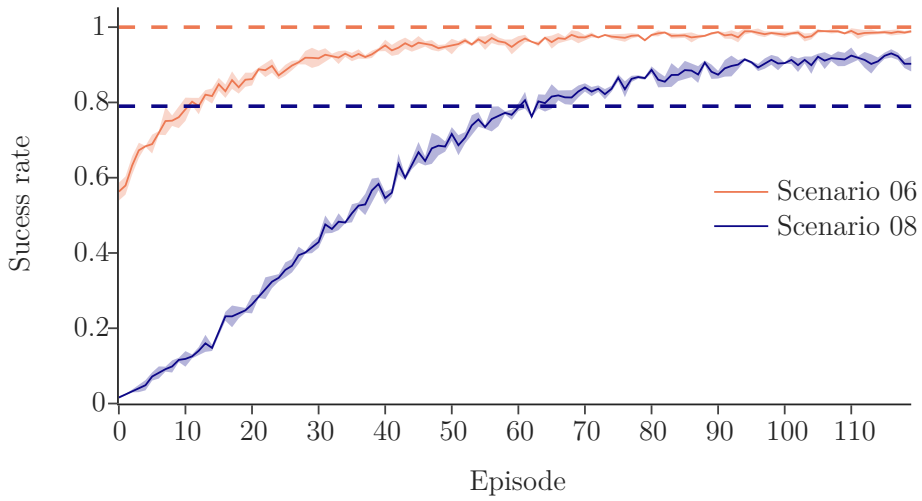



Figure 19: Training progress for scenarios 06 and 08. Models are trained over three seeds using 200 iterations. Error bands show one standard deviation. The success rate of the pure MCTS baseline for the same number of iterations is shown as dotted line.

surprising because the baseline completely solves the task for 200 iterations. For scenario 08, the guided search exceeds the performance of the MCTS at around episode 65 and continues to improve upon it.

A closer look at Table 4 confirms the hypothesis for scenario 08: The learned models guide

the search effectively, yielding a higher success and desires fulfilled rate than the baseline in all iteration settings. The difference is particularly noticeable below 200 iterations. At

| Model | Iterations / Metric | 5 | 10 | 25 | 50 | 100 | 200 | 400 | Mean |
|---|---|---|---|---|---|---|---|---|---|
| GMM 3 | **Success** | 0.3000 | 0.4967 | 0.6833 | 0.8267 | 0.8300 | 0.9433 | 0.9500 | **0.7186** |
| | **Reward** | 0.2779 | 0.2954 | 0.3119 | 0.3205 | 0.3276 | 0.3341 | 0.3367 | **0.3149** |
| | **Desire** | 0.0067 | 0.0200 | 0.0567 | 0.1767 | 0.1333 | 0.2900 | 0.3833 | **0.1524** |
| Baseline | **Success** | 0.0200 | 0.0100 | 0.0200 | 0.0900 | 0.2000 | 0.7900 | 0.8700 | **0.2857** |
| | **Reward** | 0.6738 | 0.6430 | 0.6404 | 0.6096 | 0.6145 | 0.6255 | 0.6238 | **0.6330** |
| | **Desire** | 0.0000 | 0.0100 | 0.0000 | 0.0000 | 0.0002 | 0.0002 | 0.1000 | **0.0214** |

Table 4: Performance of the GMM 3 model versus the baseline in scenario 08. The best value is shaded in cyan. Using guided search outperforms the pure MCTS baseline in both success and desires fulfilled percentage. The difference is particularly stark in low iteration settings. It is noteworthy that the MCTS consistently produces runs with higher reward despite a lower success rate. For the guided search, the results of three models are averaged.

100 MCTS traces, the guided search succeeds in 83.00% of runs compared to only 20.00% for the baseline. Starting at 200 iterations, the success rate for the baseline makes a jump and closes the gap between both approaches. Using a learned network however is still superior, which is also reflected in terms of the average success rate at 0.7186 compared to 0.2857.

The results described above also transfer to the rate of fulfilled desires. This should come as no surprise, as an agent is only able to reach its target state if it stays on the road and does not crash. The overall average rate of desires fulfilled across all iterations is 0.1524 for the guided search versus 0.0214 for the baseline.

Interestingly, the results of the previously examined evaluation metrics do not carry over to the normalized cooperative reward. Here the baseline strongly outperforms the guided search by more than a factor of 2× on average. The results for 5 iterations especially raise questions. At this setting, the baseline achieves its highest cooperative reward despite a success rate of only 2% and no desires fulfilled.

The outcomes are flipped regarding the detailed evaluation results of scenario 06 in Table 5. Here the baseline posts stronger results compared to the guided search in all iteration settings. 25 iterations are already enough for it to solve the task with 100% or 99% success. This is consistent with the findings of Section 6.2, which report that the pre-calculated maneuvers are enough to succeed in scenario 06. While the learned model is able to recover the knowledge added through the heuristic, it needs 200 iterations to do so.

| Model | Iterations / Metric | 5 | 10 | 25 | 50 | 100 | 200 | 400 | Mean |
|---|---|---|---|---|---|---|---|---|---|
| GMM 3 | **Success** | 0.4367 | 0.5733 | 0.7133 | 0.8533 | 0.9467 | 0.9900 | 0.9933 | **0.7867** |
| | **Reward** | 0.2168 | 0.2345 | 0.2564 | 0.2755 | 0.2901 | 0.2986 | 0.3045 | **0.2681** |
| | **Desire** | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | **0.0000** |
| Baseline | **Success** | 0.8700 | 0.9600 | 1.0000 | 0.9900 | 0.9900 | 1.0000 | 1.0000 | **0.9729** |
| | **Reward** | 0.5798 | 0.4976 | 0.4943 | 0.4633 | 0.4791 | 0.5126 | 0.5157 | **0.5060** |
| | **Desire** | 0.0000 | 0.0200 | 0.0100 | 0.0300 | 0.0000 | 0.0000 | 0.0000 | **0.0086** |

Table 5: Performance of the GMM 3 model versus the baseline in scenario 06. The best value is denoted in cyan. Pure MCTS outperforms the 200 iteration model in all regards. The difference is particularly noteworthy in medium iteration settings $(10, 25, 50, 100)$. Three models are averaged for the guided search.

The results described for the success rate also carry over to the other evaluation metrics. It is noteworthy that neither the baseline nor the guided search are able to achieve a percentage of desires fulfilled higher than 3%. As for the cooperative reward, the baseline substantially outperforms the guided search similar to scenario 08. However, the results fluctuate without apparent relationship to the success rate. Another important measure

| **Model** | Runs | Iterations | Success | Wall clock |
|---|---|---|---|---|
| GMM 3 | 100 | 50 | 0.8400 | $49s$ |
| Baseline | 100 | 400 | 0.8700 | $57s$ |

Table 6: The guided search approach is competitive with the baseline using eight times less iterations and $8s$ less wall clock time.

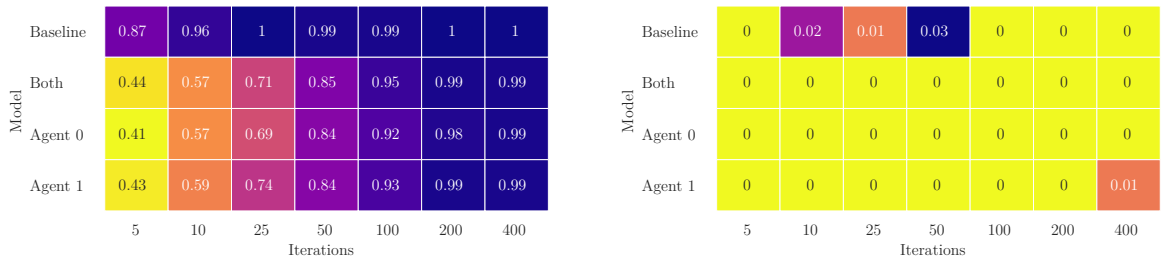
to look at — particularly important for potential real-world deployment — is the wall clock time. Tables 6 and 7 report the results. In scenario 08, the guided search achieves a similar success rate to the baseline with $8\times$ less iterations. Due to the reduced number of planning traces, it is competitive in terms of wall clock time and even slightly faster. On scenario 06 in comparison, the learned model needs 100 iterations to be as successful as the plain MCTS[24]. This results in planning that is orders of magnitude slower than the baseline. The reason for these findings lies in the computational cost of network evaluations. Profiling the code reveals that the guided search spends around 48% of its runtime performing neural network inference.

Lastly, Section 4.5 describes how the networks can use the imperfect information gained from only one agent's point of view to plan for other agents as well. The results for the corresponding evaluations are visualized in Figures 21 and 20. For the success rate, the

[24]The comparison is made between values within tables. The wall clock time between tables is not comparable due to different background workloads on the servers.

| **Model** | Runs | Iterations | Success | Wall clock |
|---|---|---|---|---|
| GMM 3 | 100 | 100 | 0.9900 | $2m43s$ |
| Baseline | 100 | 100 | 0.9900 | $4s$ |

Table 7: The guided search approach performs the same in terms of success percentage compared to the baseline. It requires the same number of iterations which results in non-competitive wall clock time due to the slowness of network evaluations.




(a) Success rates for different points of view. (b) Desires fulfilled for different points of view.

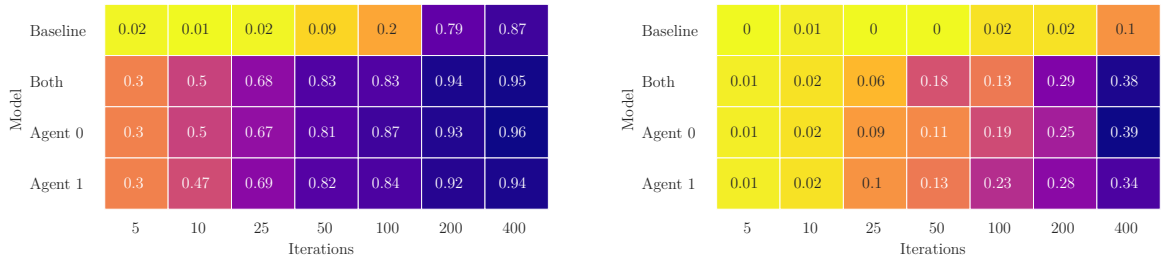
Figure 20: The two matrices show the model performance depending on which agent is using the network in scenario 06. In (a), the success percentage is shown. (b) depicts the percentage of desires fulfilled. Planning from one agent's point of view shows similar performance compared to planning for both agents. An average of three models is shown for the guided search.

findings are very similar for all points of view. Using only the visual map for a single agent is usually within a bound of three percentage points higher or lower compared to using maps for both agents. This results in comparable average success rates when utilizing all maps (0.7867) versus only agent one's map (0.7871) in scenario 06. There is a slight drop-off when using agent zero's visual input only with 0.7714.

The only outlier in scenario 08 occurs when using 100 iterations, where planning from agent zero's point of view is more successful by four percentage points compared to both agents. Apart from that, the average success across iterations is remarkably stable: 0.7186 for both compared to 0.7186 (agent zero) and 0.7119 (agent one).

When looking at the percentage of desires fulfilled, it stands out that in scenario 06 the learned models are not able to achieve values above zero with one exemption. The baseline has similarly low percentages with exceptions for 10, 25 and 50 iterations.

The results are more interesting for scenario 08, where fluctuations for the rate of desires fulfilled can be found starting at 25 iterations. Using only agent zero's map performs noticeably better for 100 MCTS traces. Both agent's point of view is more successful at 25 and 50 iterations. Overall however, the mean percentage of desires fulfilled hovers




(a) Success rates for different points of view. (b) Desires fulfilled for different points of view.

Figure 21: The two matrices show the model performance depending on which agent is using the network in scenario 08. In (a), the success percentage is shown. (b) depicts the percentage of desires fulfilled. Using only a single agent's point of view is competitive with utilizing maps from both agents. Guided search outperforms the baseline for all settings and metrics. Three models are averaged for the guided search results.

around 15% with 15.24% desires fulfilled when using all visual maps compared to 15.10% and 15.62% for agent zero and agent one, respectively. The baseline is not competitive at only 2.14%.

## 6.6 How important is the number of iterations?

For approaches that combine RL with search, there is a trade-off between time allocated to the learning component of the algorithm and time allocated to the planning component [59]. In the context of this thesis, planning corresponds to the number of MCTS iterations used during training. What number works best is not immediately obvious and therefore evaluated in the context of scenario 06.

Figure 22 shows the training progress for 50, 100 and 200 iterations. While all models

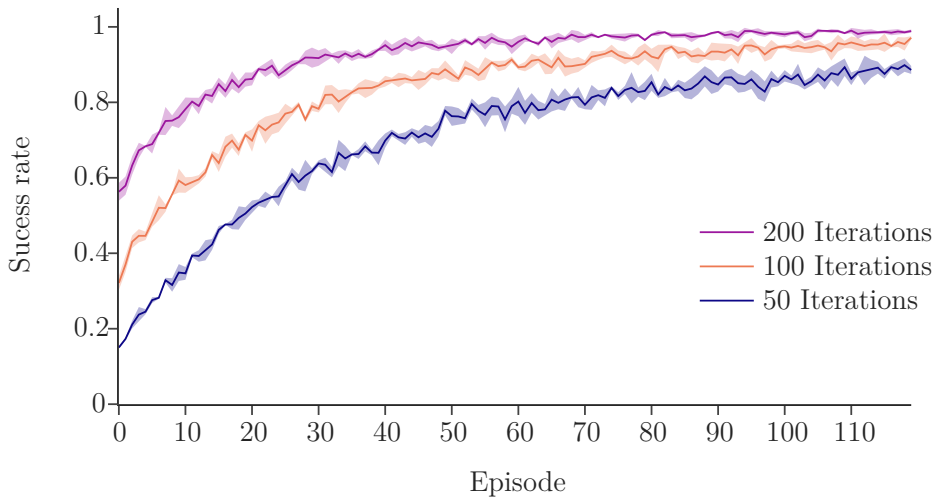



Figure 22: Training progress with different iteration numbers. The 200 iteration model learns in fewer episodes than both other models. Using 100 iterations approaches a similar level of success as using 200. For all models, three seeds are trained and one standard deviation is shown.

successfully improve over the course of the episodes, the network trained with 200 iterations exhibits the highest success rate. This is unsurprising, as more time spent in MCTS searches should not only improve the quality of the action selected in the environment but also produce enhanced training targets.

Delving deeper into the issue, Matrix 23 reveals an interesting dynamic: Despite being trained with a smaller time budget, the 100 iteration model outperforms the one trained with 200 iterations. The phenomenon is especially noticeable when using a medium number of MCTS traces (10, 25 and 50). Both the 100 as well as the 200 iteration networks are superior when compared to the 50 iteration model.

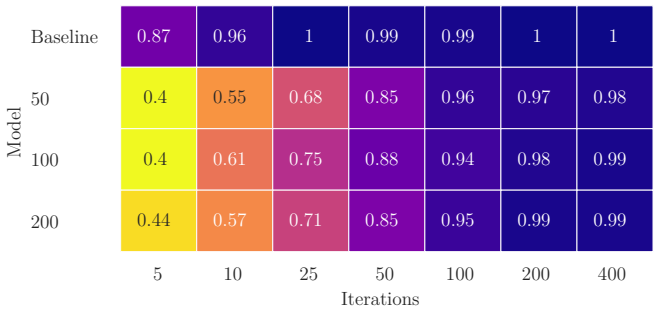



Figure 23: Evaluation of models trained with different iteration numbers. The model using 100 iterations performs best for medium iteration values. In high iteration settings, the networks show similar performance. All results are averaged over three models.

| **Iterations** | 50 | 100 | 200 |
|---|---|---|---|
| Wall clock | $8h53m$ | $14h28m$ | $32h11m$ |
| Mean success | 0.7686 | 0.7933 | 0.7867 |

Table 8: Wall clock times for different training iteration numbers in scenario 06. The mean evaluation success rate is shown below. The model using 100 iterations has a higher evaluation success rate than both 50 and 200 iterations.

Table 8 reports the mean success rates across all settings together with the wall clock time needed for the training. It confirms the findings of the matrix plot, where the 100 iteration model performs best. When looking at the wall clock time, it takes roughly 1.6× as long to train as the 50 iteration model. The network trained using 200 MCTS traces takes another 2.2× longer compared to using 100 traces. Larger models are not trained due to the computational resources required.

After this short evaluation of the number of iterations, the next section tackles an important question regarding the policy distribution: How many mixture components are needed to properly guide the search?

## 6.7 How many mixture components are needed?

As noted in Section 2.3.4, a GMM has the theoretical ability to approximate any distribution. Using a mixture model however introduces an additional hyperparameter in the number of components $K$. On one hand, having more components is desirable as it results in a more expressive model. On the other hand, it might also degrade performance or destabilize the training.

Therefore it is sensible to perform an empirical analysis to determine the optimal number of components. The training results for different $K$ in scenario 08 are shown in Figure 24.

For each setting, three runs are executed with the same three seeds each. The error

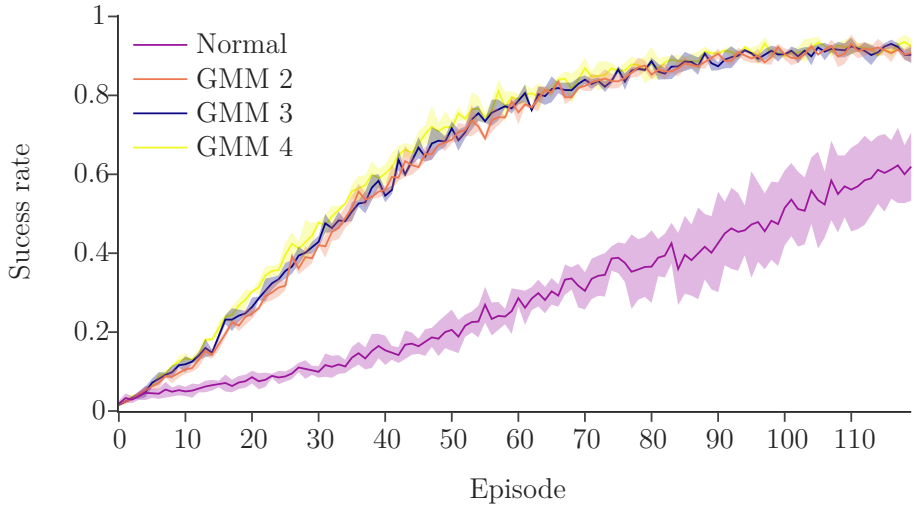
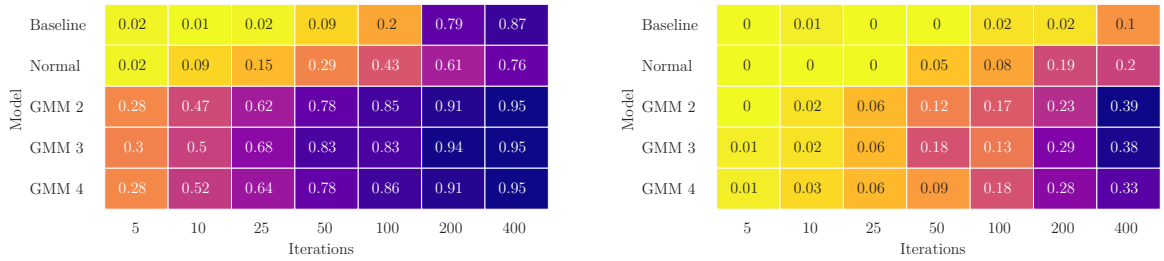



Figure 24: Training plots for models with different numbers of components $K$ in scenario 08. The Gaussian policy performs subpar compared to using a mixture model. Runs are averaged over the same 3 seeds on Scenario 08 with one standard deviation shown.

bands show one standard deviation from the mean. It can be seen clearly that a policy parameterized by only a normal distribution has a noticeably lower success rate than the mixture models. This is an indicator that a single normal distribution is not enough to fit a proper model. When increasing the number of components, no distinguishable difference in the training can be observed.

Looking at the evaluation results, a GMM with $K = 3$ shows slightly improved performance compared to the other two mixture models. Plot 25 shows the data for all evaluations. 100 runs with fixed seeds are performed for each iteration setting and averaged over three models for each $K$ to obtain the results. Matrix 25a shows a slightly higher success rate for the GMM with three components. Its mean success rate is 0.7186 versus 0.7052 for $K = 4$ and 0.6948 for the GMM 2. Additionally, it possesses the highest success rate in four out of seven iteration settings. All models outperform the baseline and all mixture models outperform a single normal distribution. Lastly, it is noteworthy that a GMM with four components shows slightly better results than using two components.

The findings regarding the success rate carry over to the percentage of desires fulfilled in Matrix 25a. Again, all learned models show stronger results than the baseline and the normal distribution cannot compete with mixture models. The GMM with three components outperforms the other networks, particularly for 50 and 200 iterations. This




(a) Success rates for different $K$.

(b) Desires fulfilled for different $K$.

Figure 25: Desires fulfilled and success percentage for different numbers of components $K$ in scenario 08. (a) shows the success rate, where the mixture models substantially outperform a single normal distribution. All learned models are superior to the baseline. These results are also reflected in the percentage of desires fulfilled (b). An average over three models is used except for the baseline.

results in the highest mean desires fulfilled percentage of 0.1524 versus 0.1419 for the GMM 2 and 0.1410 for $K = 4$.

Harnessing the results from the previous paragraphs, all other models are trained using a GMM with three components. The evaluation of whether centralized training improves the learning process therefore only uses $K = 3$ in the following section.

## 6.8 What is the effect of centralized training?

In Section 2.1.4, the *Centralized Training with Decentralized Execution* paradigm is introduced as a technique to stabilize training performance. Figure 26 shows the evaluation

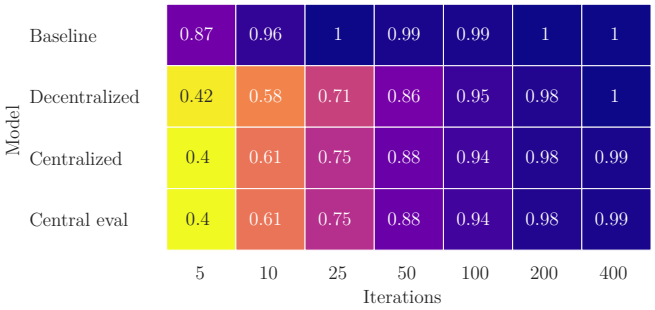



Figure 26: Model evaluations for centralized training and decentralized training on scenario 06. Both were trained using 100 iterations. Centralized training slightly outperforms decentralized training. Using centralized value estimates at evaluation time (*Central eval*) shows no success rate gains over decentralized evaluation. Plain MCTS is superior to both models.

results of models differing in their usage of centralized value targets.

During the training and evaluation of the guided search approach, all agents make decisions from their own point of view. In particular, they calculate value estimates and distributions conditioned on their own state. This produces an accurate ego reward estimate as both numerical states and a visual map is available to each agent. Centralized training in the context of this thesis refers to using only the ego reward estimates during training. An ego agent's reward estimates for the other vehicles are discarded, as they rely on incomplete information. After all, the ego agent only has a map from its own point of view at its disposal.

Centralized value targets minimally increase the mean success percentage by 0.76 percentage points (0.7857 to 0.7933). The increase is not consistent across iterations. Furthermore, adding centralized value targets at evaluation time yields an even smaller benefit of 0.05 percentage points. The final mean success rate improves from 0.7933 t 0.7938.

After centralized value targets have been shown to not be a key component to network training, the next ablation study examines whether learned value targets are needed at all.

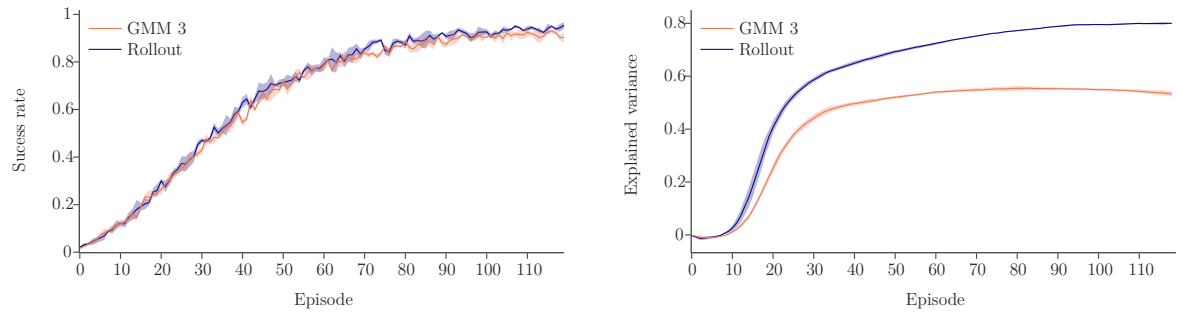
## 6.9 How important are the learned value estimates?

Both AlphaZero as well as the A0C algorithm on which this thesis is based use a network architecture with two heads. One head learns a policy, whereas the other head learns value estimates for environment states [74, 58]. This architecture has shown improved performance compared to separate networks for each task [75]. In the application to Go, truncating potentially long rollouts is reasonable, especially since a strong rollout policy can be expensive to compute [73].

Compared to learned value targets, using rollouts may be able to accelerate training progress by improving early state value estimates. The simulation policy used in the context of this thesis is able to choose from five basic actions: acceleration and deceleration without lateral movement, lane change to the left or right without change in longitudinal velocity and "doing nothing" (no velocity change in either direction). The choice between these actions is made by uniform sampling [46].

In an evaluation of whether the rollout policy described above is able to improve performance over learned value estimates, the first step comprises looking at the training progress. It is depicted in Figure 27a. Both value estimation methods show almost identical performance during the training.

A quantity which can be helpful for evaluating whether successful value learning occurs is the coefficient of determination or $R^2$. It is given in Definition 6.5 and explains how well predictions $\hat{y}_i$ explain the variation of a target variable $y_i$ [84]. For this reason it is




(a) Training success.

(b) Explained variance

Figure 27: (a) shows the training success of using MCTS rollouts versus learned value targets in scenario 08. Both models perform similarly. (b) depicts the explained variance. The rollout model converges faster and to a higher value. Both plots show one standard deviation error bands for three seeds.

also called *explained variance*.

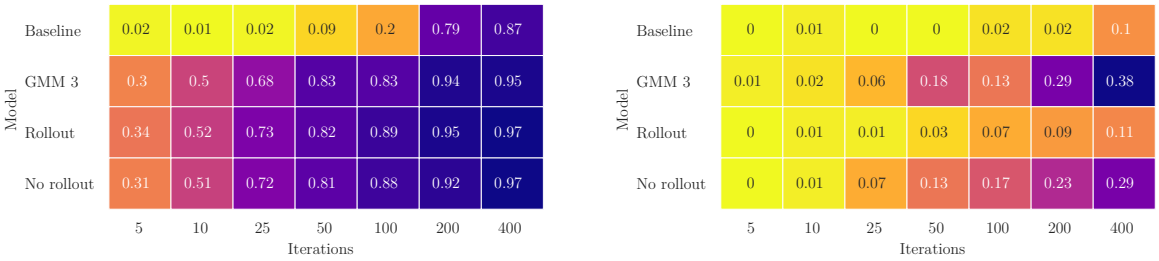
$$R^2 = 1 - \frac{RSS}{SYY} = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2} = 1 - \frac{\mathrm{Var}(\hat{V}(\mathbf{s}) - V_\theta(\mathbf{s}))}{\mathrm{Var}(\hat{V}(\mathbf{s}))} \,. \tag{6.5}$$

Here $RSS = \sum(y_i - \hat{y}_i)^2$ is the *residual sum of squares* and $SYY = \sum(y_i - \bar{y})^2$ is the *total sum of squares* [84]. Equation 6.5 then re-expresses $R^2$ in terms of the network output: value targets $\hat{V}(\mathbf{s})$ and network value estimates $V_\theta(\mathbf{s})$.

The explained variance for the guided search with and without MCTS rollouts is plotted in Figure 27b. Here one can see a clear differentiation between faster progress and higher final values for the rollout model compared to using network value estimates. This is to be expected, as the value targets $\hat{V}(\mathbf{s})$ are always produced from the same simulation policy when using rollouts. They are therefore stationary, as their distribution does not change over the course of the training. The training targets based on network value estimates as as described in Section 4.3.2 (Equation 4.30) on the other hand are non-stationary. The learning process changes their distribution during the training.

Does the effect described above impact the algorithm's performance in an evaluation setting? As shown in Figure 28a, MCTS simulations increase the success rate from 0.7186 (GMM 3) to 0.7467 (Rollout). The *Rollout* model uses simulations both at training as well as at evaluation time whereas the *No rollout* model is trained using simulations but does not perform them at evaluation time. Still, the No rollout model has a higher average success rate compared to the network using learned value targets (0.7319 to 0.7186).

In contrast to the results regarding the success rate, the ranking is flipped when analyzing the percentage of fulfilled desires. Here the GMM 3 has the highest average across different




(a) Rollout success rates.

(b) Rollout desires fulfilled.

Figure 28: Performance of different rollout strategies in scenario 08. GMM 3 performs no rollouts during training and evaluation. The *Rollout* model uses MCTS simulations as value targets during both training and evaluation. The *No rollout* model is trained with MCTS rollouts instead of network value estimates but uses the network during evaluation. Using MCTS simulations yields a slightly higher success rate but noticeably lower percentage of desires fulfilled. Results are averaged over three models.

iteration settings with 0.1524. This is followed by the model not using simulations at evaluation time (0.1290). Performing rollouts at training and test time yields an average rate of desires fulfilled of 0.0457.

The results of this section are inconclusive of whether learned value targets are required as a core component of the proposed algorithm or not. A last ablation study therefore follows next with an evaluation of which loss component drives the learning process.

## 6.10 Which loss components are critical for success?

Modifications to the value targets from the previous two sections have had minimal impact on training and evaluation performance. This motivates the question of whether the value loss is even needed for the proposed algorithm. In an ablation study in the following paragraphs, models are therefore trained using only either the policy loss or the value loss.

Figure 29 visualizes the success rate over the course of the training on scenario 06 with 100 iterations. It immediately stands out that using the value loss as the only objective results in no improvement at all. In comparison, the progress of the model trained only on the policy loss is indistinguishable from the runs minimizing the full objective detailed in Section 4.3.3.

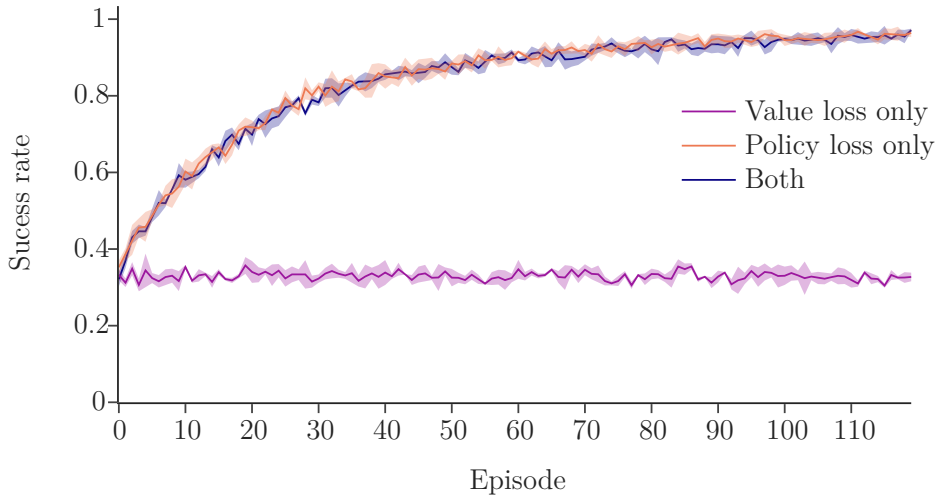
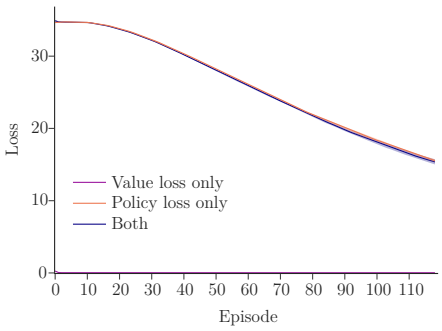
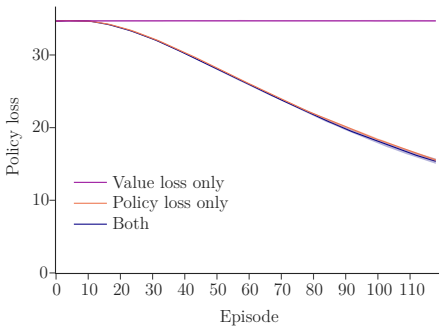


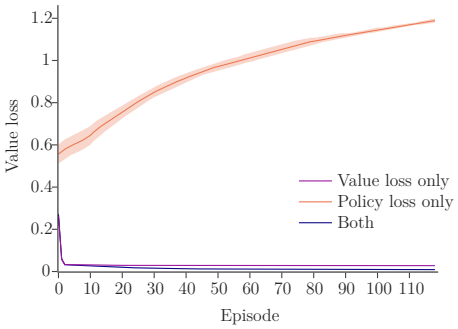

Figure 29: Training progress for models trained on different objective functions in scenario 06. Using only the policy loss progresses similarly to using the full objective. The model minimizing only the value loss shows no signs of improving. Runs are averaged over three seeds with one standard deviation shown.



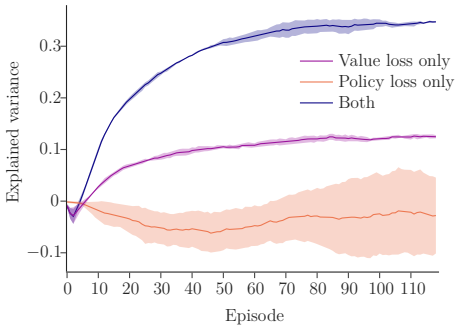

(a) Overall loss.




(b) Policy loss.




(c) Value loss.




(d) Explained variance.

Figure 30: Progress of different loss components over the course of the training on scenario 06. Three models are trained with one standard deviation shown. Models optimizing only one component are able to minimize the respective objective. The explained variance for the value loss only model is lower than for the network trained on Objective 4.33.

A closer examination is depicted in Figure 30. Runs where only one loss component is minimized show the desired behavior in Plots 30b and 30c: The loss is reduced for the target objective component and stays at the same level or increases for the other component.

Looking at the explained variance (see Equation 6.5) reveals that the model trained only on the policy loss hovers slightly below zero. This is to be expected, as random outputs should have no explanatory value in regards to the variation of the value targets. The network using only the value component as objective shows learning progress on a low level, converging to an $R^2$ of around 0.1. Training on the full loss improves upon this value by a factor of roughly $3\times$.

An interpretation of the results from this Section is given in Chapter 7.1. The evaluation concludes with an analysis of the generalization capabilities of the proposed approach.

## 6.11 How well can the learned policies generalize?

Overfitting to a specific environment is a prevalent problem in Deep Reinforcement Learning (DRL) agents [16]. Training and evaluation on the same environment as done in the previous sections is a conventional procedure for common benchmarks [16, 10]. Nevertheless, learning more abstract, general knowledge would be desirable. An agent that learns to stay in the lane and avoid obstacles should for instance be able to transfer these skills to similar environments. Whether the networks trained through the approach proposed by this thesis are able to do so is evaluated in the following section.

To benchmark generalization performance, the learned models are additionally evaluated on previously unseen scenarios. All scenarios are visualized in Appendix E. The baseline MCTS without guided search as well as an untrained model serve as standards for comparison.

| **Model**<br>**Scenario** | SC06 | SC08 | Reg. SC08 | SC06 & SC08 | SC06 & SC08 + Exp. | Random | Random + Exp. | Baseline |
|---|---|---|---|---|---|---|---|---|
| SC01 | 0.44 | 0.98 | 0.93 | 0.98 | **1.00** | 0.01 | 0.99 | **1.00** |
| SC02 | 0.27 | 0.72 | 0.83 | 0.78 | 0.95 | 0.01 | 0.90 | **0.97** |
| SC03 | 0.48 | 0.99 | 0.98 | 0.96 | **1.00** | 0.00 | **1.00** | **1.00** |
| SC04 | 0.10 | 0.82 | 0.87 | 0.60 | **1.00** | 0.00 | 0.99 | 0.99 |
| SC05 | 0.70 | 0.21 | 0.17 | 0.61 | 0.96 | 0.34 | 0.95 | **1.00** |
| SC06 | 0.99 | 0.39 | 0.32 | 0.90 | 0.98 | 0.31 | 0.98 | **0.99** |
| SC07 | 0.00 | 0.00 | 0.02 | 0.05 | 0.01 | 0.00 | 0.00 | **0.07** |
| SC08 | 0.39 | **0.84** | 0.81 | **0.84** | 0.54 | 0.00 | 0.04 | 0.20 |
| **Mean 01-05, 07** | 0.3317 | 0.6200 | 0.6333 | 0.6630 | 0.8200 | 0.0600 | 0.8050 | **0.8383** |
| **Mean unseen** | 0.3400 | 0.5871 | 0.5886 | 0.6630 | **0.8200** | 0.0838 | 0.7313 | 0.7775 |
| **Mean** | 0.4212 | 0.6187 | 0.6162 | 0.7150 | **0.8050** | 0.0838 | 0.7313 | 0.7775 |

Table 9: Generalization performance of different trained models measured by success rate. All are trained using 200 iterations. 100 different seeds are used in each evaluation with a randomly selected model. Evaluations are run using 100 iterations. Cyan marks a scenario an agent has been trained on whereas brown symbolizes an unseen setting. The best models are denoted in bold. The network trained on scenario 06 ("SC06") shows limited generalization capabilities but still improves over a random baseline ("Random"). A model trained on scenario 08 ("SC08") performs better on unseen scenarios. Adding BatchNorm and LayerNorm has little effect ("Reg. SC08"). Training a network on both scenarios ("SC06 & SC08") yields a higher mean success rate and more consistent generalization for scenarios 04 and 05. Adding three pre-calculated actions (lane change left, lane change right, no change) improves the performance of the random model ("Random + Exp.") past the trained models . A combination of these actions together with the model trained on scenario 06 and scenario 08 yields the highest average success rate ("SC06 & SC08 + Exp.").

The results are reported in Table 9, where unseen scenarios are shaded in brown and scenarios which have been trained on are marked with cyan.

Looking at the success rates, the models trained on scenario 06 ("SC06") and 08 ("SC08") are able to show improved performance in unseen scenarios compared to a random model ("Random"). The network trained on scenario 08 demonstrates evaluation results more in line with the pure MCTS baseline in scenarios 01, 02, 03 and 04. Its performance in scenarios 05, 06 and 07 remains subpar. Particularly in scenario 06, SC08 is not able to improve considerably over a random network.

Adding batch normalization [41] and layer normalization [3] during training ("Reg. SC08") minimally boosts generalization performance. At the same time, the normalization schemes decrease the success rate on the scenario that is being trained on.

Using two scenarios in the training process ("SC06 & SC08") has added benefits apart from the performance on seen tasks: The combined performance on the unseen scenarios 04 and 05 is the highest with 0.605 compared to the next best model ("SC08") at 0.515. Success rate on the training tasks stays similar to the networks trained on a single scenario.

Section 6.2 demonstrates the benefits of using pre-calculated maneuvers. Incorporating such maneuvers into the guided search approach is therefore a reasonable thing to try. However, care must be taken during network training: If the heuristic is enabled during the training phase, the model focuses on learning the pre-calculated actions. This is undesirable, as they are already added to each node anyway. Additionally, a pre-calculated lane change may introduce extreme actions at the border of the action space. The arising large negative log-probabilities negatively affect training stability.

For the experiments with pre-calculated maneuvers, three basic actions are added: lane change left, lane change right, and no change. The results when using an enhanced random model are reported in Table 9 under "Random + Exp.". Simply adding these three basic actions leads to an orders of magnitude improvement in success rate. "Random + Exp." even outperforms the "SC06 & SC08" model on average. The increase is particularly noticeable in scenarios 01-06. Adding the same maneuvers to the "SC06 & SC08" network yields the model with the highest overall success rate ("SC06 & SC08 + Exp."). It inherits the strength of the baseline MCTS on scenarios 01-06 while improving its performance in scenario 08. The success rate of "SC06 & SC08 + Exp." on this task however falls short of the models without heuristic expansion ("SC08" and "Reg. SC08"). Lastly, scenario 07 is inherently difficult with no approach achieving a success percentage of above 10%.

# 7 Discussion

The previous chapter has given detailed results on the empirical performance of the proposed approach. These are interpreted in the following section. Next, the limitations of this thesis are discussed before an outlook provides paths for future research.

## 7.1 Findings

Reflecting the experimental results in the previous chapter, it stands out that the baseline and the guided search perform well in different settings. Scenario 06 is easily solvable by pre-calculated basic actions as evidenced by the strong Monte Carlo tree search (MCTS) performance with just five iterations (refer to Table 2). On the other hand, scenario 08 seems to require more nuanced interaction between the agents and driving between two lanes. Here the guided search provides real benefits and can even realize a speed-up in wall clock time.

Looking closer at the findings of Section 6.5, it emerges that the attained cooperative reward has no consistent relationship with the success rate. This is surprising, as both crashes and driving off the road are penalized heavily (see Appendix F). It appears as if there are situations in which an agent may prefer to terminate the episode early instead of trying to resolve the situation.

Contrary to the usual choice of selection based on visitation counts [75, 58], the proposed approach performs better when selecting the action with the highest action value. This is in line with the baseline MCTS. One possible hypothesis explaining these results is that selection using the maximum action value is optimistic. Such behavior is problematic in competitive settings, where it can lead to exploitable policies. However, the tasks in this thesis are cooperative in nature. Thus agents have an incentive to accommodate behavior of other agents instead of exploiting it.

Overall, the training process of the proposed approach is remarkably stable, which is not necessarily a given in Deep Reinforcement Learning (DRL) methods [32]. Various techniques which suggest improved final performance like using MCTS rollouts, centralized value targets or iteration randomization[25] neither increase nor decrease the success rate substantially. Only using a Gaussian Mixture Model (GMM) instead of a normally distributed policy leads to noticeable success rate gains as analyzed by Section 6.7. This seems plausible given the motivation for using mixture models from Chapter 2.3.4.

Since the modifications described as having no effect in the previous paragraph all relate to value targets, it comes as no surprise that the learning is driven by the policy loss. Models

[25] See Appendix L for a short treatise of iteration randomization.

trained using only the value loss show no learning progress at all. Yet this contradicts findings in the literature, where the value component of the objective function was found to be more important [82]. Of course, there are significant differences in the environments: Board games have discrete action spaces compared to the continuous control scenarios introduced in Chapter 4.1. Additionally, domains like Go require building deeper trees to obtain a sparse reward rather than the wide trees with dense reward produced by the environments in this thesis. In conclusion, the experiment reveals that guiding the MCTS using a learned distribution is the important component of the model. Utilizing network value estimates has no further benefit.

Evaluating the generalization performance of trained networks shows higher success rates on unseen scenarios compared to a random model. The gains are more pronounced if the unseen task is similar to the scenario which is used during training. A network trained on scenario 06 for instance generalizes well to scenario 05, which has a setup closely resembling scenario 06. For reference, all scenarios are visualized in Appendix E. An exciting result relates to training on multiple scenarios, which leads to improved performance on unseen tasks in Table 9.

Motivated by the success of the baseline, actions determined through a heuristic are added to the guided search. This combination results in the algorithm with the highest overall success rate. However, one must be careful to not add too many pre-calculated actions. Adding more than the three basic maneuvers outlined in Section 6.11 reduces the average success rate again as it inhibits sampling from the network's proposal distribution.

Nevertheless, the approach proposed in this thesis does not come without limitations. These are discussed in the following section.

## 7.2 Limitations

Regarding the limitations of this thesis, the computational demands stand out in particular. Models trained using more iterations often need more than a day of training time. This makes it challenging to scale up the algorithm and either include several scenarios or use more difficult tasks which require a higher number of training iterations. In supervised learning, the scale of the models and training process is what drives computational demands (e.g. in [19]). Compared to that, the algorithm introduced in the previous chapters is bottlenecked by training data generation using network inference. As this work is a proof of concept, the aforementioned inference is not optimized for throughput speed. This in turn limits the scale at which the system can be trained.

The high demands in terms of compute also have trickle down effects concerning specific algorithm choices. For instance, one reason why the guided search does not perform better

on some scenarios might be the coarse resolution of the map. This could lead to vehicles unnecessarily colliding with obstacles. Enhancing map size however leads to a significant increase in training time. As an example, doubling the resolution requires approximately 2.5× more wall clock time.

In terms of real world deployment, the proposed approach is limited due to the slow inference speed of the network. To be competitive in terms of wall clock time, the guided search must use significantly less iterations than the baseline MCTS. While this is achieved in scenario 08, scenario 06 requires too many iterations for the proposed model to recover the performance of the baseline. As a consequence, the guided search is orders of magnitude slower for the same success rate.

Also related to the available resources is hyperparameter tuning. The approach described in this thesis has around 170 parameters to optimize[26], not including the network architecture. Optimizing performance in such a huge search space is difficult. One can therefore assume with high confidence that the used parameters are suboptimal.

The generalization capabilities of the model introduced by this thesis are highly task-dependent. The network trained on scenario 08 for example generalizes to scenarios 01-04. On the contrary, the model trained on scenario 06 also gains performance in scenario 05. What determines generalization success on unseen tasks? One might first assume it is the difficulty of the scenario. Upon closer inspection however, scenarios 01-04 and 08 have two lanes while scenarios 05 and 06 have three lanes. It is thus a reasonable hypothesis to conclude that models are only able to transfer experience to tasks with a similar number of lanes. This would preclude the learning of truly general driving skills.

Expounding the findings of the previous paragraph, it is fair to question the generality of the obtained results. Due to computational constraints, ablation studies are only performed in two scenarios. But as discussed, performance may be task dependent. More studies are therefore needed to examine whether the results and ablation studies are translatable to other scenarios and settings.

Lastly, looking at the proposed approach it stands out that the MCTS requires knowledge of the environment's transition function. As discussed in Section 3, this assumption is restrictive and limits the applications of the algorithm. In addition, a learned forward dynamics model could also increase the generalization capacities of this work's method.

Many of the limitations discussed in the current section provide avenues for future research. The chapter therefore concludes with an outlook on some particularly fruitful directions.

[26]The exact number depends on the settings. As an example, enabling $\varepsilon$ action selection requires setting additional parameters which determine the decay.

## 7.3 Outlook

Recapping the previous section, increasing the speed at which training data is generated emerges as a stream for future work. Here three orthogonal directions are equally viable: First it is possible to increase throughput via improved engineering. Techniques such as quantization in PyTorch [63] or Nvidia's TensorRT [61] are able to improve network inference speed on CPUs by factors of 2× up to 4×.

Another possibility is improving the architecture of system parts. Distilling the learned network [69] or using an additional smaller policy in parts of the MCTS [49] are some conceivable modifications. A large stream of current research in the deep learning community is made of more efficient Transformer networks: Models like the Linformer or Performer are able to better trade off speed versus accuracy and are therefore prime candidates to improve the network architecture of this work as well [79].

Lastly, a third direction consists of modifications to the MCTS itself. Parallelizing the search is a well established research direction [13] and has already been extended to continuous action spaces for the baseline used in this thesis [48]. Recently, more sophisticated approaches have been developed and shown larger performance gains [55]. MCTS parallelization in connection with guided search might however be non-trivial to implement efficiently. Another method is therefore to be more selective with network evaluations, for instance by evaluating only the root node instead of all expanded nodes [47].

Raising the system's scalability through improved engineering or more performant individual components seems dull and uninspiring at first. After all, it symbolizes incremental progress instead of an ingenious breakthrough. However, simply increasing the scale of learned models has led to important advances in research, for instance with AlexNet [44]. Phenomena emerging from large-scale training also power some of the biggest success stories in natural language processing [11].

A further pathway for future research is provided through algorithms which utilize a learned model. This could be by either performing tree search in a compact latent space [70] or by combining Reinforcement Learning (RL) with a learned model in other ways [34]. As exact knowledge of the environment's transition function is a restrictive assumption, using a learned model instead will provide routes for a more general application of the proposed approach.

Finishing up this chapter, the evaluation results in Section 6.11 hint at another rewarding direction: the combination of learned networks together with other forms of knowledge. Clearly integrating a heuristic into the guided search presented by this thesis is only a minuscule step towards such a concept. In the future, one can however envision systems that reason more abstractly about the underlying causes or constraints of an environment.

# 8 Conclusion

This thesis presents an approach for guiding the sampling of a Monte Carlo tree search (MCTS) based planner through a neural network learned by Reinforcement Learning (RL). The model is able to succesfully predict maneuvers in a continuous action space via learned proposal distributions. As the planner is used in cooperative multi-agent driving scenarios, a novel network based on the Transformer architecture is developed. It is able to learn interactions between a flexible number of agents from a hybrid numerical and visual input representation. An empirical evaluation shows that the proposed approach is able to improve sample efficiency and wall clock speed in a challenging multi-agent driving scenario over a pure MCTS baseline.

Interpreting the agents within a scenario as a sequence of objects allows the application of a Transformer encoder to learn interactions between a flexible number of agents. This newly developed network architecture is trained starting from *tabula rasa* by extending the loss function of an AlphaZero-inspired algorithm (A0C) to multi-agent settings. Action bounds are enforced by using a policy consisting of mixtures of transformed normal distributions. Once trained, the network is able to effectively predict trajectories for all vehicles in a scenario using only the incomplete sensory input of a single agent.

An empirical analysis shows that the guided search using a neural network is able to successfully recover the performance of a baseline augmented with domain-specific heuristics. In a more challenging scenario, the proposed approach achieves competitive results to the MCTS while using 8× fewer iterations and less wall clock time. Experiments disclose a stable learning process for a variety of modifications. Additional ablation studies reveal that learning in continuous domains is driven by the policy objective, which is in contrast to existing findings in board games. The networks trained by the proposed algorithm are able to generalize knowledge to unseen scenarios with some success and improve substantially over a random network. Finally, augmenting the guided search with heuristics yields a method with the strongest overall performance and a high success rate on all but one task.

# A ECA automatic kernel size

The authors state two assumptions before deriving their formula for the automatic kernel size [83]:

1. A linear mapping $\phi$ between number of channels $C$ and kernel size $k$ like $C = \phi(k)$ is too restrictive to capture the relationship.
2. Channel dimensions in Convolutional Neural Networks (CNNs) are usually set to powers of two, e.g. 16, 32, 64.

The ECA paper therefore defines the following exponential relationship [83]:

$$C = \phi(k) = 2^{\gamma \cdot k - b} \,, \tag{A.1}$$

where $\gamma$ and $b$ are constants to be chosen by the user.
Equation A.1 can now be solved for $k$:

$$\begin{aligned} & C = 2^{\gamma \cdot k - b} \\ \Leftrightarrow \quad & \ln C = (\gamma \cdot k - b) \cdot \ln 2 \\ \Leftrightarrow \quad & \frac{\ln C}{\ln 2} = \gamma \cdot k - b \\ \Leftrightarrow \quad & \log_2 C + b = \gamma \cdot k \\ \Leftrightarrow \quad & \frac{\log_2 C}{\gamma} + \frac{b}{\gamma} = k \end{aligned} \tag{A.2}$$

Rounding Formula A.2 to the nearest odd number $|\cdot|_{odd}$ and plugging in $\gamma = 2$ and $b = 1$ yields a mapping $\psi(C)$ from channels to kernel size [83]:

$$k = \psi(C) = \left| \frac{\log_2 C + 1}{2} \right|_{odd} . \tag{A.3}$$

The result in Equation A.3 is used by the authors in all their experiments.

# B Proof of Proposition 1

The starting point is to show some helpful properties of the scaled tanh transformation.

**Lemma 1.** *The scaled* tanh *Transformation* $y = c\tanh(x)$ *is bijective. Its inverse is* $x = \operatorname{artanh}\left(\frac{1}{c}y\right)$.

*Proof.* The first step is to prove that the scaled tanh transformation is invertible.

$$\begin{aligned} & y = c\tanh(x) \\ \Leftrightarrow \quad & x = \tanh^{-1}\left(\frac{1}{c}y\right) \\ \Leftrightarrow \quad & x = \operatorname{artanh}\left(\frac{1}{c}y\right) \\ \Leftrightarrow \quad & x = \frac{1}{2}\log\left(\frac{1+\frac{1}{c}y}{1-\frac{1}{c}y}\right) \end{aligned} \tag{B.1}$$

Since the range of the tanh is $(-1, 1)$, the range of the re-scaled tanh is $(-c, c)$. Thus the domain of the inverse function is $y \in (-c, c)$. Therefore $\frac{1+\frac{1}{c}y}{1-\frac{1}{c}y} > 0$ and the function is defined on $(-c, c)$.
In the second step it has to be shown that $g^{-1}(g(x)) = x$. This can be done by simply plugging in the definition of the scaled tanh transformation and simplifying

$$\begin{aligned} g^{-1}(g(x)) &= \frac{1}{2}\log\left(\frac{1+\frac{1}{c}c\tanh(x)}{1-\frac{1}{c}c\tanh(x)}\right) \\ &= \frac{1}{2}\log\left(\frac{1+\tanh(x)}{1-\tanh(x)}\right) \\ &= \frac{1}{2}\log\left(\frac{1+\frac{e^{2x}-1}{e^{2x}+1}}{1-\frac{e^{2x}-1}{e^{2x}+1)}}\right) \\ &= \frac{1}{2}\log\left(\frac{\frac{e^{2x}+1}{e^2x+1}+\frac{e^{2x}-1}{e^{2x}+1}}{\frac{e^{2x}+1}{e^{2x}+1}-\frac{e^{2x}-1}{e^{2x}+1}}\right) \\ &= \frac{1}{2}\log\left(\frac{\frac{e^{2x}+1+e^{2x}-1}{e^{2x}+1}+}{\frac{e^{2x}+1-e^{2x}+1}{e^{2x}+1}}\right) \\ &= \frac{1}{2}\log\left(\frac{e^{2x}+1+e^{2x}-1}{e^{2x}+1-e^{2x}+1}\right) \\ &= \frac{1}{2}\log\left(\frac{2e^{2x}}{2}\right) \\ &= \frac{1}{2}\log\left(e^{2x}\right) \\ &= x \end{aligned} \tag{B.2}$$

□

**Lemma 2.** *The scaled* tanh *Transformation* $y = c\tanh(x)$ *is continuously differentiable. Its derivative is* $\frac{d}{dx}c\tanh(x) = c - c\tanh^2(x)$.

*Proof.* The result follows directly from the derivative of the tanh function, which is $\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$. □

**Remark 1.** *Since the scaled* tanh *transformation is applied element-wise, these results hold in the multivariable case.*

While lemmata 1 and 2 are rather trivial, their results are nevertheless needed to proceed further and apply the multivariable inverse function theorem to the scaled tanh transformation.

**Proposition 1.** *The Probability Density Function (PDF) of the transformed density can be expressed as*

$$\pi(\mathbf{a}|\mathbf{s}) = \mu(\mathbf{u}|\mathbf{s}) \left| \det \frac{d\mathbf{a}}{d\mathbf{u}} \right|^{-1} , \tag{4.16}$$

*with*

$$\left| \det \frac{d\mathbf{a}}{d\mathbf{u}} \right| = c^D \prod_{i=1}^{D} 1 - \tanh^2(u_i) \ . \tag{4.17}$$

*Proof.* First define $\varphi(x) = \tanh^{-1}\left(\frac{1}{c}x\right)$ using Lemma 1. The Cumulative Distribution Function (CDF) of the transformed random variable $\mathbf{a} = c\tanh(\mathbf{u})$ can then be written as

$$\int_{-c}^{c} \pi(\mathbf{a}|\mathbf{s})\, d\mathbf{a} = \int_{\varphi(-c)}^{\varphi(c)} \mu(\mathbf{u}|\mathbf{s})\, d\mathbf{u} \tag{B.3}$$

Application of the change of variables to the integral in Formula B.3 yields

$$\int_{\varphi(-c)}^{\varphi(c)} \mu(\mathbf{u}|\mathbf{s}) d\mathbf{u} = \int_{-c}^{c} \mu(\varphi(\mathbf{a}|\mathbf{s})) \left| \det \frac{d\varphi(\mathbf{a})}{d\mathbf{a}} \right| d\mathbf{a} \tag{B.4}$$

Lemma 2 allows the application of the multivariable inverse function theorem to the term $\frac{d\varphi(\mathbf{a})}{d\mathbf{a}}$. It can express the derivative of the inverse function $\varphi$ in terms of the scaled tanh transformation $\mathbf{a} = c\tanh\mathbf{u}$

$$\frac{d\varphi\mathbf{a}}{d\mathbf{a}} = \frac{d\tanh^{-1}(\frac{1}{c}\mathbf{a})}{d\mathbf{a}} = \left( \frac{d\, c\tanh\mathbf{u}}{d\mathbf{u}} \right)^{-1} = \left( \frac{d\mathbf{a}}{d\mathbf{u}} \right)^{-1} . \tag{B.5}$$

For this result to hold, the Jacobian $\frac{d\mathbf{a}}{d\mathbf{u}}$ has to be invertible. Since the scaled tanh transformation is applied element-wise, $\frac{d\mathbf{a}}{d\mathbf{u}}$ is a diagonal matrix with the single-variable derivative $\frac{\partial \mathbf{a}}{\partial u_i} = c - c\tanh^2(u_i)$ of the transformation on the diagonal. Its determinant is thus

$$\begin{aligned} \left| \det \frac{d\mathbf{a}}{d\mathbf{u}} \right| &= \prod_{i=1}^{D} c(1 - \tanh^2(u_i)) \\ &= c^D \prod_{i=1}^{D} 1 - \tanh^2(u_i) \end{aligned} \tag{B.6}$$

From $\tanh^2(u_i) \in (-1, 1)$ it follows that $c - c\tanh^2(u_i) > 0$. This is enough to show that $\frac{d\mathbf{a}}{d\mathbf{u}}$ is a positive definite matrix and thus invertible everywhere, allowing the substitution of Expression B.5 into Equation B.4. Swapping the inverse and determinant is now the only thing needed to arrive at Formula 4.16

$$\mu(\mathbf{u}|\mathbf{s})\left|\det\left(\frac{d\mathbf{a}}{d\mathbf{u}}\right)^{-1}\right| = \mu(\mathbf{u}|\mathbf{s})\left|\det\frac{d\mathbf{a}}{d\mathbf{u}}\right|^{-1}. \tag{B.7}$$

□

# C Squashed normal log probability correction

The correction formula for the squashed normal log-likelihood follows from iterative application of the logarithm rules:

$$\begin{aligned}
\log\pi(\mathbf{a}|\mathbf{s}) &= \log\left(\frac{\mu(\mathbf{u}|\mathbf{s})}{c^D\prod_{i=1}^{D}1-\tanh^2(u_i)}\right)\\
&= \log\mu(\mathbf{u}|\mathbf{s}) - \log\left(c^D\prod_{i=1}^{D}1-\tanh^2(u_i)\right)\\
&= \log\mu(\mathbf{u}|\mathbf{s}) - \left(\log c^D + \log\prod_{i=1}^{D}1-\tanh^2(u_i)\right)\\
&= \log\mu(\mathbf{u}|\mathbf{s}) - \left(D\log c + \sum_{i=1}^{D}\log\left(1-\tanh^2(u_i)\right)\right)\\
&= \log\mu(\mathbf{u}|\mathbf{s}) - D\log c - \sum_{i=1}^{D}\log\left(1-\tanh^2(u_i)\right)
\end{aligned} \tag{C.1}$$

# D Numerically stable log probability formula

Before starting to derive the more stable formula it helps to review some useful definitions and identities:

1. The *Softplus* function is given as $\text{Softplus}(x) = \log(1 + e^x)$.
2. The hyperbolic secant is given as

   $$\text{sech}(x) = \frac{1}{\cosh(x)} = \frac{2e^{-x}}{e^{-2x}+1}$$

   Here the definition of $\cosh(x) = (e^{-2x}+1)/2e^{-x}$ is used.
3. $\tanh^2(x) + \text{sech}^2(x) = 1$, so $\text{sech}^2(x) = 1 - \tanh^2(x)$.

The goal here is to reformulate the entropy correction term. Using 2) and 3) yields:

$$\begin{aligned} -\sum_{i=1}^{D} \log\Big(1-\tanh^2(u_i)\Big) &= -\sum_{i=1}^{D} \log\Big(\operatorname{sech}^2(u_i)\Big) \\ &= -2\cdot\sum_{i=1}^{D} \log\Big(\operatorname{sech}(u_i)\Big) \\ &= -2\cdot\sum_{i=1}^{D} \log\left(\frac{2e^{-u_i}}{e^{-2u_i}+1}\right) \end{aligned} \tag{D.1}$$

After this key step is done, Expression D.1 can be simplified by applying logarithm rules:

$$\begin{aligned} -2\cdot\sum_{i=1}^{D} \log\Big(\operatorname{sech}(u_i)\Big) &= -2\cdot\sum_{i=1}^{D} \log\left(\frac{2e^{-u_i}}{e^{-2u_i}+1}\right) \\ &= -2\cdot\sum_{i=1}^{D} \Big(\log(2e^{-u_i}) - \log(e^{-2u_i}+1)\Big) \\ &= -2\cdot\sum_{i=1}^{D} \Big(\log 2 + \log e^{-u_i} - \log(e^{-2u_i}+1)\Big) \\ &= -2\cdot\sum_{i=1}^{D} \Big(\log 2 - u_i - \log(e^{-2u_i}+1)\Big) \end{aligned} \tag{D.2}$$

Now the definition of the Softplus function can be inserted into Equation D.2:

$$-2\cdot\sum_{i=1}^{D} \Big(\log 2 - u_i - \log(e^{-2u_i}+1)\Big) = -2\cdot\sum_{i=1}^{D} \Big(\log 2 - u_i - \operatorname{Softplus}(-2u_i)\Big) \tag{D.3}$$

D.3 is the formula commonly seen in code repositories. Below is an example snippet from OpenAI's SpinningUp library [1]:

```
import numpy as np
import torch.nn.functional as F
# pi_distribution is a pytorch normal distribution object
logp_pi = pi_distribution.log_prob(pi_action).sum(axis=-1)
# Correct the log probabilities by the formula derived above
logp_pi -= (
        2*(np.log(2) - pi_action - F.softplus(-2*pi_action))
    ).sum(axis=1)
```

# E Evaluation scenarios

This section briefly describes all the cooperative driving scenarios used for evaluation.

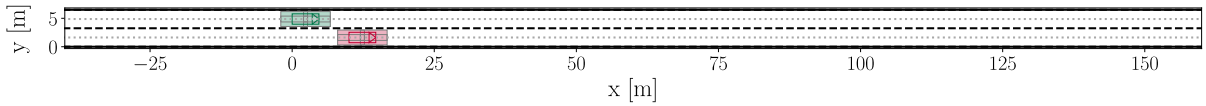



Figure 31: Scenario 01. The red agent wants to merge onto lane one, where the faster green vehicle wants to keep the lane.

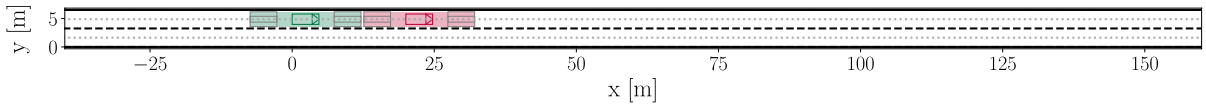



Figure 32: Scenario 02. Both vehicles want to keep the lane, but the green agent is faster and has a higher desired velocity than the red agent.

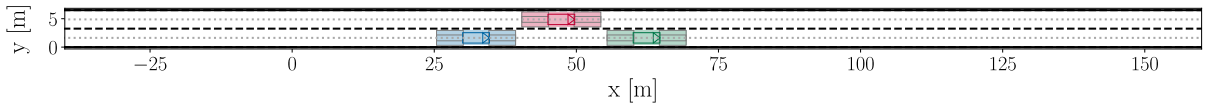



Figure 33: Scenario 03. All agents have the same target velocity, but the red agent wants to switch lanes and must merge between the other two vehicles.

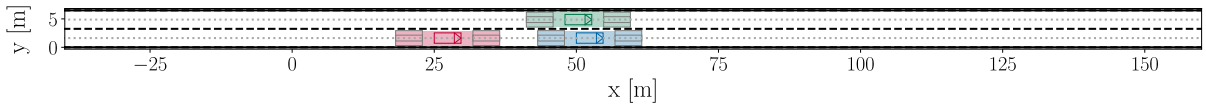



Figure 34: Scenario 04. Similar to the previous scenario, the green agent wants to merge onto lane zero, where two vehicles are approaching. To complete the maneuver, it must either accelerate or decelerate.

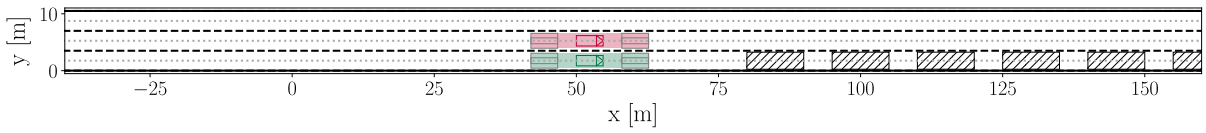



Figure 35: Scenario 05. The red vehicle must react to the green vehicle's lane change due to avoiding the obstacles on lane zero.

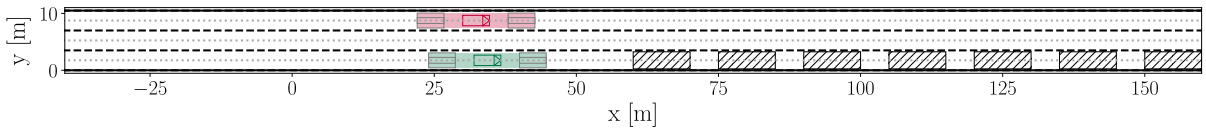



Figure 36: Scenario 06. The red agent wants to merge onto lane one at the same time as the green vehicle must avoid the obstacles.

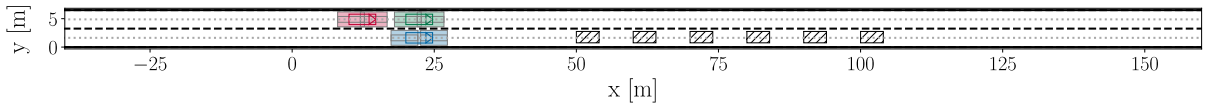



Figure 37: Scenario 07. The blue agent must merge onto lane one to avoid the obstacles in its path. However, the lane is already occupied by two approaching vehicles.

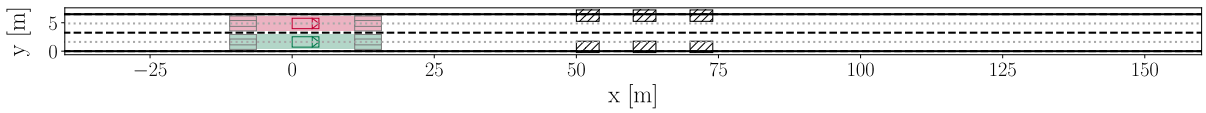



Figure 38: Scenario 08. Both agents must cooperate and merge into the middle to avoid obstacles in a bottleneck.

# F Reward parameters

Table 10 shows the values of all reward weight coefficients described in Section 4.1.2. While they can be adjusted individually for each scenario, all parameters are kept fixed for this work.

| Notation | Parameter | Value |
|---|---|---|
| $w_{LC}$ | Lane change weight | $-10.0$ |
| $w_{AX}$ | Longitudinal acceleration weight | 0.0 |
| $w_{AY}$ | Lateral acceleration weight | $-5.0$ |
| $w_{VD}$ | Velocity deviation weight | 500.0 |
| $w_{LD}$ | Lane deviation weight | 100.0 |
| $w_{LCD}$ | Lane center deviation weight | 85.0 |
| $w_{IS}$ | Punishment for invalid states | $-1000.0$ |
| $w_C$ | Punishment for collisions | $-1000.0$ |
| $w_{IA}$ | Punishment for invalid actions | 0.0 |
| $\lambda_i$ | Cooperation factor | 0.5 |

Table 10: Constants in the reward function and their settings.

# G Network architecture

This appendix gives the full PyTorch printout of the network architecture as described in Section 4.5. While it is more detailed than simply listing parameters in a table it allows for exact re-implementation of the model.

Embedding architecture. Numerical agent states are vectors of length 60.

```
(embedding): class=LinearEmbedding, input_dim=60, embedding_dim=64,
embed_dropout=False, positional_encoding=Embedding(8, 64), device=gpu
```

Transformer architecture with four encoder layers and $d_{model} = 64$. Each layer has two heads. Because the layers are identical only the first one is shown.

```
TransformerEncoder(
    (layers): ModuleList(
      (0): TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): _LinearWithBias(in_features=64, out_features=64,
          bias=True)
        )
        (linear1): Linear(in_features=64, out_features=256, bias=True)
        (dropout): Dropout(p=0.0, inplace=False)
        (linear2): Linear(in_features=256, out_features=64, bias=True)
        (norm1): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.0, inplace=False)
        (dropout2): Dropout(p=0.0, inplace=False)
      )
    )
  )
```

ResNet architecture used in the convolutional tower of this work. It consists of an initial convolution with max pooling followed by three basic convolutional blocks. The output is reduced to a vector with 128 elements by a fully convolutional head.

```
AttentionResNet(
    (trunk): Sequential(
      (block1): BasicBlock(
        (nonlinearity): ReLU()
        (conv1): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2),
        padding=(1, 1), bias=False, padding_mode=reflect)
        (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1),
        padding=(1, 1), bias=False, padding_mode=reflect)
        (channel_attention): ECALayer(
          (avg_pool): AdaptiveAvgPool2d(output_size=1)
          (conv): Conv1d(1, 1, kernel_size=(3,), stride=(1,), padding=(1,),
          bias=False)
```

```
      )
      (projection): Conv2d(16, 32, kernel_size=(1, 1), stride=(2, 2),
      bias=False)
    )
    (block2): BasicBlock(
      (nonlinearity): ReLU()
      (conv1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2),
      padding=(1, 1), bias=False, padding_mode=reflect)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
      padding=(1, 1), bias=False, padding_mode=reflect)
      (channel_attention): ECALayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (conv): Conv1d(1, 1, kernel_size=(3,), stride=(1,), padding=(1,),
        bias=False)
      )
      (projection): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2),
      bias=False)
    )
    (block3): BasicBlock(
      (nonlinearity): ReLU()
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
      padding=(1, 1), bias=False, padding_mode=reflect)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
      padding=(1, 1), bias=False, padding_mode=reflect)
      (channel_attention): ECALayer(
        (avg_pool): AdaptiveAvgPool2d(output_size=1)
        (conv): Conv1d(1, 1, kernel_size=(5,), stride=(1,), padding=(2,),
        bias=False)
      )
      (projection): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
      bias=False)
    )
  )
  (inital_conv_pool): Sequential(
    (conv7x7): Conv2d(2, 16, kernel_size=(3, 3), stride=(1, 1),
    padding_mode=reflect)
    (norm): Identity()
    (nonlinearity): Hardswish()
    (pool3x3): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
    ceil_mode=False)
  )
  (fcn_head): Sequential(
    (avgpool): AdaptiveAvgPool2d(output_size=1)
    (1x1conv1): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1),
    bias=False)
    (nonlinearity1): Hardswish()
    (1x1conv2): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1),
```

```
      bias=False)
      (nonlinearity2): Hardswish()
    )
  )
```

The policy is a four-layer Multilayer Perceptron (MLP).

```
(policy): class=DiagonalGMMPolicy,components=3, state_dim=192, action_dim=2,
action_bound=5, log_std_bounds=(-5, 2),
hidden_layers=4, hidden_units=[1024, 512, 512, 256],
nonlinearity=ReLU, layernorm=False
```

# H Hyperparameters

Table 11 lists the relevant MCTS parameters used to obtain the results from Chapter 6. Hyperparameters for the RL models were tuned by hand. A setting is deemed relevant if it differs from the baseline settings or if it is of importance to the RL training procedure.

| Notation | Parameter | Value |
|---|---|---|
| $\gamma$ | Discount factor | 0.7 |
| $C_{uct}$ | Initial UCT | 200 |
| $C_{uct}$ | UCT constant | 4 |
| | Move grouping | `False` |
| | $\varepsilon$-greedy selection | `False` |
| | Action noise | `False` |
| | Blind values | `False` |
| | Expansion policy | *expansionUCT* |
| | Final selection | *continuousMaxActionValue* |
| $C_{pw}$ | Progressive widening (PW) coefficient | 1 |
| $\alpha_{pw}$ | PW exponent | 0.7 |
| | Predetermined actions | `False` |
| | Numerical state history length | 8 |
| $\min(r)$ | Minimum unnormalized reward | $-2603.0$ |
| $\max(r)$ | Maximum unnormalized reward | 685.0 |
| $d$ | Reward interval diameter | 3288.0 |
| | Minimum scaled reward | $-1$ |
| | Maximum scaled reward | 1 |
| | Maximum longitudinal range | $128m$ |
| | Maximum lateral range | $12.8m$ |
| | Map size $C \times H \times W$ | $2 \times 32 \times 64$ |
| | $x$ axis scaling | $0.5\times$ |
| | $y$ axis scaling | $2.5\times$ |
| | Max agents | 8 |
| | Max speed | $36m/s$ |
| | Max vehicle length | $5m$ |
| | Max vehicle width | $2.2m$ |
| | Max lanes | 4 |

Table 11: MCTS parameter settings for the RL algorithm.

The training parameters for the RL algorithm are listed in Table 12. For the model trained on both scenario 06 and scenario 08, $D = 240000$ and $T = 20000$.

| Notation | Parameter | Value |
|---|---|---|
| $E$ | Training episodes | 120 |
| $D$ | Replay buffer size | 120000 |
| $T$ | Samples per episode | 10000 |
| $P$ | Training epochs | 1 |
| $B$ | Batch size | 1024 |
| | Centralized value estimates | `True` |
| | Iteration randomization | `False` |
| | Rollouts | `False` |
| | Optimizer | SGD |
| $\lambda$ | Learning rate | 0.001 |
| | Momentum | 0.9 |
| | Weight decay | 0.00002 |
| | Gradient max value | 10.0 |
| $\tau$ | Target temperature | 10.0 |
| | Loss policy coefficient | 1.0 |
| | Loss value coefficient | 1.0 |
| $\alpha$ | Loss entropy coefficient | 0.2 |
| | Scalar reduction | `action` |

Table 12: Training hyperparameters for the RL algorithm.

# I Baseline hyperparameters

The baseline MCTS parameters were optimized using Bayesian optimization in an unpublished work. Relevant values are listed in the following table. Move grouping settings refer to the semantic action groups. More information on them can be found in [46].

| Notation | Parameter | Value |
|---|---|---|
| $\gamma$ | Discount factor | 0.7 |
| | Initial UCT | 200 |
| $C_{uct}$ | UCT constant | 4.0 |
| | $\varepsilon$-greedy selection | `False` |
| | Action noise | `False` |
| | Blind values | `True` |
| | # BV candidate samples | 100 |
| | Expansion policy | *expansionUCT* |
| | Final selection | *continuousMaxActionValue* |
| $C_{pw}$ | Progressive widening (PW) coefficient | 0.55 |
| $\alpha_{pw}$ | PW exponent | 0.4 |
| $\alpha_{pw}$ | Max PW depth | 2 |
| | Move Grouping (MG) | `True` |
| | MG UCT constant | 12.0 |
| | MG detailed action classes | `True` |
| | MG final decision | `True` |
| | MG PW bias | `True` |
| | MG PW coefficient | 0.55 |
| | MG PW exponent | 0.4 |

Table 13: MCTS parameters for the baseline.

# J Set of training seeds

The training seeds used for all runs are: 1337, 7961, 4089.

# K Set of evaluation seeds

For the evaluation, a set of 100 randomly generated seeds between 0 and 9001 has been used.

2005, 5579, 4614, 3534, 2410, 5850, 5942, 6299, 4913, 3374, 1915,
5503, 8988, 1662, 432, 8051, 1246, 6407,5710, 7705, 5744, 1806,
4808, 2398, 6272, 1125, 310, 7352, 4628, 6086, 846, 3481, 8124,
1078, 118, 2017, 6829, 5608, 5550, 8619, 8887, 8063, 5530, 5517,
5240, 6898, 6097, 739, 1351, 5884, 8157, 6668, 7258, 8833, 6969,

1499, 7315, 775, 6801, 4091, 468, 475, 6290, 7100, 3328, 4484,
8618, 5698, 5920, 5762, 7542, 4347, 8677, 8040, 5191, 7558, 7924,
5753, 8613, 6194, 2475, 475, 6120, 1727, 1600, 8514, 6668, 8410,
5744, 1270, 308, 735, 3597, 8060, 4946, 3586, 615, 1646, 7605,
6723.

# L Iteration randomization

Iteration randomization is a training scheme where the number of MCTS iterations is randomized between a high value and a low value during training [88]. The goal is to generate data faster and produce more diverse situations during training. Figure 39 shows an evaluation using 400 as high and 80 as low value on scenario 08. Full searches with 400 iterations are performed 33% of the time. The iteration randomization model only slightly outperforms a Gaussian Mixture Model (GMM) with 3 components despite a 2.5× increase in wall clock training time. The GMM 3 was trained using 200 iterations.

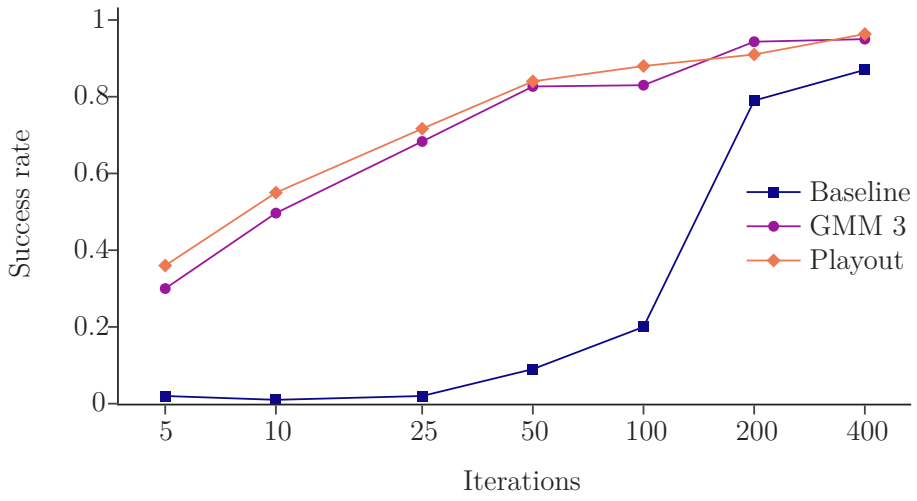



Figure 39: The iteration randomization model slightly outperforms the non-randomized GMM for low iterations. Both models outperform the baseline. Note that the $x$-axis is scaled logarithmically.

# References

[1] Achiam, J. Spinning up in deep reinforcement learning.

[2] Anthony, T., Tian, Z., and Barber, D. Thinking Fast and Slow with Deep Learning and Tree Search. *arXiv:1705.08439 [cs]* (Dec. 2017). arXiv: 1705.08439.

[3] Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer Normalization. *arXiv:1607.06450 [cs, stat]* (July 2016). arXiv: 1607.06450.

[4] Bacchiani, G., Molinari, D., and Patander, M. Microscopic Traffic Simulation by Cooperative Multi-agent Deep Reinforcement Learning. *arXiv:1903.01365 [cs]* (Mar. 2019). arXiv: 1903.01365.

[5] Bahdanau, D., Cho, K., and Bengio, Y. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473 [cs, stat]* (May 2016). arXiv: 1409.0473.

[6] Bellemare, M. G., Candido, S., Castro, P. S., Gong, J., Machado, M. C., Moitra, S., Ponda, S. S., and Wang, Z. Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature 588*, 7836 (Dec. 2020), 77–82.

[7] Bishop, C. M. Mixture density networks. Publisher: Aston University.

[8] Bishop, C. M. *Pattern recognition and machine learning*. Information science and statistics. Springer, New York, 2006.

[9] Bouton, M., Nakhaei, A., Isele, D., Fujimura, K., and Kochenderfer, M. J. Reinforcement Learning with Iterative Reasoning for Merging in Dense Traffic. *arXiv:2005.11895 [cs]* (May 2020). arXiv: 2005.11895.

[10] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI Gym. *arXiv:1606.01540 [cs]* (June 2016). arXiv: 1606.01540.

[11] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language Models are Few-Shot Learners. *arXiv:2005.14165 [cs]* (July 2020). arXiv: 2005.14165.

[12] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games 4*, 1 (Mar. 2012), 1–43.

[13] Chaslot, G. M. J. B., Winands, M. H. M., and van den Herik, H. J. Parallel Monte-Carlo Tree Search. In *Computers and Games*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., vol. 5131. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 60–71.

[14] Chen, J., Zhang, C., Luo, J., Xie, J., and Wan, Y. Driving Maneuvers Prediction Based Autonomous Driving Control by Deep Monte Carlo Tree Search. *IEEE Transactions on Vehicular Technology 69*, 7 (July 2020), 7146–7158. Conference Name: IEEE Transactions on Vehicular Technology.

[15] Chou, P.-W., Maturana, D., and Scherer, S. Improving Stochastic Policy Gradients in Continuous Control with Deep Reinforcement Learning using the Beta Distribution. 10.

[16] Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. Quantifying Generalization in Reinforcement Learning. 8.

[17] Couëtoux, A., Doghmen, H., and Teytaud, O. Improving the Exploration in Upper Confidence Trees. In *Learning and Intelligent Optimization*, Y. Hamadi and M. Schoenauer, Eds., vol. 7219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 366–371. Series Title: Lecture Notes in Computer Science.

[18] Couëtoux, A., Hoock, J.-B., Sokolovska, N., Teytaud, O., and Bonnard, N. Continuous Upper Confidence Trees. In *Learning and Intelligent Optimization*, C. A. C. Coello, Ed., vol. 6683. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 433–445. Series Title: Lecture Notes in Computer Science.

[19] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]* (May 2019). arXiv: 1810.04805.

[20] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE. 21.

[21] Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., and Hester, T. An empirical investigation of the challenges of real-world reinforcement learning. *arXiv:2003.11881 [cs]* (Mar. 2021). arXiv: 2003.11881.

[22] Everett, M., Chen, Y. F., and How, J. P. Collision Avoidance in Pedestrian-Rich Environments with Deep Reinforcement Learning. *arXiv:1910.11689 [cs]* (Apr. 2020). arXiv: 1910.11689.

[23] Foerster, J. N. Deep Multi-Agent Reinforcement Learning. 205.

[24] Francois-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., and Pineau, J. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning 11*, 3-4 (2018), 219–354. arXiv: 1811.12560.

[25] Giuliari, F., Hasan, I., Cristani, M., and Galasso, F. Transformer Networks for Trajectory Forecasting. *arXiv:2003.08111 [cs]* (May 2020). arXiv: 2003.08111.

[26] Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT Press, 2016.

[27] Guennebaud, G., Jacob, B., and others. Eigen v3, 2010.

[28] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]* (Aug. 2018). arXiv: 1801.01290.

[29] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., and Levine, S. Soft Actor-Critic Algorithms and Applications. *arXiv:1812.05905 [cs, stat]* (Dec. 2018). arXiv: 1812.05905.

[30] Hamrick, J. B., Bapst, V., Pfaff, T., Weber, T., Battaglia, P. W., Sanchez-Gonzalez, A., and Buesing, L. COMBINING Q-LEARNING AND SEARCH WITH AMORTIZED VALUE ESTIMATES. 27.

[31] He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]* (Dec. 2015). arXiv: 1512.03385.

[32] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. Deep Reinforcement Learning that Matters. *arXiv:1709.06560 [cs, stat]* (Jan. 2019). arXiv: 1709.06560.

[33] Hoel, C.-J., Driggs-Campbell, K., Wolff, K., Laine, L., and Kochenderfer, M. J. Combining Planning and Deep Reinforcement Learning in Tactical

Decision Making for Autonomous Driving. *IEEE Transactions on Intelligent Vehicles 5*, 2 (June 2020), 294–305. arXiv: 1905.02680.

[34] Hong, Z.-W., Pajarinen, J., and Peters, J. Model-based Lookahead Reinforcement Learning. *arXiv:1908.06012 [cs, stat]* (Aug. 2019). arXiv: 1908.06012.

[35] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., and Adam, H. Searching for MobileNetV3. *arXiv:1905.02244 [cs]* (Nov. 2019). arXiv: 1905.02244.

[36] Hu, J., Shen, L., Albanie, S., Sun, G., and Wu, E. Squeeze-and-Excitation Networks. *arXiv:1709.01507 [cs]* (May 2019). arXiv: 1709.01507.

[37] Huber, M. F., Bailey, T., Durrant-Whyte, H., and Hanebeck, U. D. On entropy approximation for Gaussian mixture random vectors. In *2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems* (Seoul, Aug. 2008), IEEE, pp. 181–188.

[38] Hubert, T., Schrittwieser, J., Antonoglou, I., Barekatain, M., Schmitt, S., and Silver, D. Learning and Planning in Complex Action Spaces. *arXiv:2104.06303 [cs]* (Apr. 2021). arXiv: 2104.06303.

[39] Huegle, M., Kalweit, G., Werling, M., and Boedecker, J. Dynamic Interaction-Aware Scene Understanding for Reinforcement Learning in Autonomous Driving. *arXiv:1909.13582 [cs, stat]* (Sept. 2019). arXiv: 1909.13582.

[40] Hügle, M., Kalweit, G., Mirchevska, B., Werling, M., and Boedecker, J. Dynamic Input for Deep Reinforcement Learning in Autonomous Driving. *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Nov. 2019), 7566–7573. arXiv: 1907.10994.

[41] Ioffe, S., and Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167 [cs]* (Mar. 2015). arXiv: 1502.03167.

[42] Kocsis, L., Szepesvari, C., and Willemson, J. Improved Monte-Carlo Search. 22.

[43] Kocsis, L., and Szepesvári, C. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 282–293. Series Title: Lecture Notes in Computer Science.

[44] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. *Communications of the ACM 60*, 6 (May 2017), 84–90.

[45] Kullback, S., and Leibler, R. A. On information and sufficiency. *The annals of mathematical statistics 22*, 1 (1951), 79–86. Publisher: JSTOR.

[46] Kurzer, K., Engelhorn, F., and Zöllner, J. M. Decentralized Cooperative Planning for Automated Vehicles with Continuous Monte Carlo Tree Search. *2018 21st International Conference on Intelligent Transportation Systems (ITSC)* (Nov. 2018), 452–459. arXiv: 1809.03200.

[47] Kurzer, K., Fechner, M., and Zöllner, J. M. Accelerating Cooperative Planning for Automated Vehicles with Learned Heuristics and Monte Carlo Tree Search. *arXiv:2002.00497 [cs, stat]* (May 2020). arXiv: 2002.00497.

[48] Kurzer, K., Hörtnagl, C., and Zöllner, J. M. Parallelization of Monte Carlo Tree Search in Continuous Domains. *arXiv:2003.13741 [cs, stat]* (Mar. 2020). arXiv: 2003.13741.

[49] Lan, L.-C., Li, W., Wei, T.-H., and Wu, I.-C. Multiple Policy Value Monte Carlo Tree Search. 7.

[50] Lanctot, M., Wittlinger, C., and Winands, M. H. M. Monte Carlo Tree Search for Simultaneous Move Games: A Case Study in the Game of Tron. 8.

[51] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation 1*, 4 (Dec. 1989), 541–551.

[52] Lenz, D., Kessler, T., and Knoll, A. Tactical cooperative planning for autonomous highway driving using Monte-Carlo Tree Search. In *2016 IEEE Intelligent Vehicles Symposium (IV)* (June 2016), pp. 447–453.

[53] Liao, J., Liu, T., Tang, X., Mu, X., Huang, B., and Cao, D. Decision-Making Strategy on Highway for Autonomous Vehicles Using Deep Reinforcement Learning. *IEEE Access 8* (2020), 177804–177814. Conference Name: IEEE Access.

[54] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]* (July 2019). arXiv: 1509.02971.

[55] Liu, A., Chen, J., Yu, M., Zhai, Y., Zhou, X., and Liu, J. WATCH THE UNOBSERVED: A SIMPLE APPROACH TO PARALLELIZING MONTE CARLO TREE SEARCH. 21.

[56] Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., and Mordatch, I. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *arXiv:1706.02275 [cs]* (Mar. 2020). arXiv: 1706.02275.

[57] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (Feb. 2015), 529–533.

[58] Moerland, T. M., Broekens, J., Plaat, A., and Jonker, C. M. A0C: Alpha Zero in Continuous Action Space. *arXiv:1805.09613 [cs, stat]* (May 2018). arXiv: 1805.09613.

[59] Moerland, T. M., Deichler, A., Baldi, S., Broekens, J., and Jonker, C. M. Think Too Fast Nor Too Slow: The Computational Trade-off Between Planning And Reinforcement Learning. *arXiv:2005.07404 [cs]* (May 2020). arXiv: 2005.07404.

[60] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A Distributed Framework for Emerging AI Applications. *arXiv:1712.05889 [cs, stat]* (Sept. 2018). arXiv: 1712.05889.

[61] NVIDIA. NVIDIA TensorRT, Apr. 2016.

[62] Oliehoek, F. A., and Amato, C. *A Concise Introduction to Decentralized POMDPs*. SpringerBriefs in Intelligent Systems. Springer International Publishing, Cham, 2016.

[63] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. 12.

[64] Petosa, N., and Balch, T. Multiplayer AlphaZero. *arXiv:1910.13012 [cs]* (Dec. 2019). arXiv: 1910.13012.

[65] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. ROS: an open-source Robot Operating System. 6.

[66] Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., and Dormann, N. Stable baselines3, 2019.

[67] Rolnick, D., Donti, P. L., Kaack, L. H., Kochanski, K., Lacoste, A., Sankaran, K., Ross, A. S., Milojevic-Dupont, N., Jaques, N., Waldman-Brown, A., Luccioni, A., Maharaj, T., Sherwin, E. D., Mukkavilli, S. K., Kording, K. P., Gomes, C., Ng, A. Y., Hassabis, D., Platt, J. C., Creutzig, F., Chayes, J., and Bengio, Y. Tackling Climate Change with Machine Learning. *arXiv:1906.05433 [cs, stat]* (Nov. 2019). arXiv: 1906.05433.

[68] Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review 65*, 6 (1958), 386–408.

[69] Rusu, A. A., Colmenarejo, S. G., Gulcehre, C., Desjardins, G., Kirkpatrick, J., Pascanu, R., Mnih, V., Kavukcuoglu, K., and Hadsell, R. Policy Distillation. *arXiv:1511.06295 [cs]* (Jan. 2016). arXiv: 1511.06295.

[70] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *arXiv:1911.08265 [cs, stat]* (Feb. 2020). arXiv: 1911.08265.

[71] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. Trust Region Policy Optimization. *arXiv:1502.05477 [cs]* (Apr. 2017). arXiv: 1502.05477.

[72] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]* (Aug. 2017). arXiv: 1707.06347.

[73] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of Go with deep neural networks and tree search. *Nature 529*, 7587 (Jan. 2016), 484–489.

[74] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science 362*, 6419 (Dec. 2018), 1140–1144.

[75] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of Go without human knowledge. *Nature 550*, 7676 (Oct. 2017), 354–359.

[76] Silver, D., Singh, S., Precup, D., and Sutton, R. S. Reward is enough. *Artificial Intelligence 299* (Oct. 2021), 103535.

[77] Sutton, R. S., and Barto, A. G. *Reinforcement learning: an introduction*, second edition ed. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, 2018.

[78] Tak, M. J. W., Lanctot, M., and Winands, M. H. M. Monte Carlo Tree Search variants for simultaneous move games. In *2014 IEEE Conference on Computational Intelligence and Games* (Dortmund, Germany, Aug. 2014), IEEE, pp. 1–8.

[79] Tay, Y., Dehghani, M., Abnar, S., Shen, Y., Bahri, D., Pham, P., Rao, J., Yang, L., Ruder, S., and Metzler, D. Long Range Arena: A Benchmark for Efficient Transformers. *arXiv:2011.04006 [cs]* (Nov. 2020). arXiv: 2011.04006.

[80] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need. *arXiv:1706.03762 [cs]* (Dec. 2017). arXiv: 1706.03762.

[81] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature 575*, 7782 (Nov. 2019), 350–354.

[82] Wang, H., Emmerich, M., Preuss, M., and Plaat, A. Alternative Loss Functions in AlphaZero-like Self-play. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)* (Xiamen, China, Dec. 2019), IEEE, pp. 155–162.

[83] Wang, Q., Wu, B., Zhu, P., Li, P., Zuo, W., and Hu, Q. ECA-Net: Efficient Channel Attention for Deep Convolutional Neural Networks. *arXiv:1910.03151 [cs]* (Apr. 2020). arXiv: 1910.03151.

[84] Weisberg, S. Applied Linear Regression. 370.

[85] Willemsen, D., Baier, H., and Kaisers, M. Value targets in off-policy AlphaZero: a new greedy backup. 9.

[86] Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. 28.

[87] Wright, M. A., and Horowitz, R. Neural-Attentional Architectures for Deep Multi-Agent Reinforcement Learning in Varying Environments. 8.

[88] Wu, D. J. Accelerating Self-Play Learning in Go. *arXiv:1902.10565 [cs, stat]* (Feb. 2020). arXiv: 1902.10565.

[89] Yang, X., Duvaud, W., and Wei, P. Continuous Control for Searching and Planning with a Learned Model. *arXiv:2006.07430 [cs]* (June 2020). arXiv: 2006.07430.

[90] Yarats, D., and Kostrikov, I. Soft actor-critic (SAC) implementation in PyTorch, 2020.

[91] Yurtsever, E., Capito, L., Redmill, K., and Ozgune, U. Integrating Deep Reinforcement Learning with Model-based Path Planners for Automated Driving. In *2020 IEEE Intelligent Vehicles Symposium (IV)* (Oct. 2020), pp. 1311–1316. ISSN: 2642-7214.

# Assertion

*Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.*

Karlsruhe, August 14, 2021 Timo Klein