Closing the Modelling Gap: Transfer Learning from a Low-Fidelity Simulator for Autonomous Driving

by

Aravind Balakrishnan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The behaviour planning subsystem, which is responsible for high-level decision making and planning, is an important aspect of an autonomous driving system. There are advantages to using a learned behaviour planning system instead of traditional rule-based approaches. However, high quality labelled data for training behaviour planning models is hard to acquire. Thus, reinforcement learning (RL), which can learn a policy from simulations, is a viable option for this problem. However, modelling inaccuracies between the simulator and the target environment, called the *'transfer gap'*, hinders its deployment in a real autonomous vehicle. High-fidelity simulators, which have a smaller transfer gap, come with large computational costs that are not favourable for RL training. Therefore, we often have to settle for a fast, but lower fidelity simulator that exacerbates the transfer learning problem.

In this thesis, we study how a low-fidelity 2D simulator can be used in place of a slower 3D simulator for training RL behaviour planning models, and analyze the resulting policies in comparison with a rule-based approach. We develop WiseMove, an RL framework for autonomous driving research that supports hierarchical RL, to serve as the low-fidelity source simulator. A transfer learning scenario is set up from WiseMove to an Unreal[1]-based simulator for the Autonomoose[2] system to study and close the transfer gap.

We find that perception errors in the target simulator contribute the most to the transfer gap. These errors, when naively modelled in WiseMove, provide a policy that performs better in the target simulator than a carefully constructed rule-based policy. Applying domain randomization on the environment yields an even better policy. The final RL policy reduces the failures due to perception errors from 10% to 2.75%. We also observe that the RL policy has less reliance on the velocity compared to the rule-based algorithm, as its measurement is unreliable in the target simulator. To understand the exact learned behaviour, we also distill the RL policy using a decision tree to obtain an interpretable rule-based policy. We show that constructing a rule-based policy manually to efficiently handle perception errors is not trivial. Future work can explore more driving scenarios under fewer constraints to further validate this result.

[1] Unreal Engine 4 by Epic Games

[2] Autonomous driving research platform at the University of Waterloo www.autonomoose.net

Acknowledgements

I would like to thank my supervisor, Prof. Krzysztof Czarnecki, for providing me with the opportunity and for his continued support throughout the program.

I wish to thank Sean Sedwards for his support and feedback throughout the course of this endeavour. I would also like to thank Jaeyoung Lee, Ashish Gaurav, Marko Ilievski and my colleagues at Waterloo Intelligent System Engineering Lab (WISELab) for their support and feedback.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Research on Autonomous Driving System(ADS) has gained a lot of momentum in recent years, with companies such as Waymo, BMW and Tesla racing to capitalize on the commercial autonomous vehicles (AV) market. This accelerated research has urged policymakers to update legislation [4] to accommodate autonomous vehicles alongside human drivers. The impact of such a technology becoming ubiquitous is expected to have far-reaching effects and holds the potential to greatly improve road safety, optimize transport efficiency, increase economic productivity and improve environmental conditions [34]. Autonomous vehicles have existed as far back as the 1980s, with Dickmann's autonomous van VaMoRs [15] and Carnegie Mellon University's NavLab [60] demonstrating sensor-based autonomous driving. But driving on busy public roads as well as a human driver remains a difficult engineering challenge. Despite this, significant progress has been made in that direction in the past few years due to billions of dollars being invested in this field.

Current ADS technology is made possible by complex components working in unison, each specializing in solving tasks such as localization, routing, mapping, perception, motion planning and vehicle control. An important aspect of ADS is the motion planning system. Motion planning encompasses the whole process of finding an optimal and safe route from A to B, by identifying and maneuvering around static and dynamic obstacles following the rules of the road. It is typically decomposed hierarchically into route planning, behaviour planning and local motion planning. Behaviour planning is responsible for making high-level decisions or *maneuvers* required to navigate the local environment of the vehicle. The behaviour planner needs to take into consideration not only the obstacles in the local environment but also the traffic rules, road features and the AV's state to make the optimal and, most importantly, safe decision.

A behaviour planner where the decision making relies on a set of explicitly programmed rules [62, 67] needs to account for all of these factors, as well as uncertainty and rare edge cases. This becomes very difficult to systematically develop without gaps and also to scale and test. Therefore, the prospect of learning this system is very appealing, especially with the recent advances in deep learning research and its success in solving large, complex problems. However, training a successful model requires a large amount of data. For example, Waymo's ChauffeurNet[8] uses a dataset containing 60 days of real-world driving data. Although there is no dearth of public datasets for autonomous driving [5, 17, 65], they do not contain ground truth labels for learning behaviour planning maneuvers. Creating your own dataset for this means not just collecting relevant data but also performing the laborious task of ensuring that it is properly labelled, consistent and error-free.

Reinforcement learning(RL) is useful in this case as it does not require labelled data, but rather, learns by exploring and interacting with the environment, guided only by some form of rewards. RL has been receiving a lot of attention in recent years and gained momentum after an RL agent achieved performance rivalling professional human testers in a set of classic Atari 2600 video games, using only the pixels and game score as input [35]. RL has subsequently made headlines for being the first algorithm to beat a professional player in a full game of Go without handicaps [57], and also the first to beat a team of professional players in the competitive e-sports game Dota 2 [40]. Both games have long been considered to be difficult challenges in the field of AI, due to the complexity and strategy involved.

The successes of RL have encouraged researchers to look at it as a potential candidate for solving complex real-world problems such as planning for autonomous driving. This is usually done in a simulated environment as it is not feasible and often dangerous to let the agent collect the required experiences in the real world. Indeed, RL has been applied to end-to-end continuous control of AVs where environment features are directly translated to vehicle control [55, 33]. However, from a design perspective, it is advantageous to decompose a large problem into functional modules for increasing scalability and specialization. The hierarchical decomposition of the motion planning problem stated previously is based on temporal abstraction. Temporal abstraction applied to RL gives rise to Hierarchical RL (HRL), where a policy can explore the solution space by choosing sub-policies, rather than primitive actions. Applying HRL to the motion planning problem is, thus, quite intuitive and there is a large body of existing work that explores this [56, 43, 13].

A vast majority of RL solutions, however, remains in the realm of simulated environments, and have not been successfully deployed in real AVs. Transferring the learned experiences in simulation to the actual task is still an unsolved problem. Since a simulation does not perfectly model the real task, RL agents have trouble generalizing and

overcoming this *transfer gap*. Solving this problem of transfer learning, or in this case called *sim-to-real*, is crucial for successfully applying RL to real world problems. A large body of research exists for sim-to-real transfer of robotic control policies [41, 45, 11, 18], including end-to-end motion planning [64]. RL agents for such fine robotic control tasks are sensitive to even slight deviations from the training environment [41]. However, since behaviour planning operates at a coarser temporal resolution, the transfer learning problem may not be as severe. Sim-to-real for motion planning in a hierarchical setting has so far been limited to drones and smaller robots [32, 53, 38], along with a few non-RL approaches for AVs [37].

An intuitive approach to solve this problem is to close the transfer gap as much as possible through accurate modelling. But this process is quite expensive, requiring careful calibration and large amounts of real-world data. It also becomes computationally expensive for simulating complex physical models. This is especially detrimental for training an RL agent as it learns by exploration in the simulation. Therefore, due to time and cost constraints, only a lower fidelity simulator may be available for RL training even though it furthers the transfer gap.

In this thesis, we study this problem in the context of behaviour planning. We consider a *sim-to-sim* transfer of an HRL policy for behaviour planning from a low-fidelity simulator to a target simulator, which is assumed to be as close to reality as possible. The high-fidelity simulator is based on the Unreal Engine [2] and is used in the pre-deployment testing of the software stack for the autonomous research platform, Autonomooose [1]. The objective is to study and close the transfer gap between the two simulators to enable efficient training of behaviour planning policies for Autonomooose, and compare its performance with a traditional rule-based approach as a reference.

Our main contributions in this thesis are as follows:

- We develop WiseMove, a Python framework for autonomous driving research that supports HRL, to serve as the low-fidelity simulator for training behaviour planning policies. The general framework [6] briefly described in Section 3.2.1 is a joint effort by me, Ashish Gaurav, Jaeyoung Lee and Sean Sedwards from the WISE Lab research group at University of Waterloo. All modifications to this base framework for conducting the experiments described in this thesis are solely my contribution.

- We align the two simulators such that a transferred policy is affected only by the differences in simulator modelling. Thus, we find the individual factors that cause the transfer gap and analyze the contribution of each. We also show that even roughly modelling some of the differences in WiseMove results in an RL policy that outperforms a rule-based policy.

- We further improve the performance of the RL policy by using a popular transfer learning technique, *domain randomization*.
- We analyze the importance of each feature in the final policy to learn how it differs from the rule-based policy. We also distill the RL policy into a rule-based algorithm to get an insight into why it performs better.

The remainder of this thesis proceeds as follows. Chapter 2 introduces the autonomous driving stack and presents the necessary background information upon which this work is built. Chapter 3 outlines the overall methodology and discusses the WiseMove framework, how the simulators are aligned, how the RL policies are trained and how it is improved and analyzed. We also discuss the assumptions that we make. Chapter 4 describes the experimental setup, the results obtained from our experiments and a brief discussion of the results. Chapter 5 concludes the thesis with relevant discussion and identifies future work on this research.

Chapter 2

Background

In this chapter, we provide the background necessary for understanding the later chapters. We begin by introducing the components of the autonomous software stack of Autonomooose and move on to briefly discussing the reinforcement learning algorithm used in this thesis. Then, we outline a few techniques used in sim-to-real transfer learning, one of which we successfully use in our study.

2.1 Autonomous Driving Stack

The autonomous driving stack refers to the software architecture of the ADS. Essentially, the stack is an implementation of a cycle of information gathering, decision making and vehicle control. With decades of research on autonomous driving and robotics, a consensus has emerged on how a typical ADS stack is designed [30, 24, 62]. At a high level, it should be able to perceive its surroundings to make an informed decision and maneuver the vehicle safely to its objective. In the following sections, we introduce the components of a typical autonomous driving stack. The Autonomooose stack, which is based on the Robot Operating System (ROS) [51], follows a similar design and is used by the high-fidelity simulator. Therefore, we also discuss the details of its subsystems relevant to this study.

2.1.1 Perception and Tracking

To make quick informed decisions, a human driver on public roads is constantly taking in information about their immediate driving environment, from static features like lane markings and traffic signs to dynamic objects such as other vehicles and pedestrians. The ability to gain knowledge about the local environment is, thus, fundamental to the safe operation of an AV and is made possible by its perception subsystem. Perception is the first stage in the stack's pipeline. It processes information from cameras and other sensors to identify objects in the environment that are relevant to motion planning. AVs typically use cameras, radars and lidars to collect data about the current state of the environment. In the `perception` subsystem of Autonomooose, both camera and lidar input are used by an Aggregate View Object Detection (AVOD) network [27] to create 3D bounding boxes for dynamic objects. Static objects are detected using lidar point cloud data and the output is a probabilistic 2D occupancy grid.

Perception is coupled with a tracking subsystem to provide historical tracks for moving objects. Motion planning also requires predicted tracks to make proactive decisions. The `tracker` in Autonomooose processes the dynamic object bounding boxes from perception and keeps track of pseudo-continuous temporal changes to each object relative to the moving frame of the *ego*[1] vehicle. It has to account for perception errors and occlusion to provide robust and accurate tracks. The noisy measurements are processed using a Kalman filter to get better estimates for position and velocity.

2.1.2 Mapping

Since static elements in the environment such as road markings and traffic signs, by definition, can be assumed to not change much over time, we can perform the perception step beforehand to reduce computation. In addition, this data can be maintained and improved by human engineers to ensure quality. Mapping for AVs refers to the aggregation of these curated static features into a form that can be used in the stack. Mapping inherently requires localization as well, which identifies and keep track of where the vehicle is with respect to its surroundings. This is achieved by comparing sensor data about the local environment with the map features. The mapping subsystem in Autonomooose platform uses the *lanelet* map format described in the following section.

Figure 2.1: Road map constructed using lanelets (in light blue).

Lanelet Map

The lanelet map specification provides an efficient and scalable way to define the static environment in a parsable format. It was developed initially by Bender et al. [9] and updated by Poggenhans et al [47]. The format represents the drivable environment in both geometric and topological aspects. It is easily scalable to accommodate extensions and improvements. The lanelet format is quite intuitive to use as well leading to its adoption in the autonomous driving domain [66].

The lanelet specification was inspired by the Open Street Map (OSM) [42], a free open-source editable map of the world. Therefore, it is also stored as an XML[2] file following the OSM data format. The basic element is a *node*, which is a point in the world stored as latitude-longitude coordinates. A node can be used to represent any map feature by appending other properties to it. An example of a basic node is given below:

```
<node id='-23'lat='43.51125968873' lon='-80.53082081321'></node>
```

[1] autonomous vehicle that the stack operates
[2] eXtensible Markup Language designed to store and transport data

A polyline is represented as an ordered list of nodes, called a *way*. It can form a polygon as well if they form a closed loop. These are used for representing linear features such as roads, parking areas and buildings. Like a node, a way can be extended using additional properties if necessary. A basic way is given below:

```
<way id='-62661' action='modify' visible='true'>
    <nd ref='-695' />
    <nd ref='-697' />
    <nd ref='-699' />
</way>
```

Thus, a minimal *lanelet* is constructed with two ways labelled 'left' and 'right'. It can represent various road elements such as a simple driving lane with its left and right boundaries and can be easily extended to define complex elements as there are few restrictions on how a lanelet can be used. Additional properties can be applied to lanelets as well, and this defines its semantic identity. Thus, a new lanelet needs to be defined if the property changes.

Using these basic building blocks, a road map for any area can be constructed in the lanelet format. Figure 2.1 shows a lanelet road map constructed for a real world location.

2.1.3 Motion Planning

The autonomous vehicle motion planning problem can be defined as producing a sequence of control actions, in the continuous domain, to move a vehicle from its current pose to a destination while adhering to specified requirements of safety, comfort, progress and energy efficiency. The actions should also be valid; that is, it can be safely executed by the vehicle hardware taking into account its physical constraints. The decision making and control in a typical stack often follows the hierarchical decomposition of the motion planning problem, which is (*i*) mission planning; (*ii*) behaviour planning; and (*iii*) local planning. This decomposition of the problem is often reflected in many ADS architectures, although the boundaries are rather fuzzy, with varying design choices being found in the literature.

Mission Planning

Mission planning is at the highest level of decision making for an AV and it involves deciding the sequence of roads to take for the proposed journey, given the desired destination, the current location, the user's preference, road conditions and traffic. If the mission planner's assumptions change along the way, or the user makes changes to the requirements, it should be able to re-plan dynamically. Mission planning is considered a solved problem and it is standard on current GPS navigation devices. In Autonomooose, the route is calculated from the lanelet map in the mapping subsystem.

Behaviour Planning

The next component in the planning hierarchy, the behaviour planner, generates control actions at an abstracted or relatively high level to navigate safely through the route provided by the mission planner [39]. The output is then interpreted further down the pipeline to finely control the vehicle. These control actions might be basic maneuvers such as *change lane*, *come to a stop* and *maintain speed* [43], or high-level waypoints made up of future poses [8]. However, the output is design-specific and is constrained by the subsystems that consume it (such as local planning).

The behaviour planner must react to both the static and the dynamic nature of the environment and therefore, mapping and perception are crucial inputs. The `behaviour planner` in Autonomooose transforms all its input to the `base_link` coordinate frame[3] before processing it. In the REP 105 specification for ROS, `base_link` is one of the main coordinate frames in robotic applications. It is a body-fixed frame that serves as the primary reference for the body of the robot. In the Autonomooose platform, `base_link` is located at the centre of the vehicle's rear axle. The coordinate axes are as follows:

- X-axis is positive forwards, negative backwards
- Y-axis is positive left, negative right
- Z-axis is positive up, negative down

The `behaviour planner` constructs an abstracted representation of the environment for decision making. The input from mapping and `tracker` are processed to extract only the relevant details in the form of abstract predicates. A rule processing system, the `rule`

[3] a frame of reference for any spatial measurement or data in robotics

`engine`, then processes the predicates through a set of rules to generate an appropriate basic maneuver depending on the rules that the predicates satisfy. The `behaviour planner` then adds additional information and constraints to produce a high-level maneuver for the local planner to execute. Some examples of the maneuvers used here are `trackspeed` - tracks a target speed along the route, `decelerate-to-stop` - gradually stop at an upcoming stop location and `yield` - yield to oncoming traffic when the ego vehicle does not have right of way.

Local Planning

The behaviour planner's high-level directive is fulfilled by the local planner. The objective of local planning is to devise a safe and smooth trajectory from the autonomous vehicle's current pose to a target pose given or implied by the behaviour planner's output. It is the responsibility of the local planner to avoid obstacles, satisfy comfort requirements and respect kinodynamic constraints.

2.2 Unreal Simulator

For testing the Autonomooose stack before deployment in the real world, a simulator built on top of the Unreal Engine 4 [2] is used. Unreal is a popular video game engine, produced by Epic Games and developed in C++, that provides a rich set of well-documented features for creating complex 3D environments. The Unreal simulator interfaces with the Autonomooose stack to apply the generated control actions on a high-fidelity vehicle dynamics model, modelled from real vehicle data [63, 21]. The resulting response inside the simulated environment is then fed back to the stack ecosystem. The simulator also replicates perception by simulating the lidar sensor in Autonomooose by using ray tracing to create a point cloud. Thus, noise and errors from perception are present in this simulator.

GeoScenario

A test scenario is specified using the GeoScenario format [50], which is the standard OSM format extended with custom properties for scenario definition. Therefore, it is also constructed using nodes and ways. Each node in a Geoscenario represents the various scenario objects, such as ego, dynamic objects and static objects. A GeoScenario specification can

be parsed only in the context of a lanelet map, and thus, it is always associated with a lanelet OSM file that defines the road network. A basic scenario contains ego's start position and a series of goal points as nodes. The scenario is successful when ego reaches each goal point in order.

More complex scenarios can be created using dynamic objects and trigger nodes. Dynamic objects can be spawned in the scenario using a node as the start point and associating it with a way element, which provides it with a fixed trajectory to follow. The target speed and acceleration can be included in each way node to provide a speed and acceleration profile for the object, which the simulator will attempt to match within the bounds of the object dynamics. The dynamic objects are always spawned at the start of the scenario. To control their behaviour, triggers can be placed in the scenario to associate one or more actions with an event. For example, 10 seconds after the start of the scenario, a dynamic object will start moving from its spawn point and follow a preset trajectory.

2.3 Reinforcement Learning

2.3.1 Markov Decision Process

The Markov Decision Process (MDP) provides a mathematical framework for decision making in an environment under the control of a decision-maker or *agent*. An MDP can be defined as the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions that can be taken, transition probability $\mathcal{T}(s, a, s') = P(s' \mid s, a) \in [0, 1]$ is the probability of transitioning from the state s to s' by taking action a, reward function $\mathcal{R}(s, a, s')$ gives reward r incurred from transitioning from state s to state s' by taking action a, and $\gamma \in [0, 1]$ is the discount factor.

A policy $\pi : \mathcal{S} \to \mathcal{A}$ specifies an action that the agent will choose in state s. The optimum policy $\pi^*(s)$ will maximize the expected discounted cumulative reward:

$$\mathbb{E}_{\pi^*} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right],$$

where $r_t = \mathcal{R}(s_t, a_t, s_{t+1})$ after taking action $a_t = \pi^*(s_t)$ in state s_t which transitions to state s_{t+1}.

To quantify how well a policy performs, we define a *value* function $V_\pi(s)$ and an *action-value* function $Q_\pi(s, a)$. The value function $V_\pi(s)$ is the expected cumulative reward start-

ing from state s following policy π.

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \mid s_{t=0} = s \tag{2.1}$$

The action-value function, which gives the quality of a state-action pair (and thus called *Q-function*) is the expected cumulative reward starting from state s with action a and following policy π then onwards.

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \mid s_{t=0} = s, a_{t=0} = a \tag{2.2}$$

Among all the possible value functions and Q-functions, there exists an optimal one of each, denoted by $V^*(s)$ and $Q^*(s,a)$ respectively, corresponding to the optimal policy $\pi^*(s)$.

2.3.2 Q-Learning

There exist dynamic programming algorithms [58, Chap. 4], such as *value iteration* and *policy iteration*, that finds the optimal policy assuming complete knowledge of the environment in the form of the transition probabilities $\mathcal{T}$. The model can also be learned before applying these algorithms to satisfy this assumption. This class of algorithms are called *model-based* and have been shown to be quite successful in some domains [48]. However, it may be non-trivial to learn an accurate environment model and also, the agent may only be as good as the environment model allows.

Model-free algorithms do not make this assumption and learn the optimal policy directly by interacting with the environment. Q-learning [58, Chap. 6] is a model-free algorithm that has earned considerable success. Q-learning is extended from the value iteration algorithm, which converges to $V^*(s)$ by iteratively improving an estimate of the value function. The estimate is updated according to the *Bellman Equation*[58, Chap. 3], where $\hat{V}(s)$ is the current estimate:

$$\hat{V}(s) = \max_a \sum_{s' \in \mathcal{S}} P(s'|s,a)[\mathcal{R}(s,a,s') + \gamma \hat{V}(s')], \forall s \in \mathcal{S}$$

In Q-learning, the Q-values are initialized to an arbitrary estimate (usually zero) and the agent interacts with the environment to generate samples. The Q-function estimate,

denoted by $\hat{Q}(s,a)$, is updated by using the sampled *Bellman error*, also known as the *temporal difference error (TD error)*, denoted by $\delta(s,a)$:

$$\delta(s,a) = \mathcal{R}(s,a,s') + \gamma \max_a \hat{Q}(s',a) - \hat{Q}(s,a) \tag{2.3}$$

Q-learning iteratively computes the Q-values using the following update rule such that the TD error is minimized:

$$\hat{Q}(s,a) := \hat{Q}(s,a) + \alpha\delta(s,a) \tag{2.4}$$

where α is the learning rate.

2.3.3 Deep Q-Learning

The iterative algorithm using Equation 2.4 can be applied successfully only in the tabular[4] case and is not tractable for problems with large state spaces. By using function approximation, with the Q-function being parameterized by θ, we can apply the algorithm to problems with larger and even continuous state space. However, the update rule in Equation 2.4 cannot be applied directly. The objective is still to reduce the TD error each update and therefore, we can apply gradient descent in a direction that minimizes $\delta(s,a)$. Therefore, we define the loss function as follows:

$$\begin{aligned} L(\theta) &= \nabla_\theta \left[\frac{1}{2}\delta^2\right] \\ &= \delta\nabla_\theta \left[\mathcal{R}(s,a,s') + \gamma \max_a \hat{Q}_\theta(s',a) - \hat{Q}_\theta(s,a)\right] \\ &= -\delta\nabla_\theta \hat{Q}_\theta(s,a) \end{aligned} \tag{2.5}$$

Thus, the update equation for θ becomes:

$$\theta := \theta + \alpha\nabla_\theta \hat{Q}_\theta(s,a) \tag{2.6}$$

A Q-learning algorithm where the function approximator is a deep neural network is called Deep Q-Learning, with the neural network that representing the Q-Network called the Deep Q-Network (DQN) [36]. However, basic Q-learning using any function approximator is observed to be unstable during training, and therefore, Mnih et al. [36] introduced two techniques to successfully apply this algorithm in practice.

[4] problems where the state space is small enough to be represented as tables

The first problem arises from the moving *Q-target* which is the first component $\mathcal{R}(s, a, s')$ $+ \gamma \max_a \hat{Q}(s', a)$ in Equation 2.3. When the weights are updated, the same weights apply to both the target and the predicted value $\hat{Q}(s, a)$. That means as we move closer to the target, the target also moves leading to oscillations in the training process. To alleviate this problem, a second *target network* $\tilde{Q}(s, a)$, which is essentially a copy of the original Q-network, is introduced. The target network is synchronized with the original Q-network less frequently so that it serves as a pseudo-fixed target for calculating the TD error.

The second problem is that the samples used for the TD learning are temporally correlated since the next state s' depends on the current state and action. This can cause the Q-values to oscillate or diverge over the course of training. To break this correlation, an *experience replay* buffer stores the sampled state transition tuple (s, a, s', r) and samples batches of experiences randomly from the buffer for training. This also improves the sample efficiency as it allows for efficient use of past experiences, by learning with it multiple times.

2.3.4 Double Dueling DQN

There have been many improvements to the vanilla DQN over the years. In this study, we make use of two modifications that help the algorithm converge faster. These improvements are complementary and can, thus, be combined into a single Deep RL architecture.

The first improvement, known as Double DQN, was introduced by Hado et al [19]. The motivation comes from the fact that DQN tends to overestimate the Q-values since the Q-target is based on the action that has the highest value at state s'. Therefore, double DQN decouples the action selection from the Q-target generation by selecting the best action using the target network and then generating the Q-value using the primary Q-network. Equation 2.3 becomes:

$$\delta(s, a) = \mathcal{R}(s, a, s') + \gamma \hat{Q}(s', \arg\max_a \tilde{Q}(s', a)) - \hat{Q}(s, a) \tag{2.7}$$

Dueling DQN (DDQN) is the second modification which decomposes the Q-function into the value function $V(s)$, which is the value of being in a state, and an advantage function $A(s, a)$, which represents how much better or worse it is to take action a over the other actions. This is an architectural change that requires separate estimators for both elements and a final aggregation layer to obtain the Q-value. The decoupling allows the agent to know the value of a state without learning the effect of all actions and also to better differentiate the value of actions in states where the Q-values are similar.

2.3.5 Hierarchical Reinforcement Learning

The basic RL algorithm treats the state space as a large search space, which can result in paths from the start to goal states becoming very long. Such long horizons can weaken the signal from future rewards leading to inefficient learning. Hierarchical RL (HRL) attempts to solve this issue by decomposing the problem into smaller pieces and learning to operate over multiple temporal abstractions. A policy in HRL can choose to execute sub-policies, which are temporally abstracted actions with their own sub-goals. Thus, the long path to the goal can be thought of as achieving a sequence of sub-goals. This also allows modularization of RL solutions, which in turn increases scalability, and also allows the reuse of sub-policies for related tasks.

Accommodating HRL into the MDP framework is achieved by defining a semi-MDP (SMDP) [59], in which the state transition function becomes $\mathcal{T}(s, a, s') = P(s', \tau \mid s, a)$, to consider the small amount of time τ passed between decision instants. The Options framework is the most well-known formulation for HRL [59, 7]. It extends the action space to include abstract macro-actions called options $m \in \mathcal{M}$, which are temporally extended sequences of primitive actions $a \in \mathcal{A}$. An option $m = (I_m, \pi_m, \beta_m)$ where $I_m \subseteq \mathcal{S}$ is the initiation set, $\pi_m(s) : \mathcal{S} \to \mathcal{A}$ is the policy for the option and $\beta_m(s) : \mathcal{S} \to [0, 1]$ is the termination condition. Options are considered as actions at a higher level of temporal abstraction.

Consider an Options framework with two levels of temporal abstraction. The top-level or *high-level* policy $\Pi(s) : \mathcal{S} \to \mathcal{M}$ is a policy over options that uses the environment observation to output an option m to execute. An option is available in state s only if $s \in I_m$. When an option is selected, primitive actions are taken according to *low-level* policy $\pi_m(s)$ until it terminates with probability $\beta_m(s)$. An algorithm operating in the Options framework is proven to converge to an optimal policy [49].

2.4 Transfer Learning

Since RL uses random exploration for collecting training data, its training is impractical for many applications in physical systems without employing a simulator. However, discrepancies between the simulated model and the real system, referred to as the *transfer gap* or *reality gap*, form a challenging barrier to deploying an RL policy learned using simulated data. This problem in RL is generally known as transfer learning and arises when a policy is learned in a *source* environment, but its intended application is in a *target* environment

with a slightly different data distribution. We can categorize the approaches to tackle the transfer learning problem into three broad types.

2.4.1 System Identification

System Identification [25] uses statistical methods to build models of a physical system using measurements of its response to external influences (input). A common approach is to assume the system as a black box and not be concerned about the details of what is happening inside the system. This process is the most intuitive step used to reduce the transfer gap as it aims to make the simulator as close to the target as possible by careful calibration and tuning.

Unfortunately, this is an expensive procedure for physical systems and is time-consuming. Even if this process is performed very well, most simulators are unable to capture the nuances of real world physics and replicate the richness and noise present in high-fidelity sensors. Many physical parameters of the same system can also vary after calibration due to changes in factors such as temperature, pressure, and even normal wear-and-tear. Another challenge is that high-fidelity models are often computationally expensive.

2.4.2 Domain Adaptation

Domain adaptation (DA) techniques, on the other hand, attempt to align the knowledge gained from the source simulator such that the agent can perform well in the target environment. The main assumption here is that there exists some common characteristics such that behaviour or knowledge representations learned in the source will be useful in the target. Thus, they fall into one of two categories: those which make use of structural similarities between the source and target domains, and those exploiting semantic similarities with prior knowledge of both domains [26].

One common approach is to align the feature space to a common representation or learn invariant features [18], in a supervised or unsupervised fashion. Recent DA techniques, especially for image-based feature spaces, use adversarial loss with respect to a domain discriminator to either align the representation or the behaviour [11]. Progressive networks [54] have also been used to significantly reduce the amount of data required from the target environment by learning on top of the behaviour learned in simulation. However, all these methods have a reliance on data from the target environment which may be costly to collect.

2.4.3 Domain Randomization

Unlike the previous approaches, *domain randomization* (DR) does not need data, labelled or otherwise, from the target environment. In DR, the RL agent is exposed to variances in the environment during training by randomizing the simulator properties (parameters). The "discrepancies between the source and target domains are modelled as variability in the source domain" [45]. Thus, the policy is trained to maximize the expected cumulative reward over a distribution of environment instances. DR has been used successfully to model feature space differences such as lighting conditions and camera placement in image-based tasks [61], as well as to develop policies that are robust to uncertainty in the target environment dynamics [41].

In Uniform DR, which we use in this study, each randomization parameter is sampled from a fixed range, which is selected either using domain knowledge or through trial-and-error. The parameters can vary any aspect of the simulator, ranging from its physical dynamics and behaviour to the level of observation noise. As an example, DR for a fine robotic control task can include parameters such as mass, material and dimensions of the robot and other objects, surface friction and damping of joints, gains for the PID controller and action delay [41]. In contrast, adaptive DR [52] aims to set the randomization parameters such that they best fit observed data from the target environment.

Chapter 3

Methods

In this chapter, we discuss the infrastructure developed for setting up the sim-to-sim transfer scenario and our methodology. We define a constrained driving scenario for the transfer learning experiments. We discuss the customizable HRL framework we developed to serve as the low-fidelity simulator and how we set up a transfer learning scenario from it to the target simulator targeting only the modelling differences. We also explain the algorithm used for the baseline rule-based approach and the architecture of the HRL agent. We discuss the transfer learning approach used and also briefly discuss how we analyze the behaviour of an RL policy.

3.1 Driving Scenario

In this study, we choose a simple, yet challenging driving problem for evaluating behaviour planning models. AVs on public roads have been observed to be especially slow at unprotected left turns and merges [23]. This is not surprising, as these are hard tasks for novice human drivers as well. It requires them to not only judge the trajectory of other fast-moving vehicles but also to be confident in their own driving ability, to find and make use of an opportunity to cross or merge. In busy traffic, the safe window for this is very small and deciding to move at the wrong time could result in disastrous consequences. Due to this, rule-based approaches opt to err on the side of caution, which most often means waiting or going slower until the traffic is more manageable to the system. However, adopting an overly defensive driving strategy is not only inefficient in terms of traffic flow but can also frustrate other drivers [3].

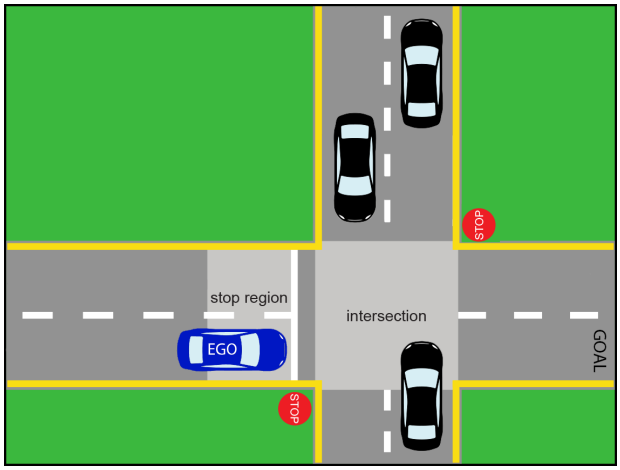

Figure 3.1: The `cross-intersection` scenario.

We consider a scenario where the AV needs to cross a busy stream of traffic, as shown in Figure 3.1. The scenario is as follows: the ego vehicle is waiting at a stop sign where a minor road (horizontal) intersects a major road (vertical), and wishes to cross to the opposite side, by finding a safe gap in between the stream of oncoming vehicles. This can be considered a simplified form of the unprotected left turn scenario. We refer to this as the `cross-intersection` scenario from here onward.

We assume that the ego vehicle follows the same trajectory after each cross decision so that whether it crashes or not depends entirely on the decision timing. This assumption also requires that the ego stays its course once it commits to crossing. There are situations where it is favourable to not fully commit, such as to slow down in the middle of the road or to react to new objects entering the scene. However, to put all the consequence on that one decision, we model it as a 'to-go-or-not-to-go' problem [22], where the `behaviour planner` need only decide when to cross the intersection. The local planner is assumed to then generate the same trajectory each time. It forces a strong association between the decision and the current environment state, as there is no room for correcting a mistake. This makes it easier to analyze the decision boundaries as well as to effectively compare learned and programmed solutions without unduly compromising the study. Indeed, this assumption is acceptable for a deterministic local planner. In Section 3.4, we see that the local planner in Autonomooose does indeed generate similar trajectories given the same driving environment.

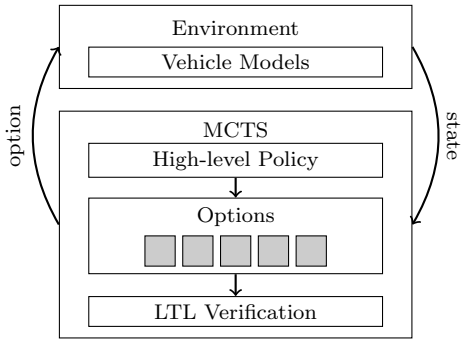

Figure 3.2: Overview of WiseMove planning architecture.

3.2 WiseMove - Low-Fidelity Simulator

To train learned behaviour planning models, we require an architecture similar to the motion planning hierarchy in the Autonomooose software stack. Therefore, as a part of this study, we developed WiseMove [6], a software framework to investigate safe deep RL in the context of motion planning for autonomous driving research. We use it as the low fidelity source simulator for training RL policies for behaviour planning and also to conduct transfer learning experiments.

3.2.1 Overview

WiseMove is an options-based modular deep RL framework for autonomous driving, written in Python, with a hierarchical structure designed to mirror the motion planning architecture of Autonomoose. Options [58, Chap. 17] model high-level temporally abstracted behaviour planning maneuvers, to which are associated low-level local planning policies that implement them. The low-level policies can either be learned or programmed, in advance, each encoding the continuous action space of the ego vehicle. A learned high-level policy over options decides which option to take in any given situation, while Monte Carlo tree search (MCTS [58, Chap. 8]) can be used to improve overall performance during execution. To define correct behaviour and option termination conditions, WiseMove incorporates runtime verification [28] to validate simulation traces and assign additional rewards during both learning and planning.

When an option is chosen by the decision-maker (the high-level policy or MCTS), a sequence of control actions is generated according to the option's low-level policy. An option

terminates if there is a violation of a logical requirement, a collision, a timeout, or successful completion. When an option terminates, the decision-maker then chooses the next option to execute, and so on until the whole episode ends. Figure 3.2 gives a diagrammatic overview of WiseMove's planning architecture. The current state is provided by the environment. The planning algorithm (MCTS) explores and verifies hypothesized future trajectories using the learned high-level policy as a baseline. MCTS chooses the best next option it discovers, which is then used to update the environment.

WiseMove comprises four high-level Python modules: `env`, `options`, `backends`, and `verifier`. The `env` module provides support for environments that adhere to the OpenAI Gym [12] interface, which includes methods to initialize, update and visualize the environment, among others. WiseMove comes with a default intersection environment that enables the learning of hierarchical motion planning models that can navigate the intersection. The `options` module defines the hierarchical decision-making structure of HRL training. The `backends` module provides the code that implements the learned or programmed components of the hierarchy. WiseMove currently supports Keras [14], Keras-RL [46] and Stable Baselines [20] for deep RL training.

The `verifier` module provides methods for checking LTL-like properties constructed according to the following syntax:

$$\varphi = \texttt{F}\ \varphi \mid \texttt{G}\ \varphi \mid \texttt{X}\ \varphi \mid \varphi\ \texttt{=>}\ \varphi \mid \varphi\ \texttt{or}\ \varphi \mid \varphi\ \texttt{and}\ \varphi \mid \texttt{not}\ \varphi \mid \varphi\ \texttt{U}\ \varphi \mid \texttt{(}\varphi\texttt{)} \mid A \tag{3.1}$$

Literal symbols `U`, `F`, `G` and `X` are the standard *until*, *eventually* (*finally*), *always* (*globally*) and *next-state* temporal operators, respectively. The other literal symbols have their obvious meanings. Atomic propositions A are functions of the global state represented by human-readable strings. We use the term LTL to mean properties written according to Equation 3.1. The verifier decides during learning and planning when various LTL properties are satisfied or violated, to assign the appropriate reward. Learning proceeds one step at a time, so the verifier works incrementally, without revisiting the prefix of a trace. WiseMove uses LTL to express the preconditions and terminal conditions of each option, as well as to encode traffic rules.

3.2.2 Scenario Generation

The `cross-intersection` scenario can be easily created by modifying the default intersection environment in the `env` module of WiseMove. An episode starts with ego vehicle, indexed by $i = 0$ or *ego* for readability, stopped on the horizontal road at the stop line, waiting to cross. Figure 3.3 shows the output from WiseMove where the ego vehicle is

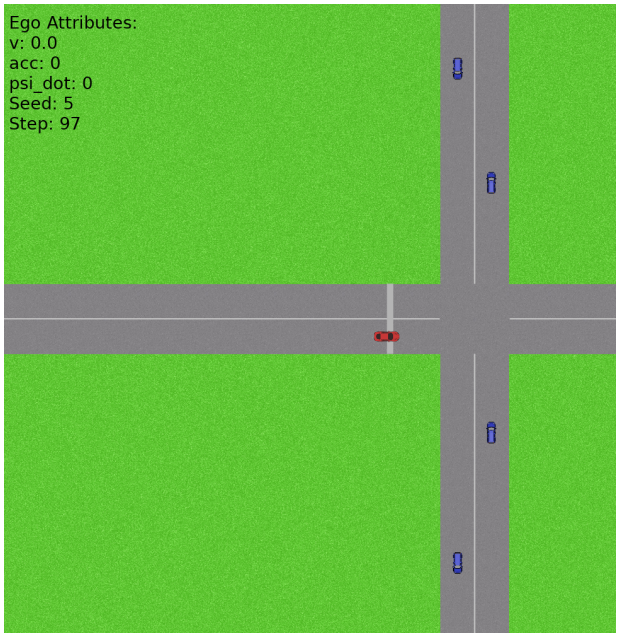

Figure 3.3: The `cross-intersection` scenario rendered in WiseMove

waiting at the stop line. Both the horizontal and vertical roads are bi-directional, with one lane in each direction. No other vehicles are spawned in the horizontal direction. A stream of vehicles travel in both directions in the vertical road, with a maximum of N other vehicles in the scene, indexed by $i \in \{1, ..., N\}$. The only dynamic objects in this environment are these vehicles. With this configuration, the driving goal of the ego vehicle is to safely arrive at the right end of the horizontal route without any collision and, ideally, as fast as possible.

Dynamics

We describe the continuous dynamics of the `cross-intersection` scenario in WiseMove. Let $\phi_{i,\mathsf{veh}} := (X_i, Y_i, \theta_i, v_i, \psi_i)$ and $u_i := (\mathfrak{a}_i, \rho_i)$ be the continuous state and control input, respectively, of vehicle i, with its centre position (X_i, Y_i), velocity v_i, acceleration $\mathfrak{a}_i$, heading angle θ_i, steering angle ψ_i, rate of change of steering angle ρ_i, and wheel base L. Each vehicle i has a maximum velocity $v_i^{\max}$, and a maximum and minimum acceleration, $\mathfrak{a}_i^{\max}$ and $\mathfrak{a}_i^{\min}$. All vehicles have the same maximum steering angle $\psi^{\max}$ and maximum rate of change of steering angle $\rho^{\max}$.

A vehicle's continuous dynamics are thus

$$\begin{cases} \dot{X}_i = v_i \sin\theta_i & \dot{Y}_i = v_i \cos\theta_i & \dot{\theta}_i = v_i \tan(\psi_i/L) \\ \dot{v}_i = \mathfrak{a}_i \quad (|\mathfrak{a}_i| \leq \mathfrak{a}_i^{\max}) & \dot{\psi}_i = \rho_i \quad (|\rho_i| \leq \rho^{\max},\ |\psi_i| \leq \psi^{\max}). \end{cases} \tag{3.2}$$

The state is updated every Δt by numerical integration according to state transition function $\phi'_{i,\mathsf{veh}} = g_{\mathsf{veh}}(\phi_{i,\mathsf{veh}}, u_i)$. We use the previous input $u_{i,\mathsf{prev}} := (\mathfrak{a}_{i,\mathsf{prev}}, \rho_{i,\mathsf{prev}})$, to approximate the jerk by $\dot{\mathfrak{a}}_i \approx (\mathfrak{a}_i - \mathfrak{a}_{i,\mathsf{prev}})/\Delta t$. Defining $\phi_i := (\phi_{i,\mathsf{veh}}, u_{i,\mathsf{prev}})$, the complete continuous dynamics of vehicle i is $\phi'_i = g(\phi_i, u_i)$, where continuous transition function $g(\phi_i, u_i) := (g_{\mathsf{veh}}(\phi_{i,\mathsf{veh}}, u_i), u_i)$. The global state is given by $s := (\boldsymbol{\phi}) \in \mathcal{S}$, with $\boldsymbol{\phi} := \{\phi_i\}^\intercal$, where $\{\cdot\}^\intercal$ denotes the column vector formed by all the indexed elements.

The full update dynamics is given by

$$s' = (\boldsymbol{\phi}')^\intercal = (f(s,a))^\intercal := f(s,a), \tag{3.3}$$

where action $a \in \mathcal{A} = [\mathfrak{a}_i^{\min}, \mathfrak{a}_i^{\max}] \times [-\rho^{\max}, \rho^{\max}]$ is the ego vehicle's control input u_0 and $f(s,a) := \{g(\phi_i, u_i)\}^\intercal$.

Driving Behaviour

Typically, we adopt a different policy μ for the non-ego vehicles. We use a modified version of the aggressive driving policy of [43] and, hence, $u_i = \mu(s)$ for all $i \neq 0$. According to policy μ, vehicle i accelerates smoothly until it reaches its velocity preference $v_i^{\max}$. It also maintains individual vehicle-to-vehicle (V2V) distances $\mathsf{v2v_i}$ with the vehicle ahead. $\mathfrak{a}_i^{\max}$, $v_i^{\max}$ and $\mathsf{v2v_i}$ are sampled randomly for the spawned vehicles each episode, to create variations in individual behaviour. The speed limit for the road is $v^{\max}$. The vehicles do not react to the ego vehicle's actions to further increase the significance of the decision timing, which is in line with the thinking in Section 3.1.

Feature Space

In `cross-intersection`, the ego vehicle's state does not influence the behaviour planner's decision, as we assume it always follows the same trajectory. Therefore, we can safely exclude the ego state ϕ_0 from the observation. Also, all relative distances are calculated with respect to the ego's `base_link` position $(X_{\mathsf{ego}}, Y_{\mathsf{ego}})$ to mirror the Autonomooose stack. Thus, the agent's feature space or observation $o \in \mathcal{O}$, which is a function of state s, is

$$o := \{o_1, o_2, \cdots, o_N\}^\intercal, \tag{3.4}$$

where o_i is the feature vector for vehicle i consisting of features $f \in \mathcal{F}$. $o_i := (x_i, y_i, \theta_i, v_i, \mathfrak{a}_i)$, where $x_i = X_{\mathsf{ego}} - X_i$ and $y_i = Y_{\mathsf{ego}} - Y_i$.

The perception component in Autonomooose cannot reliably identify dynamic objects beyond a perception range d_{max}. Let n be the number of vehicles inside in the perception range. Thus, all features in $o_j \; \forall \; j > n$, denoting vehicles outside the perception range, are set to 0.

3.2.3 Hierarchical Learning

An option in `cross-intersection` is a high-level maneuver. We use the notation $m \in \mathcal{M}$ to denote elements of the set of options, and define $m := (I_m, \pi_m, \beta_m)$. $I_m \subseteq \mathcal{O}$ is the initiation set in which m is available, $\pi_m : \mathcal{O} \to \mathcal{A}$ is a low-level policy - a map from the observation space $\mathcal{O}$ to the action space $\mathcal{A}$, and β_m is the termination condition.

Learning Objective

The objective of a behaviour planning RL agent in `cross-intersection` is to learn a high-level policy, $\Pi^* : \mathcal{O} \to \mathcal{M}$, given a set of programmed low-level policies, $\{\pi^*_m : \mathcal{O} \to \mathcal{A}\}_{m \in \mathcal{M}}$, that maximizes

$$V(s_{0:T}, a_{0:T}, m_{0:K}) := \sum_{t=0}^{T-1} \gamma^t \cdot \mathsf{inst}(o_t, a_t, m_k) + \sum_{k=0}^{K} \gamma^k \cdot \mathsf{inst_o}(m_k) + \gamma^T \cdot \mathsf{term}(o_T), \tag{3.5}$$

where $\mathsf{inst}(\cdot)$ is the instantaneous low-level step reward, $\mathsf{inst_o}(\cdot)$ is the high-level step reward and $\mathsf{term}(\cdot)$ is the terminal reward: $T, K \in \mathbb{N} \cup \{\infty\}$ are the terminal time and decision instants of an episode respectively, $\gamma \in (0, 1)$ is the discount rate, s_t is the state at time t, o_t is the ego's observation given as a function of s_t, $m_k = \Pi(o_{t_k})$ is the option chosen at decision time t_k of decision instant k by the high-level policy and applied until its termination i.e., until the next decision time t_{k+1}, and $a_t = \pi_{m_k}(o_t)$ is the action given by the low-level policy π_{m_k} for $t = t_k, t_k + \Delta t, t_k + 2\Delta t, \cdots, t_{k+1} - \Delta t$.

The inst, $\mathsf{inst_o}$ and term rewards in (3.5) are described as follows:

- $\mathsf{term}(o_T)$ is the reward given at the terminal time T, which is $+r_T$ when achieving the driving goal and $-r_T$ when the run terminates due to collision or violation of the traffic rules.

- $\mathsf{inst}(o, a, m)$ represents the quality of driving when seeing the observation o and taking action a under the option m, where a is determined by the policy π_m for that given option.

- $\mathsf{inst}_\mathsf{o}(m_k)$ is the reward given at the termination of each option, which is a constant negative reward r_o. This is a standard reward specification which encourages the agent to achieve the goal as fast as possible. We set each episode to time out at $K^{\max}$ high-level steps if the ego has not reached its goal and, therefore, $\mathsf{inst}_\mathsf{o}^{\mathsf{max}} = \sum_{k=0}^{K^{\max}} \gamma^k \mathsf{inst}_\mathsf{o}(m_k)$ is the upper bound for this term in (3.5). We choose r_o such that $\mathsf{inst}_\mathsf{o}^{\mathsf{max}} = r_T$, to penalize the agent equally for collisions and timeouts.

WiseMove also has the capability to use model checking during learning and MCTS during execution to ensure adherence to safety constraints. Although these features may provide a better policy, we do not include them in our experiments, to reduce the number of variables in the transfer learning study.

Low-level Policies

Option m terminates when the termination condition β_m becomes true. In this case, the agent chooses the next option and executes it until it terminates. This process continues until the whole episode ends. The `options` module defines β_m of each option as the disjunction of (i) a violation of an LTL requirement, (ii) successful completion, (iii) collision, and (iv) timeout.

In `cross-intersection`, we only require two options, `yield` and `trackspeed`, in order for the ego vehicle to reach its goal. Its behaviour and programmed low-level policies are as follows:

1. `yield`: The ego vehicle stays in a stopped position to yield to oncoming traffic. The ego's control input $u_0 = (0, 0)$.

2. `trackspeed`: The ego vehicle accelerates along its travel route to the desired speed and then maintains it. The ego's control input is

$$u_0 = \begin{cases} (\mathfrak{a}_0^{\max}, 0), & \text{if } 0 \leq v_0 \leq v_0^{\max} \\ (0, 0), & \text{otherwise} \end{cases}$$

An ideal high-level policy chooses `yield` at time $k \in (0, K-1)$ until it detects an opportunity to cross from the environment state, where it chooses `trackspeed` at K. Choosing

`trackspeed` ends the episode and returns whether the decision resulted in a failure or a success by rolling out the trajectory in the background using the programmed `trackspeed` maneuver following the 'to-go-or-not-to-go' [22] problem definition.

Metrics

We use two metrics to assess the performance of a policy over a test set of randomized episodes. The first metric, `success`, captures the ability of the agent to solve the `cross-intersection` successfully. It is the number of episodes in which the agent reached the goal point, expressed as a percentage of the total number of test episodes.

The other metric is `wait-time` calculated as the average time spent waiting at the intersection. `wait-time` for one episode is the number of high-level steps where the maneuver output is `yield`. This quantifies the aggressiveness of the agent and is incentivized by the negative reward per time step r_o and the collision penalty $-r_T$. Being overly defensive or aggressive detrimentally affects the `success` metric as either the episode will time out or the agent may crash by being too hasty.

Parameters

After some trial and error, we set the maximum number of vehicles N to be 5. To have a minimum level of complexity, an episode will always have at least 2 vehicles. For rewarding the agent fairly, we set $r_T = 12$ and $r_o = -0.04$. $\mathsf{inst}(o, a, m)$ is set to always evaluate to 0 since the driving trajectories for each option are determined by fixed low-level policies, and therefore the driving performance need not be optimized by the learning algorithm. Also, an option can only terminate through a `timeout` to mirror the Autonomooose stack. We set a simple fixed `timeout` of $\Delta K = 0.1$ seconds which is the decision frequency of the `behaviour planner` in Autonomooose. Thus, by setting $K^{\max} = 300$, an episode is at most 30 seconds.

In the Autonomooose platform, the range at which the perception component can reliably identify dynamic objects is around 50 metres. However, vehicles beyond this range can still collide with ego considering the range of velocities possible. Thus, this perception range does not allow ego to consistently succeed in the `cross-intersection` scenario. Therefore, we set d_{max} to 80 metres so that a policy can theoretically achieve 100% success.

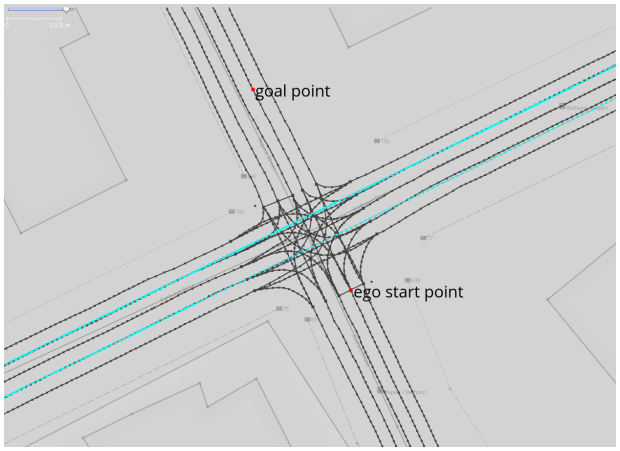

Figure 3.4: Intersection lanelet map with ego's start location, goal point and vehicle paths (light blue) in UNREAL.

3.3 Unreal - High-Fidelity Simulator

The UNREAL simulator is the target environment for the RL policies trained using WISEMOVE. We define the `cross-intersection` scenario in UNREAL using the Geoscenario format. In order to provide a fair comparison for transfer learning, we also replicate each test episode generated in WISEMOVE in the UNREAL simulator by dynamically creating the Geoscenario OSM file during evaluation. We also provide a way for the `behaviour planner` to seamlessly switch between rule-based and learned behaviour planning models.

3.3.1 GeoScenario Definition

Since Geoscenarios are defined on top of an existing lanelet map, we have to choose the map before constructing the OSM. We choose the lanelet map created for the Bathurst, Waterloo area and pick an intersection for testing, which is shown in Figure 3.4. The ego start location and goal point, shown in the figure, are added to the OSM as nodes with latitude-longitude coordinates. This basic Geoscenario file, containing no dynamic objects, is used as the base for all `cross-intersection` tests in the UNREAL simulator. Dynamic object trajectories can be added using nodes and ways to this file, to generate one episode of the `cross-intersection` scenario.

3.3.2 Importing Tests from WiseMove

Since we evaluate a policy over multiple random episodes, the base Geoscenario OSM is updated with the dynamic object trajectories for each episode. The trajectories are replicated from WiseMove so that the policy evaluation is not affected by test set differences. To achieve this, we generate a set of episodes for testing in WiseMove and save the trajectories of all dynamic objects in a JSON[1] file. The saved trajectory contains the position and speed at each high-level timestep in the `base_link` frame of reference.

The saved trajectories can be imported into Unreal to replicate an episode by converting them to the Geoscenario specification. We generate Geoscenario nodes and ways for each trajectory by transforming the positions from `base_link` to the required latitude-longitude coordinates and add them to the base OSM file. We can safely use `base_link` origin in the Unreal simulation as the reference for this transformation since the scenario always starts with ego stopped at the stop line. Trigger nodes are used to make a dynamic object start moving along its path at the timestep specified in the JSON. The vehicle paths in the OSM after importing a WiseMove episode is shown in blue in Figure 3.4.

We also ensure that the test set on which a policy is evaluated is the same across trials in both simulators. A random seed can be set for each episode generated in WiseMove which makes it repeatable across trials. We generate a test set using the same sequence of seeds each time. This allows individual episode comparison, which makes policy comparison more effective and also makes debugging easier.

3.3.3 Policy Server

As seen in Section 2.1.3, the `behaviour planner` queries a `rule engine` to generate a decision. However, we require the use of both rule-based policies and learned RL policies from WiseMove for this study. Therefore, we modify the behaviour planning architecture as shown in Figure 3.5 and create a new ROS node, `policy server`, that takes the place of the `rule engine`. The `behaviour planner` processes its input normally but does not create the abstracted representation required by the `rule engine`. Instead, it sends the processed inputs directly to the `policy server`.

The `policy server` can be configured to use a specific behaviour planning policy, which in this case is either a rule-based policy or a learned model that solves the scenario. The `policy server` is responsible for further processing the inputs as required by the selected

[1] JavaScript Object Notation to store and transport data.

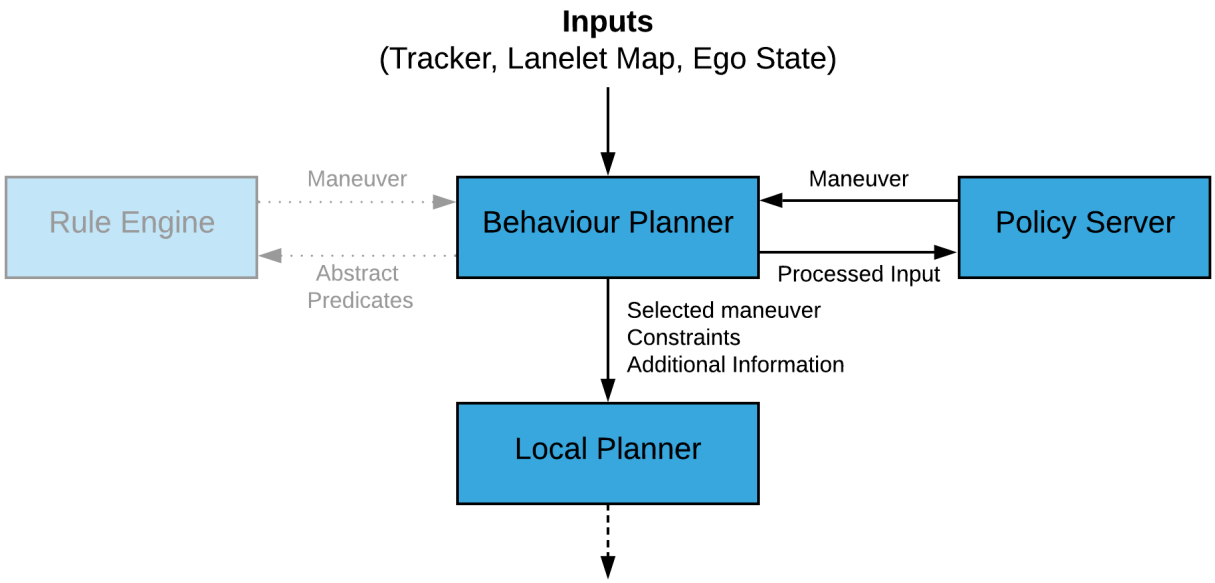

Figure 3.5: Information flow in the modified `behaviour planner` architecture. Unused components are greyed out. Blue boxes represent ROS nodes.

policy. For this study, it uses the same processing steps used in WiseMove to create the observation o specified in Equation 3.4. Therefore, o is the same in both Unreal and WiseMove, given the same environment state s. A learned behaviour planning model is loaded in `policy server` using the same RL library used in the WiseMove `backends` module for ease of transfer.

3.4 Aligning WiseMove with Unreal

To isolate the transfer gap between the simulators, we eliminate or minimize the differences that do not arise from how the simulator is modelled. For this, we first analyze and compare the *factors* that can affect the agent's observation in the simulation. For the `cross-intersection` scenario, the factors are ego dynamics, road map, dynamic object trajectories, and observation noise. We then identify whether it is a simulator modelling gap and if not, adjust or *align* the WiseMove environment with Unreal such that the gap is minimized. After performing this process, we obtain a `cross-intersection` environment in WiseMove, denoted the `wm` environment and a counterpart in the Unreal simulator, denoted `unreal`, which have only modelling differences as the transfer gap.

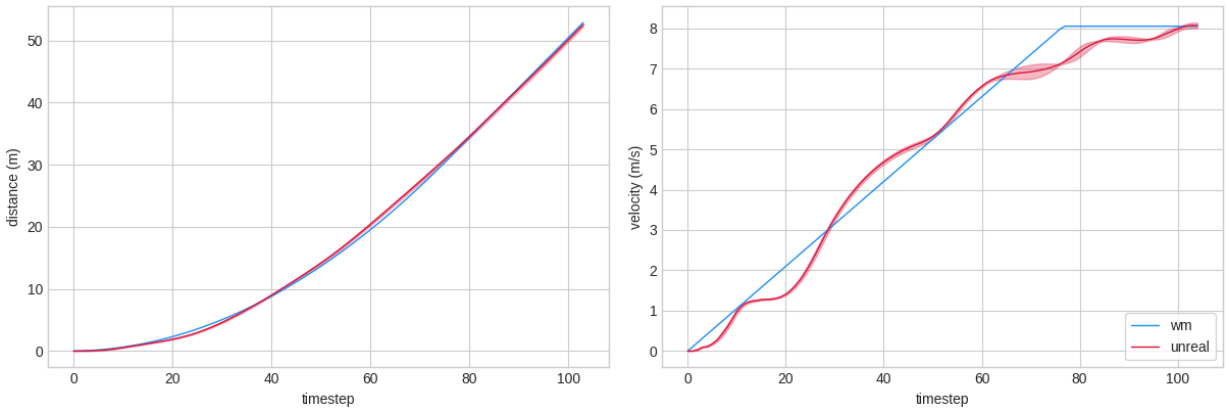

Figure 3.6: Distance covered by ego (left) and velocity per timestep (right) for `trackspeed` maneuver

3.4.1 Ego Dynamics

For `cross-intersection`, we only compare the `trackspeed` of the `behaviour planner` in UNREAL and the programmed `trackspeed` low-level policy in WISEMOVE, since the `yield` maneuver's behaviour is simple. Although this discrepancy is a simulator modelling gap, caused by the simplistic ego dynamics and `trackspeed` model in WISEMOVE, we do not attempt to bridge this gap in this study. We consider this to be a crucial transfer gap that needs to be studied separately using domain adaptation techniques or by using the actual ego vehicle model and local planner of UNREAL. Therefore, we leave that to future work and assume that ego's `trackspeed` maneuver is aligned to UNREAL in `wm`. This can be done easily as only the distance covered per time step by ego affects the behaviour planning policy. We adjust $\mathtt{a}_{\mathtt{ego}}^{\max}$ and $v_{\mathtt{ego}}^{\max}$ in WISEMOVE to align the behaviour of `trackspeed` with UNREAL, although there is a slight discrepancy as seen in Figure 3.6. The data used for generating the plots is collected starting from the moment ego starts moving, over multiple trials in the UNREAL simulator. We observe that it generates a similar trajectory each time, which is expected of the deterministic local planner in Autonomooose.

3.4.2 Road Map

It is clear that differences in the road map are not a result of a simulator modelling gap and therefore, we align it with UNREAL. The selected intersection from Section 3.3 has the following properties:

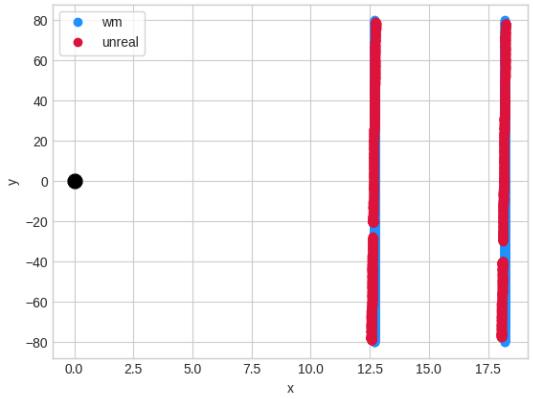

Figure 3.7: Vehicle paths for intersection scenario in WiseMove and Unreal. Ego is at (0,0).

- Lane width: 5.5 metres
- Distance from the stop line to the intersection: 7.2 metres

We adjust the intersection in `wm` to reflect these values. A top-down view of the vehicle paths after alignment is shown in Figure 3.7 with respect to ego at `base_link` origin.

3.4.3 Dynamic Objects

Although we convert the WiseMove trajectories to the GeoScenario format accurately, its interpretation by the Unreal simulator may cause some changes. To confirm this, we analyze the observed trajectory of a single vehicle generated in WiseMove by importing it into Unreal as a GeoScenario OSM. Figure 3.8 plots x and y feature at each high-level timestep. The y feature observed from Unreal is slightly noisy, which could result from either the trajectory interpretation or from perception errors. On analyzing the ground truth positions of the dynamic objects provided by Unreal, we conclude that the GeoScenario interpretation is accurate and therefore, this noise is the result of perception errors.

The estimated velocity v of the dynamic object differs significantly in Figure 3.9. We attribute this to the Kalman filter used for velocity estimation in the `tracker` which requires a few measurements before it can produce an accurate estimate. Thus, it takes an

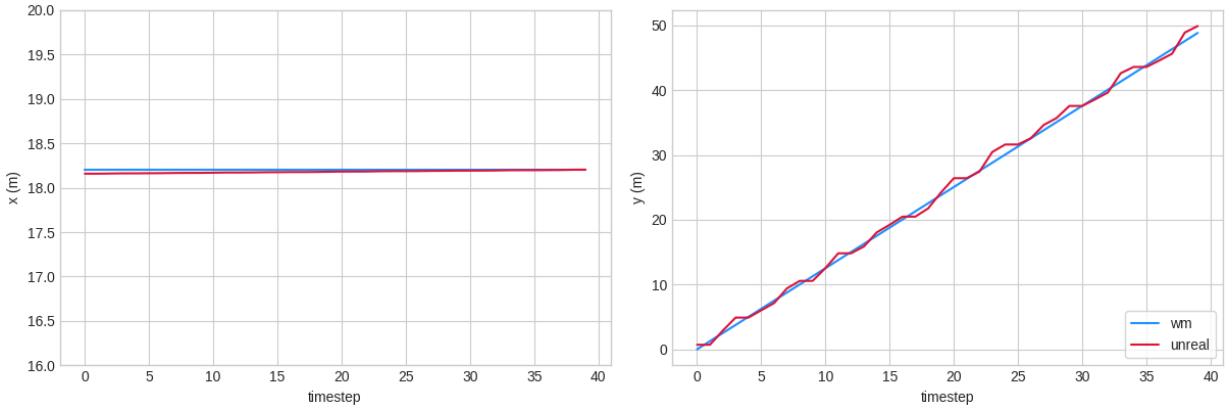

Figure 3.8: x and y feature of a dynamic object over time in WiseMove and Unreal.

average of 11 high-level steps to reach a stable estimation, as seen in Figure 3.9. Moreover, the final estimation is also underestimated by an average of 10%, which we suspect is due to the consistent error in position estimation seen in Figure 3.8, where y stays the same for a few timesteps. The y noise and v error are inherent modelling differences between the simulators and, thus, are not adjusted in `wm`.

The `tracker` does not provide the acceleration and we have to infer this from the velocity. However, since the velocity estimation is itself unreliable, the acceleration calculated is not very accurate. Therefore, we choose to remove this from the observation of the agent, such that $o_i := (x_i, y_i, \theta_i, v_i)$ in Equation 3.4.

3.4.4 Perception Lag and Other Errors

In the Autonomooose stack, producing the inputs to the `behaviour planner` from raw sensor data involves significant computation. This results in a time lag between the input used for decision making and the current state of the environment, which on average is found to be 0.235 seconds. Also, there is a time delay between the output of the `trackspeed` decision and ego starting to move. This delay is measured to be 0.105 seconds on average. Thus, the total average perception lag, $t_{\mathsf{lag}} = 0.34$ seconds.

Also, there also exist other occasional perception errors. The first one occurs when the `tracker` confuses dynamic objects close to each other, resulting in wrong trajectory estimations. This can be seen in Figure 3.10, where vehicle 8, moving to the left of the image, is mislabelled as vehicle 2 (moving right), which causes a sharp change in the

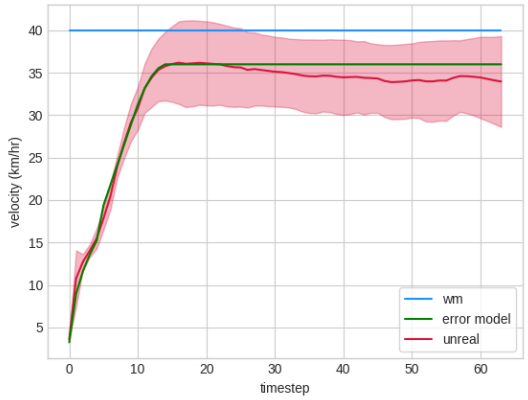

Figure 3.9: Velocity v of a dynamic object over time in WiseMove and Unreal.

estimated trajectory of vehicle 2. The second error occurs when a vehicle does not get detected for a few timesteps for some reason, causing it to essentially vanish and then re-appear as a 'new' vehicle in the scene. This 'vanishing vehicle' problem can be observed as a gap in the plotted vehicle path in Figure 3.7. All perception errors are categorized as simulator differences.

3.5 Naive Perception Error Model

The transfer gap caused by perception errors observed in Section 3.4 can be closed simply by modelling them in WiseMove. Accurately modelling them is a non-trivial task. However, we can roughly model them using the observed data, which may help the RL policy perform well in the target environment. Thus, we create a variant of the `wm` environment called `lagkf` that naively models the perception lag and `tracker` velocity estimation errors. `lagkf` lags the environment observation by t_{lag} time steps using a queue data structure to simulate the average lag in the Unreal simulator. `lagkf` also models the velocity estimation error by underestimating by a fixed 10% and by simulating the initial stabilization time. The observed velocity using the error model is shown in Figure 3.9. We also create the environments `kf` and `lag`, which models only the velocity estimation error and the perception lag respectively, to study their individual effects.

The `lagkf` environment is also used for validation testing. The Unreal simulator is not ideal for running multiple rounds of testing as the scenarios run in real-time. There also

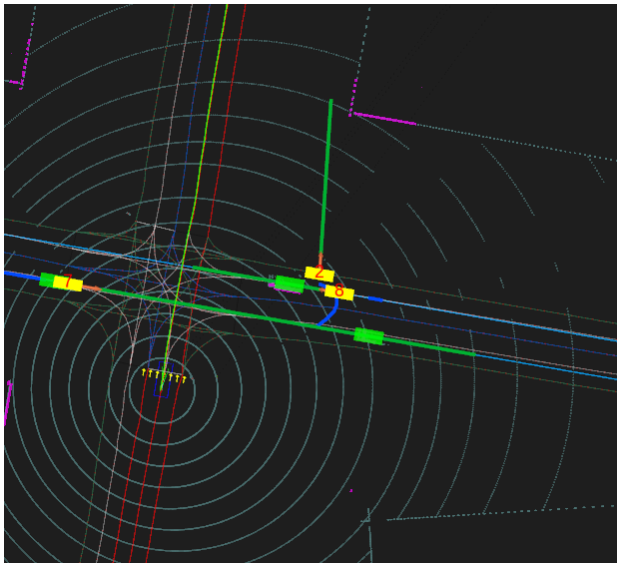

Figure 3.10: Incorrect labelling of vehicle 2 by `tracker` causes sharp erroneous change in trajectory.

exists a ROS restart overhead for each run, making a scenario of 30 seconds take on average around 50 seconds to run. This makes it impractical to use it for validation testing during training. Thus, using `lagkf`, we can quantify and evaluate the success of transferring a behaviour planning policy without actually testing in the Unreal simulator.

3.6 Behaviour Planning Models

In this section, we describe the policies used for solving `cross-intersection`. We create a rule-based policy that can solve the `cross-intersection` consistently with perfect information and modify it using the insights about the target environment in Section 3.4 to make it more successful. We also describe the architecture of the RL policy trained in WiseMove.

3.6.1 Rule-based

To provide a baseline for solving this scenario, we create a rule-based behaviour planning policy that uses a time-to-collision (TTC) based algorithm to calculate when to cross the

intersection.

TTC Algorithm

At each high-level step, we consider all vehicles that are moving towards the ego's intended route. For each such vehicle i, we calculate the time ttc_i taken to reach a point $p_c = (X_c, Y_c)$ in the ego's route that intersects with the centre of the vehicle's current lane. This is the point of collision between ego and vehicle i. The distance between the observed position of the vehicle i, $p_i = (X_i, Y_i)$ and p_c is $d_i = \|p_c - p_i\|$, where $\|\cdot\|$ is the Euclidean distance. It is assumed that the vehicle's velocity v_i stays the same beyond the current timestep. We use the standard kinematic equations of motion for calculating the time:

$$\mathsf{ttc}_i = \frac{d_i}{v_i}$$

Since we have modelled ego's `trackspeed` maneuver, we can also calculate time $\mathsf{ttc}_{\mathsf{ego}}$ that ego takes to reach p_c in the route. The distance ego has to travel to reach p_c is $d_{\mathsf{ego}} = \|p_c - p_{\mathsf{ego}}\|$. Since the `trackspeed` maneuver in WiseMove accelerates uniformly with maximum acceleration from a stopped position,

$$\mathsf{ttc}_{\mathsf{ego}} = \sqrt{\frac{2d_{\mathsf{ego}}}{\mathfrak{a}_{\mathsf{ego}}^{\max}}}$$

Thus, $\mathsf{ttc}_{\mathsf{i,ego}} = \ |\ \mathsf{ttc}_i - \mathsf{ttc}_{\mathsf{ego}}\ |$, where $|\cdot|$ is the absolute value or modulus, provides us with the time by which ego would have missed a collision with vehicle i. The chance of a collision is greater the closer this value is to zero. If the value is greater than a threshold t_{safe} for any vehicle, it is safe for the ego to cross. Thus, a simple rule-based behaviour planning policy $\Pi_{\mathtt{ttc}}$ for `cross-intersection` is defined as:

$$m_k = \begin{cases} \texttt{yield}, & \text{if } \mathsf{ttc}_{\mathsf{i,ego}} \leq t_{\mathsf{safe}}, \text{for any } i \\ \texttt{trackspeed}, & \text{otherwise} \end{cases}$$

t_{safe} is set to 1.5 seconds to have an extra buffer and to account for the vehicle length. The current observation o of the RL agent in WiseMove does not contain the TTC for each vehicle. Thus, the inputs of the rule-based and the learned policies are different. This does not allow for a fair comparison even though TTC is a derived feature calculated from position and velocity, which are both present in o. Therefore, we augment the observation o with the derived feature ttc_i such that $o_i := (x_i, y_i, \theta_i, v_i, \mathsf{ttc}_i)$ in Equation 3.4. The ttc_i value for any vehicle that is not on a collision course is set to a very large value, `MAX_FLOAT`.

Handling Perception Error

The above algorithm assumes we have near-perfect information about the world and may fail if there are observation errors or noise. Using our knowledge of the target environment from Section 3.4, we can make this policy more robust to the perception errors present in UNREAL by:

- Using the predicted position $\hat{p}_i = (\hat{X}_i, \hat{Y}_i)$ of the vehicle to account for the perception lag. Thus, distance to p_c becomes $\hat{d}_i = \|p_c - \hat{p}_i\|$
- Waiting until the velocity estimation stabilizes. Since the algorithm does not utilize history, this is achieved by ignoring velocities under a threshold v_{th} as they are unreliable. We set $v_{\mathsf{th}} = 26$km/hr as we observe it to be the lower bound for velocity in generated episodes.
- Increasing observed velocity by 10% to compensate for the `tracker` under-estimation.

Thus, $\mathsf{ttc}_i = \dfrac{\hat{d}_i}{1.1 v_i}$ and the resulting policy is:

$$m_k = \begin{cases} \texttt{yield}, & \text{if } \mathsf{ttc}_{\mathsf{i,ego}} \leq t_{\mathsf{safe}} \text{ or } v_i \leq v_{\mathsf{th}}, \text{ for any } i \\ \texttt{trackspeed}, & \text{otherwise} \end{cases}$$

This gives us an arguably better rule-based policy $\Pi_{\texttt{r-ttc}}$ which uses knowledge about the target environment.

3.6.2 Reinforcement Learning

Using the WISEMOVE framework, we can learn a behaviour planning RL agent for solving this scenario. We denote a learned RL policy by Π_E, where E is the name of the environment it is trained in, such as `wm` or `lagkf`. A double dueling DQN is used for learning RL behaviour planning policies. The Q-network is a four-layer fully connected neural network whose architecture is shown in Figure 3.11. Each dense layer D consists of 32 nodes, except the final output layer D_o having 2 outputs, corresponding to the high-level actions `yield` and `trackspeed` respectively. It takes as input the observation o in Equation 3.4. The first layer D_{shared} transforms the features of each vehicle o_i to produce h_i, an encoded latent representation of each vehicle. The weights of D_{shared} are shared between each vehicle such that the same feature encoding process is learned and applied, thereby making the network

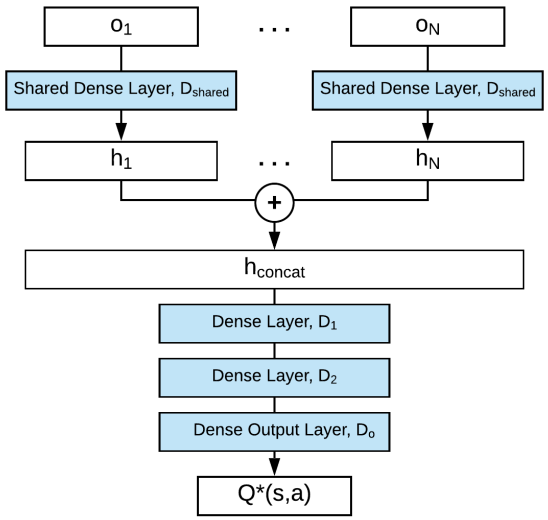

Figure 3.11: Architecture diagram for Q-Network.

invariant to the order of the vehicles. Every layer, except the last, uses the Rectified Linear Unit (ReLU) activation function. The final layer uses a linear activation function since it outputs the Q-value for each action.

The DQN agent stores and samples the experiences randomly from an experience replay buffer during training. During the course of random exploration and exploitation, the buffer may accumulate an unequal distribution of success and failure cases, which can destabilize the training. We design a dual experience replay buffer similar to the architecture in [22] that stores success and failure trajectories separately. The experiences are sampled equally from both during training and it is observed that this helps the agent converge faster.

Keras-RL and Keras in the `backends` module are used to define the double dueling DQN agent with the dual experience replay. Keras-RL supports the standard OpenAI Gym interface, *reset* and *step*, for enabling the interaction between the agent and the WiseMove environment. The reset function generates a new random episode of `cross-intersection` and returns the initial observation o to the agent. The agent then uses the step function to execute actions (maneuvers in this case) inside the environment until termination.

3.7 Domain Randomization

Although `lagkf` models some of the simulation differences and is built on top of `wm`, which closes the non-essential gaps, it does not completely close the transfer gap. If we wish to improve the *transfer success*, which is the `success` metric obtained on evaluating a policy in the target environment, we need to close this gap further. However, it is also not trivial to accurately model all observed differences between `wm` and UNREAL. Domain randomization (DR) for fine robotic control policies has been shown to improve the transfer success considerably [41]. We apply this technique in an HRL setting to define new environments that have randomized parameters to train on. All the environments described in the following paragraphs are defined on top of `lagkf`.

Perception Lag

The perception lag in `lagkf` environment is modelled using the average value of t_{lag}. In the UNREAL simulator, however, it varies a little to either side of this value. The RL agent can benefit from being exposed to variations of this parameter, which we model in a new environment `lag-dr` by sampling the lag from a Gaussian distribution $\mathcal{N}(\mu = t_{\mathsf{lag}}, \sigma = 0.5)$. We also use `lag-dr` to evaluate the individual effect of a deviation from the average perception lag on the transfer success

Velocity Estimation Model

Similarly, the velocity estimation error model in `lagkf` is a simple approximation of the observed velocity v in UNREAL. It is not easy to reproduce this accurately without fitting a model, but applying DR here is trivial. Rather than applying a fixed 10% under-estimation after the initial stabilization time, we under-estimate by a value sampled from the Gaussian distribution $\mathcal{N}(\mu = 0.1v, \sigma = 0.05v)$. This, however, produces a high variance signal compared to individual velocity plots obtained from UNREAL. Therefore, we smooth it using a 5-point moving average filter to obtain an error model closer to the observed data as shown in Figure 3.12. The resulting environment, `kf-dr`, thus applies DR to the velocity estimation.

Vanishing Vehicles and Misdetection

In Section 3.4, we observed a perception error where vehicles do not get detected for a few timesteps, causing the vanishing vehicles issue. Moreover, this also causes the velocity

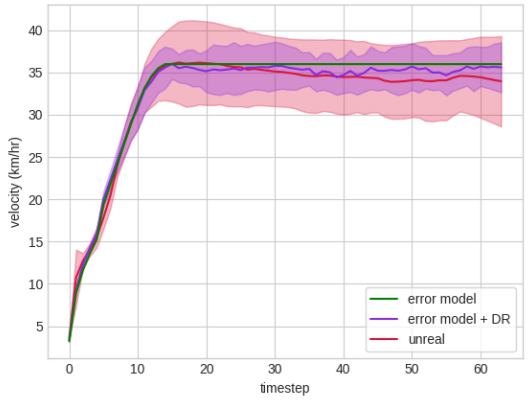

Figure 3.12: Domain randomization of velocity estimation.

estimation to reset to 0 when it re-appears, as the `tracker` module considers it as a new vehicle in the scene. Therefore, this is a significant difference that the agent needs to consider during learning. We include this behaviour in the `vanish` environment as a random phenomenon that triggers a vanishing vehicle with a probability of 0.005 every high-level timestep. Once triggered, the vehicle is excluded from the observation o for the next few timesteps (sampled uniformly between 1 and 10). Note that these values are based on only a few data points from UNREAL, as it is a rare event. When the vehicle re-appears, WISEMOVE treats it as a new vehicle, which is, therefore, subject to the average velocity estimation error model in `vanish`.

We do not model the misdetection error shown in Figure 3.10 because of its unpredictable interaction with other dynamic objects and the complex trajectory changes it induces. Since this phenomenon is rare, we assume its impact to be minimal and, therefore, leave it to be studied in future work.

Position Noise

We also observed minor differences in the observed dynamic object positions between `wm` and UNREAL. The y feature of the dynamic objects had minor estimation errors and the x feature differed due to a slight angle. The `xy-dr` environment applies DR to both the x and y feature, but in different intensities corresponding to the discrepancy in Figure 3.8. y is varied by a zero-mean Gaussian $\mathcal{N}(\mu = 0, \sigma = 0.75)$, while x, by $\mathcal{N}(\mu = 0, \sigma = 0.025)$.

Environment	Lag		Velocity		Vanishing Vehicles	Position DR
	Mean	DR	Mean	DR		
`wm`						
`lag`	✓					
`kf`			✓			
`lagkf`	✓		✓			
`lag-dr`		✓				
`kf-dr`				✓		
`vanish`					✓	
`xy-dr`						✓
`percept`		✓		✓	✓	✓

Table 3.1: Overview of the changes in each variation of `cross-intersection` environment in WiseMove. All environments are aligned with Unreal.

Combined Model

We also merge the above environments in `percept` to study the combined effect of these DR parameters on the transfer success. For reference, Table 3.1 shows an overview of all the defined environments and their features.

3.8 Policy Analysis

To obtain some insights on the RL policy, we first find which features it relies on for optimal decision making. We also extract the behaviour of the policy for further analysis by running a supervised learning algorithm on a dataset sampled using the agent.

3.8.1 Feature Importance

We use the Permutation Feature Importance method [16] that measures the importance of a feature as directly proportional to the drop in performance after permuting it in a dataset. This method is used for supervised learning using Random Forests in the cited work, but the concept can be applied to RL. For supervised learning, this involves training a model and evaluating it on a test dataset to get a baseline performance, p_{base}. The column for feature f in the dataset is randomly shuffled (to preserve the distribution) and

the same model is evaluated again to obtain p_f. The importance of the feature f is given by

$$I_f = p_{base} - p_f$$

, where higher values signify more importance. This process is performed for all features to obtain a feature importance order. Applying this to an RL policy only requires knowledge of the distribution of each feature $f \in \mathcal{F}$ in the test set used for evaluation.

In this study, the test set remains the same for each trial, as we use the same sequence of seeds for scenario generation. This makes it trivial to obtain the actual distribution $\mathcal{D}_f$ of each feature. During the evaluation for obtaining p_f, the value of f in each observation $o_i \in o$ from the environment is replaced by a sample from the corresponding distribution $\mathcal{D}_f$, to replicate the permutation process in supervised learning. Thus, I_f is calculated for each feature and they are then sorted by descending order to obtain an importance order.

3.8.2 Policy Distillation

Using just feature importance, it is not possible to discern the behaviour of the policy. We attempt to understand how the features are used by extracting or *distilling* an interpretable algorithm from the policy.

First, we generate a labelled dataset that maps the observation o_t to the decision $m_t \in \mathcal{M}$ taken by the agent at timestep t. To obtain samples for the dataset, the agent uses policy Π to solve a large set of randomly generated episodes in WiseMove. However, we do not require the observation and decision at every step. We are primarily interested in the last two observations at the end of each episode where it transitions from a `yield` to the `trackspeed` maneuver. Since the difference between these two observations causes the Q-network to change its decision, it is sufficient to provide us with a decision boundary. To avoid misleading data points, only the successful episodes are included. The dataset is then used for training a supervised learning algorithm in order to obtain a decision boundary. Since we are comparing with a rule-based approach, we use a decision tree classifier that can provide an if-else-then interpretation of the distilled policy.

Chapter 4

Experiments and Results

4.1 Training and Evaluation

We perform an initial hyperparameter search for the DQN with the `wm` environment using a grid search and set the learning rate $\alpha = 0.0004$ and discount factor $\gamma = 0.99$. The total size of the dual experience replay buffer is set to 100,000 with a sampling batch size of 32.

We train an RL policy for a maximum of 150,000 steps, but training usually does not reach this far as we employ early stopping. Every 2500 steps, which we consider as 1 epoch, we evaluate the policy on a validation set of 100 episodes generated using a fixed seed sequence. The training is stopped early if the validation performance has not improved over the past 10 epochs. At the end of the training, the policy with the best validation performance is chosen.

The DQN uses a modified epsilon-greedy policy with $\epsilon = 0.3$ for exploration. Instead of a fixed epsilon, however, we linearly anneal it from 1.0 to 0.3 over the initial 15,000 steps, to promote more exploration in the beginning. Also, in each epoch following the annealing, epsilon is tuned using the validation performance such that a better performing policy will have a lower epsilon (up to a minimum of 0.01 at 100% `success`). This is to ensure that better policies explore less and exploit more. We perform this training process multiple times and select the best 10 policies $\mathbf{\Pi}_{\mathsf{E}}$ in terms of the `success` metric, to account for variance in training.

We evaluate each set of policies on WiseMove in its corresponding source environment, which is the training environment for RL policies and `wm` for rule-based policies. The policy is executed on a test set test_E consisting of 1000 episodes generated from environment E

Π	success		wait-time
	wm	unreal	unreal
random	71.70	68.0	2.1
ttc	100.0	90.5	8.29
wm	99.05	90.125	13.46

Table 4.1: Results for baseline policies.

using the same sequence of seeds (different from training) for consistency across trials. To quantify the transfer success of a policy, it is evaluated on the UNREAL simulator. Ideally, we should use for the same number of episodes as in WISEMOVE but it takes a long time to run 1000 episodes for a single policy on UNREAL. It is also time-consuming to evaluate all 10 policies in the set $\mathbf{\Pi}_{\mathsf{E}}$. Therefore, we create a smaller test set $\mathsf{test}_{\mathtt{unreal}}$ in UNREAL by importing the first 200 episodes of $\mathsf{test}_{\mathtt{wm}}$ using the method outlined in 3.3, and select the best four policies for evaluation in $\mathsf{test}_{\mathtt{unreal}}$.

4.2 Baselines

We first establish some baseline results in the `wm` environment. The policy $\Pi_{\mathtt{ttc}}$ policy is expected to perform very well here since `wm` has perfect information and no perception lag. We train and evaluate RL policy $\mathbf{\Pi}_{\mathtt{wm}}$ to study how it fares given the same information. A lower bound to the `success` metric is established by evaluating a random policy $\Pi_{\mathtt{random}}$ over multiple trials.

The baseline policies, however, may not perform well in the UNREAL simulator due to the perception errors. The results are shown in Table 4.1 where it is indeed observed that the policies fail in around 10% of episodes in UNREAL due to the transfer gap. It is also seen that the RL policy takes more time to achieve the goal.

4.3 Naive Error Model

We make use of the `lagkf` environment to train policy $\mathbf{\Pi}_{\mathtt{lagkf}}$. As described in Section 3.5, `lagkf` considers the major perception errors by modelling the mean perception lag and velocity estimation error. Also, their individual contributions to closing the transfer gap are analyzed by training policies in `lag` and `kf` environments respectively and evaluating

Π	env diff % (w.r.t wm)	success		wait-time
		source	unreal	unreal
r-ttc	-	100.0	90.75	7.41
wm	0	99.05	90.125	13.46
lag	0.15	99.17	91.75	12.59
kf	4.19	99.10	95.75	20.53
lagkf	4.99	99.07	95.875	20.56

Table 4.2: Results for policies trained with naive perception error model.

them on UNREAL. These are compared against the robust rule-based policy $\Pi_{\texttt{r-ttc}}$, which is designed using the same information. wm is the base environment for this experiment, with the other environments built on top of it.

For each RL policy $\mathbf{\Pi}_E$, we calculate the loss in performance incurred when transferring the policy $\Pi_{\texttt{wm}}$ learned in the base environment to the training environment E. This quantity, denoted **env diff %**, is the difference between E and the base environment with respect to its effect on the RL agent. The results are shown in Table 4.2, where it can be seen that the perception lag has little impact on the policy (only 0.15% drop in performance for $\Pi_{\texttt{wm}}$), even though the dynamic objects could have moved as much as 4 metres from their original positions within this timeframe. The velocity estimation error is a large contributor to the transfer gap and modelling it even naively results in a large improvement to the policy, although it does increase the average wait-time. Both together gives a very good performance in the target simulator. However, we do not see similar results for the robust rule-based policy $\Pi_{\texttt{r-ttc}}$, even though the same information about the transfer gap was considered when designing it. The performance is only as good as the normal TTC-based policy in UNREAL, which is surprising. The cause of this is not very clear. It may be that a universal 10% increase in velocity estimation is doing more harm than good, as it marks up accurate estimates as well.

4.4 Domain Randomization

The previous experiments resulted in a policy that failed in only around 4% of the total number of episodes in the target simulator. Although the scope for improvement is less, we attempt to further improve the performance of $\Pi_{\texttt{lagkf}}$ using domain randomization. The remaining differences – namely nuanced perception lag and velocity estimation, the vanishing vehicle phenomenon and position estimation errors – are modelled as variations

Π	env diff % (w.r.t `lagkf`)	success		wait-time
		source	`unreal`	`unreal`
`lagkf`	0.0	99.07	95.875	20.56
`lag-dr`	0.0	99.11	96.75	21.47
`kf-dr`	0.35	98.87	96.75	20.08
`vanish`	4.02	95.71	94.25	26.51
`xy-dr`	0.10	98.87	96.375	18.62
`percept`	3.86	95.81	95.50	27.66
`best`	0.41	98.89	97.25	21.04

Table 4.3: Results for policies trained with domain randomization.

in the base environment `lagkf` using DR. Thus, we train and evaluate policies in `lag-dr`, `kf-dr`, `vanish` and `xy-dr`, and also their combination `percept`. Similar to the previous experiment, we measure the difference in each training environment with respect to the base environment policy $\Pi_{\texttt{lagkf}}$. The results are shown in Table 4.3.

The `vanish` environment detrimentally affects the transfer success. Considering that the drop in performance for $\Pi_{\texttt{lagkf}}$ in `vanish` is 4.02% and the performance of $\mathbf{\Pi}_{\texttt{vanish}}$ in the source environment itself only reached 95.71%, we can conclude that the RL algorithm cannot solve the vanishing vehicle problem. This is because the current input does include the history and, therefore, the agent is not able to recognize this phenomenon.

The changes in `kf-dr`, `lag-dr` and `xy-dr` environments are small, since there is almost no performance loss for $\Pi_{\texttt{lagkf}}$. However, they all lead to an increase in performance in the UNREAL simulator. The performance of $\mathbf{\Pi}_{\texttt{percept}}$, on the other hand, suffers due to the inclusion of the vanishing vehicles model. Thus, the best performing policy $\mathbf{\Pi}_{\texttt{best}}$ is obtained by combining the changes in `kf-dr`, `lag-dr` and `xy-dr` environments. Its results are also shown in Table 4.3 and it indeed has the best transfer success out of all the policies.

We experimented with other techniques such as feature discretization and feature noise injection to further improve the transfer success but these did not improve the performance further. It may be that the remaining failures are due to the unmodelled (misdetection of closely spaced vehicles) or unresolvable (vanishing vehicles problem) components of the transfer gap.

Π	x	y	θ	v	ttc
`wm`	0.4	**-26.8**	-3.1	**-16.6**	**-11.5**
`lagkf`	-0.5	**-24.3**	-0.2	-8.8	**-11.2**
`best`	-0.2	**-22.1**	0.5	-7.4	**-14.6**

Table 4.4: Importance of each feature in RL policies represented as the loss in performance. The lower the value, the more important the feature.

4.5 Policy Analysis

The rule-based policy relies on all the features to make an optimum decision. To find what features the best performing RL policy uses, we calculate the importance of each feature in $\Pi_{\texttt{best}}$ using the method in 3.8.1. For comparison, $\Pi_{\texttt{wm}}$ and $\Pi_{\texttt{lagkf}}$ are also analyzed using the same method. The test set for each consists of 1000 validation episodes generated from the corresponding training environment in WiseMove. The results are shown in Table 4.4.

From the table, it is clear that both $\Pi_{\texttt{lagkf}}$ and $\Pi_{\texttt{best}}$ rely the most on the y feature followed by ttc and v, in that order. In comparison, `wm`, which is trained using perfect information, has more reliance on velocity. However, we expect that the policies would also require the direction of the vehicle, either through x or θ, but it does not use these features at all. This may be because the `ttc` value for vehicles that are not on a collision course is `MAX_FLOAT` and therefore, the agent uses this information instead.

We use the information in Table 4.4 when distilling the behaviour of the policy $\Pi_{\texttt{best}}$. The dataset is generated from Unreal using the method outlined in Section 3.8.2 and a decision tree classifier is trained on it. The splitting criteria used in the decision tree algorithm chooses which features to use for splitting. Instead of using all the features, we allow the decision tree algorithm to split using only the important features from Table 4.4. Also, before training, we manually split the dataset into two using θ (one for each lane) to obtain a more human-interpretable tree.

The extracted decision tree representation of $\Pi_{\texttt{best}}$ is shown in Figure 4.1. The tree provides if-else-if conditions that can be easily translated into code, which allows us to verify its behaviour. Thus, using the conditions and values in the tree, we create a rule-based policy $\Pi_{\texttt{distill}}$ and evaluate it in the Unreal simulator. $\Pi_{\texttt{distill}}$ achieves a `success` of 95% and has a `wait-time` of 20.575 steps.

Distillation using the simple decision tree classifier is clearly not perfect, but it nevertheless offers some insights. From the tree, we can observe that the agent yields for `ttc`

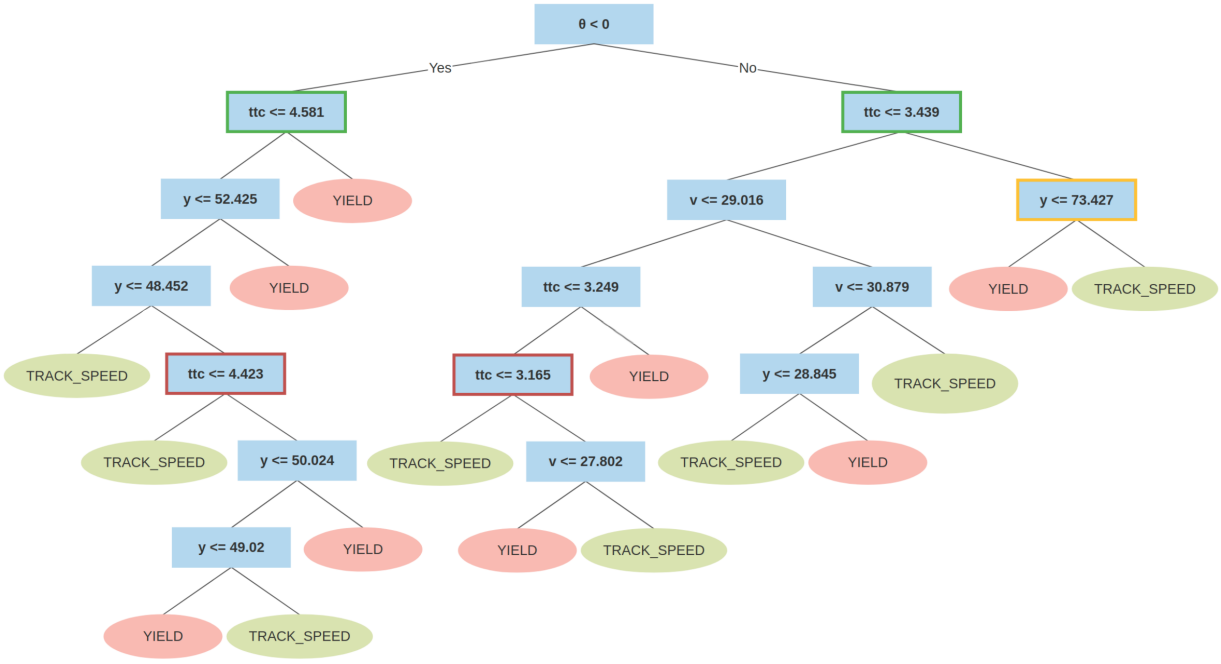

Figure 4.1: Decision tree distilled from best performing RL policy.

values above a certain threshold (outlined in green in the figure). This could be a threshold for dismissing unreliable velocities, since `ttc` is derived from velocity and the agent chooses to wait until the velocity stabilizes. Also, vehicles in the lane closest to ego ($\theta > 0$) with y feature greater than a threshold (outlined in orange in the figure) are considered safe by the policy because they are too far away to collide. Within the reliable `ttc`, the agent chooses to either use v or y to make more nuanced decisions. There also exists a lower threshold for `ttc` (outlined in red in the figure) below which it chooses the `trackspeed` maneuver, since a vehicle near enough to ego is not in danger of collision.

We use the two lower `ttc` thresholds and the safe y distance for the right lane to craft a very defensive rule-based policy $\Pi_{\mathtt{d-ttc}}$, with the rules being:

$$m_k = \begin{cases} \texttt{yield}, & \text{if } (\theta > 0 \text{ and } \mathtt{ttc}_i \geq 3.165 \text{ and } y \leq 73.427) \\ & \text{or } (\theta < 0 \text{ and } \mathtt{ttc}_i \geq 4.423), \text{ for any } i \\ \texttt{trackspeed}, & \text{otherwise} \end{cases}$$

$\Pi_{\mathtt{d-ttc}}$ obtains a `success` of 90.5% in UNREAL, but with a large `wait-time` of 30.5 steps. However, its failures due to collision are only 2.5%. Thus, it does not crash as

often as $\Pi_{\texttt{ttc}}$, but rather waits until the episode times out if it cannot find an opportunity to cross. Comparing with $\Pi_{\texttt{distill}}$, it is evident the decisions taken within the reliable `ttc` value are crucial for not yielding unnecessarily. Hand-designing these nuances with a rule-based approach to efficiently handle perception errors is, therefore, not trivial.

Chapter 5

Conclusion

Due to monetary or hardware constraints, only a low-fidelity simulator may be available for training RL policies. Deploying RL solutions in the real world is already plagued by the sim-to-real transfer learning problem, while using a low-fidelity simulator, which naturally has a larger transfer gap, only makes the problem worse. In this thesis, we examine this problem in the context of autonomous driving for learning an HRL policy for the behaviour planning problem.

We set up a sim-to-sim transfer learning problem for behaviour planning using an existing high-fidelity simulator and develop a low-fidelity simulator specifically for training behaviour planning HRL policies. We analyze the details of the transfer gap between the two and show that even by roughly modelling the observed perception errors that cause the transfer gap, it is possible to create a behaviour planning policy that is better than a traditional rule-based approach. Further, using the error model and domain randomization, we are able to reduce the failures due to perception errors from 10% to 2.75%. In addition to obtaining a good RL policy, we also distill its behaviour and compare it with the rule-based approach to provide insights on their differences.

We hope to study this further and resolve the transfer gap due to ego vehicle modelling differences as well. That would make it possible to use the WiseMove simulator or an equivalent low-fidelity simulator such as SUMO [31] for training behaviour planning policies for the Autonomooose system. This work also adds additional constraints in the driving scenario and the policy to provide a fairer analysis between RL and rule-based policies. A direction for future work could be to remove these constraints to demonstrate this approach in a more generalized setting.

The current policies in this study only depend on the current state. The observation

history and even predicted tracks can be used to generate better decisions. Also, the 'to-go-or-not-to-go' problem formulation is limiting in a practical setting. For example, creeping behaviour is usually exhibited by human drivers, where they slowly move closer to the intersection to reduce the crossing distance. Another direction could be to validate this approach in the real-world with more maneuvers and driving scenarios. The transfer gap between the low-fidelity simulator and a real autonomous vehicle would be larger and may come with additional challenges.

References

[1] Autonomoose. *University of Waterloo*. www.autonomoose.net.

[2] Unreal Engine. www.unrealengine.com/en-US.

[3] Slow drivers cause the most frustration. *Telegraph*, Jul 2011. www.telegraph.co.uk/motoring/news/8649662/Slow-drivers-cause-the-most-frustration.html.

[4] Automated vehicles: Driving innovation in Ontario. *Ministry of Transportation*, Oct 2013. www.mto.gov.on.ca/english/vehicles/automated-vehicles.shtml.

[5] Waymo open dataset: An autonomous driving dataset. www.waymo.com/open, 2019.

[6] *WiseMove: A Framework to Investigate Safe Reinforcement Learning for Autonomous Driving*, Glasgow, Scotland, 2019. Springer.

[7] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 1726–1734. AAAI Press, 2017.

[8] Mayank Bansal, Alex Krizhevsky, and Abhijit Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. In *Proceedings of Robotics: Science and Systems*, FreiburgimBreisgau, Germany, June 2019.

[9] P. Bender, J. Ziegler, and C. Stiller. Lanelets: Efficient map representation for autonomous driving. In *2014 IEEE Intelligent Vehicles Symposium Proceedings*, pages 420–425, June 2014.

[10] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[11] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, S. Levine, and V. Vanhoucke. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4243–4250, May 2018.

[12] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[13] J. Chen, Z. Wang, and M. Tomizuka. Deep hierarchical reinforcement learning for autonomous driving with distinct behaviors. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1239–1244, June 2018.

[14] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[15] E.D. Dickmanns and A. Zapp. Autonomous high speed road vehicle guidance by computer vision1. *IFAC Proceedings Volumes*, 20(5, Part 4):221 – 226, 1987. 10th Triennial IFAC Congress on Automatic Control - 1987 Volume IV, Munich, Germany, 27-31 July.

[16] Aaron Fisher, Cynthia Rudin, and Francesca Dominici. All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously, 2018.

[17] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[18] Abhishek Gupta, Coline Devin, Yuxuan Liu, Pieter Abbeel, and Sergey Levine. Learning invariant feature spaces to transfer skills with reinforcement learning. *ArXiv*, abs/1703.02949, 2017.

[19] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2094–2100. AAAI Press, 2016.

[20] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[21] Hosking, Bryce Antony. Modelling and model predictive control of power-split hybrid powertrains for self-driving vehicles, 2018.

[22] David Isele and Akansel Cosgun. To go or not to go: a case for q-learning at unsignalized intersections. 2017.

[23] Jillianiles. Locals reportedly are frustrated with Alphabet's self-driving cars. *CNBC*, Aug 2018. www.cnbc.com/2018/08/28/locals-reportedly-frustrated-with-alphabets-waymo-self-driving-cars.html.

[24] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, November 2015.

[25] Karel J Keesman. *System identification: an introduction*. Springer Science & Business Media, 2011.

[26] Sangjun Koo, Hwanjo Yu, and Gary Geunbae Lee. Adversarial approach to domain adaptation for reinforcement learning on dialog systems. *Pattern Recognition Letters*, 128:467 – 473, 2019.

[27] Jason Ku, Melissa Mozifian, Jungwook Lee, Ali Harakeh, and Steven Lake Waslander. Joint 3d proposal generation and object detection from view aggregation. *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–8, 2017.

[28] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07).

[29] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. http://arxiv.org/abs/1509.02971, 2015.

[30] S. Liu, J. Tang, Z. Zhang, and J. Gaudiot. Computer architectures for autonomous driving. *Computer*, 50(8):18–25, 2017.

[31] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018.

[32] Antonio Loquercio, Elia Kaufmann, René Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. Deep drone racing: From simulation to reality with domain randomization. *CoRR*, abs/1905.09727, 2019.

[33] Jeff Michels, Ashutosh Saxena, and Andrew Y. Ng. High speed obstacle avoidance using monocular vision and reinforcement learning. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 593–600, New York, NY, USA, 2005. ACM.

[34] Dimitris Milakis, Bart van Arem, and Bert van Wee. Policy and society related implications of automated driving: A review of literature and directions for future research. *Journal of Intelligent Transportation Systems*, 21(4):324–348, 2017.

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[37] Matthias Müller, Alexey Dosovitskiy, Bernard Ghanem, and Vladlen Koltun. Driving policy transfer via modularity and abstraction. *CoRR*, abs/1804.09364, 2018.

[38] Ofir Nachum, Michael Ahn, Hugo Ponte, Shixiang Gu, and Vikash Kumar. Multi-agent manipulation via locomotion using hierarchical sim2real, 2019.

[39] R. Okuda, Y. Kajiwara, and K. Terashima. A survey of technical trend of adas and autonomous driving. In *Proceedings of Technical Program - 2014 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, pages 1–4, April 2014.

[40] OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018.

[41] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub W. Pachocki, Jakub Pachocki, Arthur Petron, Matthias Plappert,

Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *CoRR*, abs/1808.00177, 2018.

[42] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org, 2017.

[43] C. Paxton, V. Raman, G. D. Hager, and M. Kobilarov. Combining neural networks and tree search for task and motion planning in challenging environments. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6059–6066, Sep. 2017.

[44] Terry Pender. Waterloo's 'Autonomoose' hits 100-kilometre milestone. *Waterloo Region Record*, August 2018. www.therecord.com/news-story/8859691-waterloo-s-autonomoose-hits-100-kilometre-milestone.

[45] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. *CoRR*, abs/1710.06537, 2017.

[46] Matthias Plappert. keras-rl. www.github.com/keras-rl/keras-rl, 2016.

[47] F. Poggenhans, J. Pauls, J. Janosovits, S. Orf, M. Naumann, F. Kuhnt, and M. Mayr. Lanelet2: A high-definition map framework for the future of automated driving. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 1672–1679, Nov 2018.

[48] Athanasios S. Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

[49] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[50] R. Queiroz, T. Berger, and K. Czarnecki. Geoscenario: An open dsl for autonomous driving scenario representation. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 287–294, June 2019.

[51] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[52] Fabio Ramos, Rafael Carvalhaes Possas, and Dieter Fox. Bayessim: adaptive domain randomization via probabilistic inference for robotics simulators. *CoRR*, abs/1906.01728, 2019.

[53] Alejandro Rodriguez-Ramos, Carlos Sampedro, Hriday Bavle, Paloma de la Puente, and Pascual Campoy. A deep reinforcement learning strategy for uav autonomous landing on a moving platform. *Journal of Intelligent & Robotic Systems*, 93(1):351–366, 2019.

[54] Andrei A. Rusu, Matej Vecerík, Thomas Rothörl, Nicolas Manfred Otto Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. In *CoRL*, 2016.

[55] AhmadEL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):7076, Jan 2017.

[56] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *CoRR*, abs/1610.03295, 2016.

[57] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[58] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

[59] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181 – 211, 1999.

[60] C. Thorpe, M. H. Hebert, T. Kanade, and S. A. Shafer. Vision and navigation for the carnegie-mellon navlab. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):362–373, May 1988.

[61] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In

2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 23–30, Sep. 2017.

[62] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M. N. Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William Red Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[63] Van Gennip, Matthew. Vehicle dynamic modelling and parameter identification for an autonomous vehicle, 2018.

[64] Yurong You, Xinlei Pan, Ziyan Wang, and Cewu Lu. Virtual to real reinforcement learning for autonomous driving. *CoRR*, abs/1704.03952, 2017.

[65] Fisher Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan, and Trevor Darrell. BDD100K: A diverse driving video database with scalable annotation tooling. *CoRR*, abs/1805.04687, 2018.

[66] J. Ziegler, P. Bender, M. Schreiber, H. Lategahn, T. Strauss, C. Stiller, T. Dang, U. Franke, N. Appenrodt, C. G. Keller, E. Kaus, R. G. Herrtwich, C. Rabe, D. Pfeiffer, F. Lindner, F. Stein, F. Erbs, M. Enzweiler, C. Knppel, J. Hipp, M. Haueis, M. Trepte, C. Brenk, A. Tamke, M. Ghanaat, M. Braun, A. Joos, H. Fritz, H. Mock, M. Hein, and E. Zeeb. Making bertha drivean autonomous journey on a historic route. *IEEE Intelligent Transportation Systems Magazine*, 6(2):8–20, Summer 2014.

[67] N Zimmerman, C Schlenoff, and S Balakirsky. Implementing a rule-based system to represent decision criteria for on-road autonomous navigation. In *2004 AAAI Spring Symp. on Knowledge Representation and Ontologies for Autonomous Systems*, 2004.