Universiteit Leiden
The Netherlands

Data Science and AI

Simulating driving behaviour using Inverse Reinforcement Learning

Enrico Bonsu

Supervisor:
Prof. dr. A. Plaat
Dr. W.A. Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

18/07/2023

Abstract

Currently research on self-driving vehicles is one of the most researched topics in artificial intelligence. Being able to generalize driving behaviour has shown to be a very challenging task. The need to define what "correct" driving behaviour is rests at the core of this challenge. With the increasing number of examples of how correct driving behaviour looks like it seems fitting to apply a technique that would utilize these examples. This is the core idea behind Inverse Reinforcement Learning (IRL). In this thesis demonstrations of correct driving behaviour are used to extract a reward structure that can be applied on every situation possible, even those which have not been observed yet. In this thesis it is proposed that IRL might be able to solve the challenges of creating a self-driving vehicle.

Contents

1 Introduction

Currently many vehicle manufactures are researching the possibility to create a self-driving vehicle. With the increasing number of onboard sensors in modern vehicles the use of artificial intelligence (AI) to solve the problem of self-driving vehicles becomes most appealing. In this thesis a possible solution to the self-driving vehicle problem is proposed using Inverse Reinforcement Learning (IRL). IRL is a technique that uses demonstration data to generalize the "internal reward system" of the expert observed. If it is possible to successfully extract the system the expert agent used during the demonstration, then using that system would make it easy to produce another expert agent. In comparison to other AI techniques this would be less cumbersome. The reason for this is that defining "correct" driving behaviour is difficult, while many examples of correct driving behaviour can be observed daily. It would seem most natural to utilize the demonstrations of correct driving behaviour and extract the system that was presumably used to achieve this correct behaviour. Once the system that was used by the expert agent is available, then it can easily be used to create a vehicle that drives like the expert. Thus this would produce a self-driving vehicle.

1.1 Problem statement

The problem statement of this thesis is:

> Can IRL be used to simulate (correct) driving behaviour by learning from examples?

To operationalize this problem statement the following two research questions are introduced:

1. Can IRL be used to create an agent which is able to cross an intersection correctly if the correct behaviour is observed for that intersection?

2. Can IRL be used to create an agent which is able to cross an intersection correctly if the correct behaviour for that intersection has not been observed?

These research questions help to answer whether or not the IRL technique applied would result in extracting a system that is able to produce good results in new/unobserved scenarios. An intersection is considered crossed correctly if the agent respects the traffic light. This means that the agent only crosses the intersection if the traffic light is green.

1.2 Thesis overview

In Chapter 2 the foundation is provided for Inverse Reinforcement Learning (IRL). In that chapter the algorithm that will be used for the experiments is also provided. In the chapter thereafter, Chapter 3, related work will be discussed. That chapter will mainly look at IRL that was applied on real-world scenarios and IRL used to extract or generate driving behaviour. After that the environment and the setup for the experiment will be provided in Chapter 4, and the results of the experiments in Chapter 5. Based on the results a conclusion will be drawn for the research questions and problem statement in chapter 6. In that chapter limitations of the work will also be mentioned alongside possibilities for further research. This thesis was conducted as a bachelor project at LIACS, supervised by Prof. dr. A. Plaat and Dr. W.A. Kosters.

2 Inverse Reinforcement Learning

This chapter will look at the basis of the Inverse Reinforcement Learning (IRL) problem. First the building blocks that are needed to solve an IRL problem will be discussed. Given next is the IRL algorithm that is used in this thesis. This chapter concludes by briefly mentioning other IRL algorithms that could have been used.

2.1 The foundation of IRL

The foundation of IRL is built on the basis of a concept called Markov Decision Process (MDP). This concept is introduced first. Next the relation between Reinforcement Learning (RL) and IRL is provided before the formal definition of IRL is presented.

2.1.1 Markov Decision Process

The Markov Decision Process (MDP) is named after the Russian mathematician Andrey Andreyevich Markov. The primary subject of his research is known as the Markov chain. The Markov Decision Process is an extension of the Markov chain. The Markov chain concerns a sequence of random events, which corresponds to the states of a certain system [CN06]. Within the Markov chain each event has a probability of occurring as can been seen in Figure 1.
A Markov chain satisfies the Markov property. For a process to satisfy the *Markov property* it demands that the future only depends on the present state and does not depend on the past history [Gra15]. This property allows for predicting the future outcomes by only considering the present state.

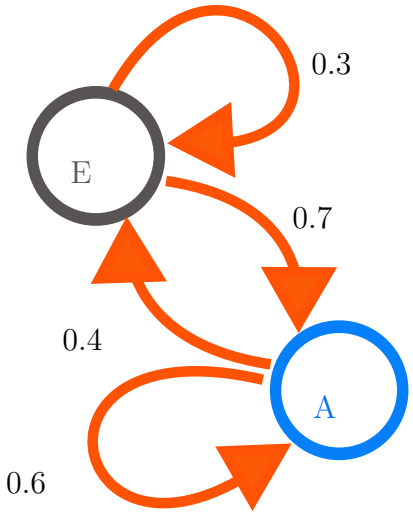

Figure 1: An example of a Markov chain containing two states. Note that every event in the chain has a probability assigned which is the probability of the event occurring from a given state. Source: https://en.wikipedia.org/

As mentioned earlier the MDP extends the Markov chain. By adding actions and rewards to a Markov chain a MDP is created. Adding actions and rewards would allow for choice and motivation

within the process.
There are many variants of the MDP (see [Put94, Gra15]), however in this thesis the introduction of the finite MDP variant is sufficient. A finite MDP can be formally defined as tuple (S, A, T, R, γ) [Lit01]:

- State space (S): A finite set of states of the environment.
- Action space (A): A finite set of actions. When the possible actions depend on a state, then the finite set of actions for state s is denoted as A_s, with $A_s \subseteq A$.
- State transition function (T): A function $T\colon S \times A \times S \to [0, 1]$, giving for each state and action, a probability distribution over states.
- Reward function (R): A function $R\colon S \times A \times S \to \mathbb{R}$ that maps the reward that will be obtained for executing a triplet (s, a, s').
- Discount factor (γ): The discount factor determines the present value of future rewards, with $\gamma \in [0, 1]$.

An example of a finite MDP can be seen in Figure 2. Notice how the reward is paid out only for a complete transition, meaning executing action a in a state s resulting in a transition to state s'. State s and s' may be the same state (i.e., performing action a did not result in a transition to a different state). Since the MDP extends the Markov chain it also inherits the Markov property making it an useful framework used for both RL and IRL.

2.1.2 Reinforcement Learning

Reinforcement Learning (RL) can be considered a third machine learning paradigm, alongside supervised and unsupervised learning [SB18]. As mentioned in the previous Subsection 2.1.1 the framework used in RL is that of a MDP. In RL a learner (often called the agent) interacts with the environment. These interactions are used to estimate how rewarding it is to be in an arbitrary state s. To quantify how rewarding it is to be in a state it is necessary to know the rules on which the agent acts within the environment. This allows for the possibility to predict the future rewards obtained by the agent. The rules on which the agent decides which action to take (in every state) is called a policy function π. The policy function π provides the probability of picking action a in state s for every $s \in S$ and every $a \in A$.
As a result of the dynamics of the environment being unknown to the agent, the agent generates a state-value function $V_\pi(s)$ that is used to calculate the expected return when starting in state s and following a policy π thereafter. To formally define the state-value function V_π additional notations need to be introduced:

- Trajectory (τ): A sequence $\tau = ((s_0, a_0), (s_1, a_1), \ldots, (s_n, a_n))$ of state-action pairs (with $n \in \mathbb{N}$).
- Expected return (G_t): A sequence $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{n=0}^{\infty} \gamma^n r_{t+n+1}$ of (expected) rewards received starting from discrete time step t, with discount factor γ.

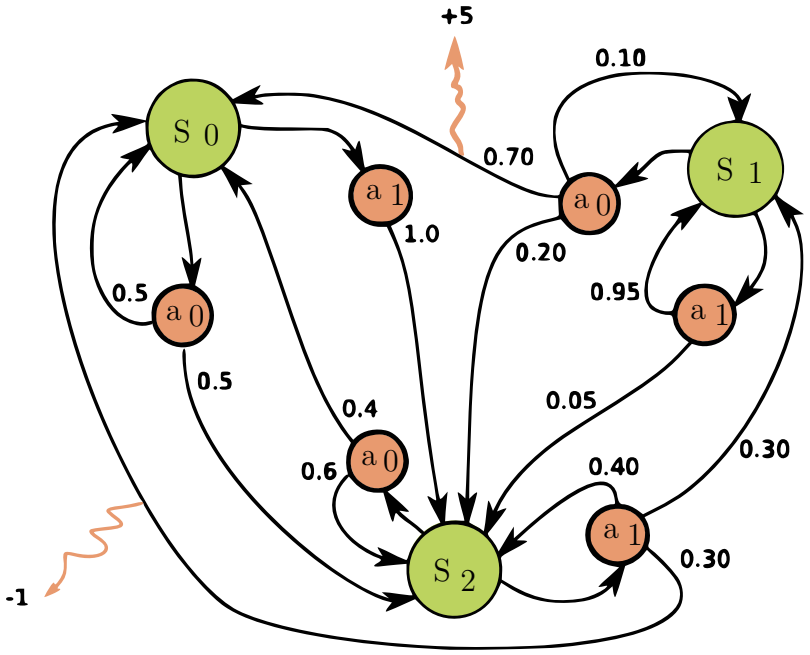

Figure 2: An example of a Markov Decision Process containing three states. Notice how the MDP extends the Markov chain by adding rewards and actions. Each state-action pair has a transition probability and a reward assigned. State-action pairs that do not have an explicit reward mentioned have a reward of 0. Source: https://en.wikipedia.org/.

With this additional information the definition of the state-value function V_π can be introduced.

Definition 1 (State-value function V_π [SB18])**.** Let $s \in S$ and given a policy function π, then the expected value of state s can be calculated using the state-value function V_π:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\right],$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected return G_t given that the agent follows policy π, and t is any discrete time step.

Another function that is very similar to the state-value function is called the action-value function Q_π. The only difference between state-value function and action-value function is that the action-value function also requires an initial action that will be taken in the starting state. This provides the ability to evaluate an individual action for a state. The definition of the action-value function is the following:

Definition 2 (Action-value function Q_π [SB18])**.** Let $s \in S$, $a \in A_s \subseteq A$ and given a policy function π, then the expected value of taking action a in state s can be calculated using the action-value function Q_π:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\right],$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected return G_t given that the agent follows policy π, and t is any discrete time step.

Both the state-value function V_π and the action-value function Q_π can be estimated from experience. The typical way experience is gained is visualized in Figure 3.

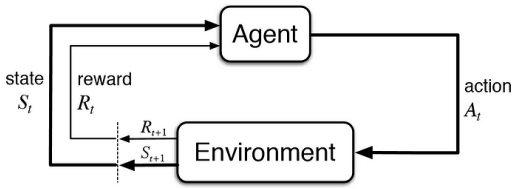

Figure 3: A typical example of how the interaction process between the agent and the environment looks like. Source: https://towardsdatascience.com/.

Note that the state-value function and action-value function can be written to show the relationship between the current state s and its successor states:

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \Big[\sum_{s' \in S} T(s, a, s') \big[R(s, a, s') + \gamma \cdot V_\pi(s') \big] \Big]$$

$$Q_\pi(s, a) = \sum_{s' \in S} T(s, a, s') \Big[R(s, a, s') + \gamma \cdot \sum_{a' \in A} \pi(a'|s) \big[Q_\pi(s', a') \big] \Big]$$

These versions of equations are known as the *Bellman equations* for the state-value function and action-value function.
The goal in RL is often to maximize the reward the agent can receive. The policy function that satisfies the goal is called the optimal policy π_*. A policy π_* is considered optimal when for every other policy π it holds that $\pi_* \geq \pi$ under the expected return. This means essentially that $\pi_* \geq \pi$ if and only if $V_{\pi*}(s) \geq V_\pi(s)$ for all $s \in S$. $V_{\pi*}$ is called the optimal state-value function and is defined as:

Definition 3 (Optimal state-value function $V_{\pi*}$ [SB18])**.** Let $V_\pi(s)$ be a state-value function under policy function π in the state s, then the optimal state-value function is defined as:

$$V_{\pi*}(s) = \max_\pi V_\pi(s)$$

for every $s \in S$.

Just like the optimal state-value function there also exists an optimal action-value function $Q_{\pi*}$ and this is defined as:

Definition 4 (Optimal action-value function $Q_{\pi*}$ [SB18])**.** Let $Q_\pi(s, a)$ be a action-value function under policy function π in the state s performing action a, then the optimal action-value function is defined as:

$$Q_{\pi*}(s, a) = \max_\pi Q_\pi(s, a)$$

for every $s \in S$ and $a \in A_s$.

While multiple optimal policy π_* can exist, the optimal state-value function and optimal action-value function are unique for a given MDP. It is also possible to write the optimal state-value function and optimal action-value function as a Bellman equation:

$$V_{\pi*}(s) = \max_a \sum_{s' \in S} T(s, a, s') \big[R(s, a, s') + \gamma \cdot V_{\pi*}(s') \big]$$

$$Q_{\pi*}(s, a) = \sum_{s' \in S} T(s, a, s') \Big[R(s, a, s') + \gamma \cdot \max_{a'} \big[Q_{\pi*}(s', a') \big] \Big]$$

Solving the optimal state-value function or optimal action-value function is one way to find an optimal policy, however this method is not often used in RL. This is due to the fact that solving these equations requires an exhaustive search. In general to solve the equations there are at least three assumptions applied [SB18]:

1. The dynamics of the environment are accurately known.
2. Computational resources are sufficient to complete the calculation.
3. The states have the Markov property.

Since these assumptions are often not completely met, it is typical in RL to settle for an approximate solution.
Two methods used in RL to find an approximate solution are called model-based RL and model-free RL. In model-based RL the optimal behavior can be found by learning the dynamics of the environment and observing the outcome for interacting with the environment. This method is based on planning. As one can assume this method is computationally expensive. The model-free RL method is computationally more efficient. The model-free RL method does not try to learn the dynamics of the environment. In this method sampling is used to compute the expected return for the state-action pairs observed in the samples. It is possible to combine these methods which can improve performance depending on the task at hand. An example of an algorithm that have combined both methods is the Dyna-Q algorithm [SB18].

2.1.3 Formal definition of IRL

The goal of Reinforcement Learning (RL) is to find an optimal policy. Following the optimal policy would result in the maximum reward obtainable by an agent. Often the reward function is manually specified. The problem that can arise when manually specifying a reward function is that the behaviour that is produced by the optimal policy does not match the expected behaviour of those who have provided the reward function. The reason for this is that the reward function indicates what the agent wants to accomplish, not how the agent needs to behave in order to accomplish it. For complex tasks it can be hard to specify a reward function because it requires an accurate balance between all the different variables [Rus98].
In some cases the complex task that needs to be solved is already being solved by animals or other organisms. Whether the task is throwing a ball, balancing objects in the air or driving a vehicle,

often it can be observed how to act in order to solve these tasks. If there is some way to extract the reward function of the agents that are already solving these tasks, then this reward function can be used. Extracting the reward function from agents that are already solving the task is the goal of Inverse Reinforcement Learning (IRL).
IRL requires either observations of the task that is being solved or a policy from which it can produce these observations. In IRL it is typically assumed that the observations are produced by an expert agent E for which it is assumed that its behaviour is optimal (however there are IRL methods proposed to extract the reward function for sub-optimal behaviour produced by an expert agent [BTR09]). The other assumption is that the environment of the problem can be defined as a Markov Decision Process (MDP). With the assumptions and requirements given for an IRL agent it is possible to present the formal definition of IRL:

Definition 5 (Inverse Reinforcement Learning (IRL) [AD21])**.** Let an MDP without reward function, $\mathcal{M} \backslash_{R_E}$, model the interaction of the expert agent E with the environment. Let $\mathcal{D} = \{\langle (s_0, a_0), (s_1, a_1), \ldots, (s_j, a_j)\rangle_1, \ldots, \langle (s_0, a_0), (s_1, a_1), \ldots, (s_j, a_j)\rangle_{i=2}^{N}\}$, $s_j \in S, a_j \in A$, and $i, j, N \in \mathbb{N}$ be the set of demonstrated trajectories. A trajectory in $\mathcal{D}$ is denoted as τ. We may assume that all $\tau \in \mathcal{D}$ are perfectly observed. Then, determine $\hat{R}_E$ that best explains either policy π_E if given or the observed behaviour in the form of demonstrated trajectories.

Basically in IRL the assumption is that the behaviour produced by the expert agent E reflects the inner preference of the agent. Intuitively this makes sense. The behaviour an agent produces is in accordance with some goal the agent tries to achieve. The relation between RL and IRL can be seen in Figure 4.

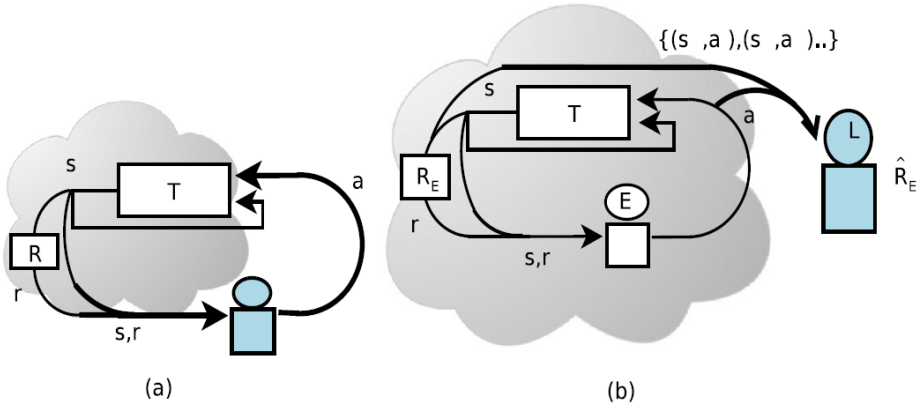

Figure 4: A visual representation of the difference between RL and IRL. In (a) the typical process of an RL can be seen (similar to Figure 3). The typical process of an IRL agent can be seen in (b). From the observation of the behaviour of an expert agent E the IRL agent tries to estimate the expert's reward function $\hat{R}_E$ Source: [AD21].

There are typically two use cases for IRL. The first use case is to use the estimated reward function to create a new agent with the expert's preferences. This would result in expert behaviour within the environment observed, but would likely also produce (very) good behaviour in environments that differ from the observed environment of the expert agent. The second use case uses the estimated

reward function to predict the behaviour of other agent(s) in the environment. An example in which this can be useful is that of merging onto a highway. If the agent that wants to merge has knowledge of the preferences of the other agents, it would be easy to predict the behaviour of the other agents and allows the merging agent to easily adapt its behaviour to match its goal (merging onto the highway). In the next section the IRL algorithm used in this thesis will be introduced.

2.2 Maximum Entropy Inverse Reinforcement Learning

In this thesis the Maximum Entropy Inverse Reinforcement Learning (MAXENTIRL) algorithm is used for the experiments. This algorithm has been introduced by Ziebart et al. [ZMBD08] in 2008. In order to understand this algorithm it is necessary to understand the principle of maximum entropy. This is introduced first. In order to generate an estimated reward function $\hat{R}_E$ it is necessary to assume a reward function structure. This is provided thereafter. At the end of this section a summary of the MAXENTIRL algorithm is given.

2.2.1 The principle of maximum entropy

Entropy is a concept found in many fields of science, however in this thesis entropy refers to the concept in information theory. In information theory, entropy, also known as Shannon Entropy, is a concept first introduced by Claude Shannon in 1948 [Sha48]. Entropy can be described as a way of measuring the uncertainty of a probability distribution. When the entropy is small, an outcome of the probability distribution is quite easily determined. If the entropy for a probability distribution is large, then an outcome is hard to predict. A different way of viewing entropy for discrete events (as derived from information theory) is to consider the minimum number of YES/NO questions that are required in order to determine which event occurred from a probability distribution. In order to understand how this would look in practice it is useful to consider the formula for entropy $H(X)$:

$$H(X) = \sum_{x \in X} p(x) \log_2\Big(\frac{1}{p(x)}\Big) = -\sum_{x \in X} p(x) \log_2(p(x))$$

with $p\colon X \to [0, 1]$, which provides the probability of an event x occurring from the set of events X. With the formula just introduced, an intuition for entropy can be built by looking at the entropy value for three different scenarios when tossing a coin twice:

- **Scenario 1:** Both sides of the coin contains head
  This would produce the following probabilities for each possible outcome:

Event	Probability of observing
HEAD, HEAD	$1.0 \times 1.0 = 1.0$
HEAD, TAIL	$1.0 \times 0.0 = 0.0$
TAIL, HEAD	$0.0 \times 1.0 = 0.0$
TAIL, TAIL	$0.0 \times 0.0 = 0.0$

  The following entropy would be the result for this probability distribution[1]:

$$H(X) = -1 \log_2(1) = 0$$

  The resulting entropy value of 0 makes sense. We do not need to ask any questions in order to determine which event occurred, because we know that the outcome will always be [HEAD, HEAD].

[1] A property of Shannon entropy is that adding or removing an event with probability zero does not contribute to the entropy. Thus in this example the other three events can be ignored.

- **Scenario 2:** The coin is fair (equal probability for landing head or tail)
  This would produce the following probabilities for each possible outcome:

Event	Probability of observing
HEAD, HEAD	$0.50 \times 0.50 = 0.25$
HEAD, TAIL	$0.50 \times 0.50 = 0.25$
TAIL, HEAD	$0.50 \times 0.50 = 0.25$
TAIL, TAIL	$0.50 \times 0.50 = 0.25$

  The following entropy would be the result for this probability distribution

$$H(X) = -(0.25 \log_2(0.25) + 0.25 \log_2(0.25) + 0.25 \log_2(0.25) + 0.25 \log_2(0.25)) = 2$$

  Again, this resulting entropy value of 2 makes sense. In order to determine which event occurred it is necessary to "ask" at minimum two questions. The first question could be "was the first toss head?", then the second question could be "was the second toss head?". Based on the YES/NO answers for these two questions one can easily determine which event occurred.

- **Scenario 3:** The coin has a 0.8 probability of landing head
  This would produce the following probabilities for each possible outcome:

Event	Probability of observing
HEAD, HEAD	$0.80 \times 0.80 = 0.64$
HEAD, TAIL	$0.80 \times 0.20 = 0.16$
TAIL, HEAD	$0.20 \times 0.80 = 0.16$
TAIL, TAIL	$0.20 \times 0.20 = 0.04$

  The following entropy would be the result for this probability distribution

$$H(X) = -(0.64 \log_2(0.64) + 0.16 \log_2(0.16) + 0.16 \log_2(0.16) + 0.04 \log_2(0.04)) \approx 1.44$$

  This scenario requires more reasoning. Intuitively, when comparing this scenario with the second scenario it is obvious that with knowledge of the fact that this coin is way more likely to land head, that we can "ask" better questions. One obvious question one should always ask is if the event contains two heads. With a probability of 0.64 more often than not this will be true, resulting in only using one question to determine the occurred event. Entropy states the minimum number of "questions" that are required (on average) in order to determine which event occurred. Using the formula it can be proven that at least 1.44 "questions" are required on average.

From these three scenarios it can be observed that the largest entropy will be produced when the probability of each event occurring is equally likely. This is also the scenario that describes the system for which no additional information is known. The only information on which we base our probability distribution is of knowledge that the coin toss has two possible outcomes and the coin will be tossed twice. No additional assumption or biases are used to generate that probability distribution. This idea is at the core of the principle of maximum entropy. The principle of maximum

entropy has been introduced first by Edwin Thompson Jaynes in 1957 [Jay57]. Edwin Thompson Jaynes argued that apart from the initial assumptions no other assumptions should be made in order to generate the probability distribution. Another way to phrase this is that apart from the initial certainty the remaining uncertainty should be as large as possible. The principle of maximum entropy can be used to generate the probability distribution for a system that complies with the initial assumptions and none else. These assumptions are called the constraints (of the system). Two constraints that apply to any system are inherited from the probability mass function:

1. The sum over all events sums up to 1: $\sum_{x \in X} p(x) = 1$.
2. The probability for every event is non-negative: $p(x) \geq 0$.

To understand why this makes sense let us look at a six-sided die. A balanced die would have equal probability of landing on each of its sides: ($p(x) = \frac{1}{6}$ for $x = 1, \ldots, 6$). It is possible to calculate the average outcome for one die roll:

$$\sum_{x=1}^{6} x \cdot p(x) = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5$$

Now for a balanced die the average outcome is 3.5. Let us now say we received another six-sided die for which it is known that the average outcome is 4. To find a probability distribution the following equation must be solved:

$$\sum_{x=1}^{6} x \cdot p(x) = 1 \cdot p(1) + 2 \cdot p(2) + 3 \cdot p(3) + 4 \cdot p(4) + 5 \cdot p(5) + 6 \cdot p(6) = 4.0$$

Many probability distributions would satisfy this equation. Three probability distributions that would satisfy the equation are given in Table 1. While these distributions would satisfy the constraint, it is easy to observe that all three are a product of a probability distribution with additional assumptions. All three distributions assume that the die would never land on certain sides. With the initial assumption (the average outcome is 4.0) such additional assumptions could not be derived, and thus none of the three distributions would be generated by the principle of maximum entropy.

2.2.2 Reward function definition

In order to approximate the reward function R of the expert agent E the MAXENTIRL algorithm makes two assumptions [ZMBD08]. The first assumption is that every state s can be mapped to the features of that state. The function that would map this relation can be expressed as:

$$\boldsymbol{\phi} \colon S \to \mathbb{R}^k$$

This ensures that for every state s a value for each of the k state features will be produced.
The second assumption is that the reward function can be expressed as a linear sum of weighted features. For a given feature weight vector $\boldsymbol{w}$ the reward function is defined as:

$$R(s) = w_1\phi(s)_1 + w_2\phi(s)_2 + \ldots + w_k\phi(s)_k = \boldsymbol{w}^\top \boldsymbol{\phi}(s) \tag{1}$$

where k is the number of features extracted from the states.

Event (die side)	P distribution 1	P distribution 2	P distribution 3
1	0.0	0.0	0.0
2	0.5	0.0	0.0
3	0.0	0.0	0.5
4	0.0	1.0	0.0
5	0.0	0.0	0.5
6	0.5	0.0	0.0
Average outcome	**4.0**	**4.0**	**4.0**

Table 1: In this table three probability distributions are provided for a six-sided die. All three of these distributions would average an outcome of 4.0.

2.2.3 The algorithm

In this subsection the MAXENTIRL algorithm, as provided in [ZMBD08], is summarized. In MAXENTIRL the number of features that will be extracted from each state is manually specified. The goal in MAXENTIRL is to find the appropriate feature weight vector $\boldsymbol{w}$ that would produce the observed demonstration $\mathcal{D}$. MAXENTIRL employs the principle of maximum entropy to find the feature weight vector $\boldsymbol{w}$ that does not exhibit any additional preferences beyond matching feature expectations.
To accomplish this goal the MAXENTIRL algorithm makes use of the trajectory feature count $\boldsymbol{\phi}(\tau)$. The trajectory feature count $\boldsymbol{\phi}(\tau)$ is generated by summing the features observed for each state visited in the trajectory τ and can be expressed as:

$$\boldsymbol{\phi}(\tau) = \sum_{s \in \tau} \boldsymbol{\phi}(s)$$

This concept can also be extended to any demonstration $\mathcal{D}$ performed by the expert agent E. Let $|\mathcal{D}|$ be the number of trajectories for a given demonstration $\mathcal{D}$, then the expected empirical feature count is:

$$\boldsymbol{\phi}(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \boldsymbol{\phi}(\tau)$$

Abbeel et al. [AN04] demonstrated that matching the expected empirical feature count $\boldsymbol{\phi}(\mathcal{D})$ is both necessary and sufficient for a learning agent to achieve the same performance as the expert agent E if the expert agent were in fact solving an MDP with a reward function linear in those features. With this fact the problem of ambiguity arises. Many reward functions and many policies lead to the same feature counts. To resolve the problem of this ambiguity the principle of maximum entropy is applied. For each reward function that would satisfy the matching of the expected empirical feature count $\boldsymbol{\phi}(\mathcal{D})$ its entropy can be calculated. The reward function with the largest entropy would be the reward function with no other preference but the preference of matching the expected empirical feature count $\boldsymbol{\phi}(\mathcal{D})$.
In the principle of maximum entropy the goal is to find the probability distribution p that yields the highest entropy. Within the context of MAXENTIRL the probability distribution is subject to two constraints. The first constraint is that the feature expectation for both the expert agent and the learner agent must match. The second constraint is that under the found probability distribution

the demonstration $\mathcal{D}$ must be observed. This constrained optimization problem can also be formally described as:

$$\begin{aligned} \arg\max_{p} \quad & H(p) \\ \text{subject to} \quad & \mathbb{E}_{\pi_{IRL}}[\boldsymbol{\phi}(\mathcal{D})] = \mathbb{E}_{\pi_E}[\boldsymbol{\phi}(\mathcal{D})] && \text{(feature-expectation matching)} \\ & \sum_{\tau} p(\tau) = 1, \forall \tau \colon p(\tau) > 0 && \text{(probability constraints)} \end{aligned}$$

The technique to solve this is a well-known technique which makes use of the method of Lagrange multipliers. For MDPs with deterministic transition dynamics the parameterized probability distribution can be expressed as:

$$p_d(\tau|\boldsymbol{w}) = \frac{1}{Z(\boldsymbol{w})}\exp(\boldsymbol{w}^\top \boldsymbol{\phi}(\tau)), \qquad Z(\boldsymbol{w}) = \sum_{\tau} \exp(\boldsymbol{w}^\top \boldsymbol{\phi}(\tau))$$

while for MDPs with stochastic transitions the parameterized probability distribution is approximated by multiplying the deterministic solution by the transition probability of the trajectory:

$$p_s(\tau|\boldsymbol{w}) \approx \frac{1}{Z(\boldsymbol{w})}\exp(\boldsymbol{w}^\top \boldsymbol{\phi}(\tau)) \prod_{(s_{t+1},a_t,s)\in\tau} p(s_{t+1}|s_t, a_t)$$

Optimizing the parameterized probability distribution can be achieved by maximizing the log-likelihood of observing the demonstration $\mathcal{D}$ under the probability distribution using gradient-based optimization methods:

$$\boldsymbol{w}_* = \arg\max_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \arg\max_{\boldsymbol{w}} \sum_{\tau\in\mathcal{D}} \log\, p(\tau|\boldsymbol{w})$$

The gradient can then be expressed as the difference between the expected empirical feature counts and the learner's expected feature counts under the feature weight vector $\boldsymbol{w}$:

$$\nabla\mathcal{L}(\boldsymbol{w}) = \mathbb{E}_{\pi^E}[\boldsymbol{\phi}(\mathcal{D})] - \sum_{\tau\in\mathcal{D}} p(\tau|\boldsymbol{w})\boldsymbol{\phi}(\tau)$$

The learner's expected feature counts under the feature weight vector $\boldsymbol{w}$ can also be defined in terms of expected state visitation frequencies D_{s_i}:

$$\sum_{\tau\in\mathcal{D}} p(\tau|\boldsymbol{w})\boldsymbol{\phi}(\tau) = \sum_{s_i\in S} D_{s_i}\boldsymbol{\phi}(s_i)$$

Calculating the expected state visitation frequency requires the policy function that is used by the learner agent. In order to obtain this policy function a so-called backward pass is performed. The backward pass recursively "backup" from terminal states and computes the state and state-action partition functions Z_{s_i} and Z_{s_i,a_j} which are defined as:

$$Z_{s_i,a_j} = \sum_k p(s_k|s_i, a_j)\, \exp(R(s_i))Z_{s_k} = \sum_k p(s_k|s_i, a_j)\, \exp(\boldsymbol{w}^\top \boldsymbol{\phi}(s_i))Z_{s_k}$$

$$Z_{s_i} = \sum_{a_j} Z_{s_i,a_j}$$

The state partition function for the terminal state $s_{terminal}$ is set to 1 ($Z_{s_{terminal}} = 1$) and recursively compute Z_{s_i} and Z_{s_i,a_j} for N iterations. Afterwards the local action probability (i.e., the policy function) can be computed for any state:

$$\pi(a_j|s_i, \boldsymbol{w}) = \frac{Z_{s_i,a_j}}{Z_{s_i}}$$

With these local action probabilities a so-called forward pass can be performed. The forward pass starts by setting the probability for the initial starting state(s), and then iterative (starting from $t = 1$ to N) computing the state visited next:

$$D_{s_k,t+1} = \sum_{s_i \in S} \sum_{a_j \in A} D_{s_i,t} \pi(a_j|s_i, \boldsymbol{w}) p(s_k|s_i, a_j)$$

Afterwards the state-visitation frequency for an individual state s_i can be obtained by taking the sum of its occurrence during the forward pass:

$$D_{s_i} = \sum_t D_{s_i,t}$$

By use of the backward pass and forward pass the expected state visitation frequency under the learner's policy can be obtained which in turn can be used to calculate the gradient.

2.3 Other Inverse Reinforcement Learning algorithms

In Section 2.2 a summary of the MAXENTIRL algorithm has been provided. This algorithm solves the IRL problem by assuming that the reward function can be expressed as a linear sum of weighted features and optimizes the feature weights by using entropy optimization. In general there are four approaches to solve the IRL problem, and entropy optimization is one of them. The other three are margin optimization, Bayesian update, and classification and regression [AD21]. Margin optimization assumes, just like entropy optimization, that the reward function can be expressed as a linear sum of weighted features. The IRL problem is solved by finding the solution that maximizes some margin. The two most known margin optimization algorithms applied to the IRL problem are MAX-MARGIN and PROJECTION [AN04].

While a structure of the reward function for both the margin optimization and entropy optimization is assumed, the Bayesian update approach does not impose a structure. This approach uses all observed state-action pairs to update a prior distribution over the possible reward functions. From this the likelihood of observing each of the trajectories in the demonstration under the estimated reward function $\hat{R}_E$ can be calculated.

The last approach that can be used to solve the IRL problem is that which uses classification and regression. However, this approach has not be as popular as the approaches mentioned above. To use this approach each state-action pair is seen as a data label pair, with the action being the label. From here a regression model or classification model can be trained. When the state space and/or action space is large a regression tree may be preferred in which each path of the tree captures a region of the state and action space, while the whole tree itself captures all regions. An example of this approach is that of FIRL [LPK10].

3 Related Work

In this chapter related work of IRL applied on real-world problems is provided. Afterwards related work which used IRL to create a self-driving agent will be looked at.

3.1 Real world application of IRL

While IRL is a relatively young concept compared to RL, many interesting results have been obtained by deploying IRL on complex real-world problems. An example of a complex problem is predicting goal-directed human attention when searching visually. Obtaining the strategy employed by a human when using vision to search for something can significantly improve computer vision. In the paper by Yang et al. [YHC+20] they compared the results of their IRL algorithm to five other predicting search scanpaths algorithms. For training the IRL model they used the COCO-Search18 dataset for which the search fixations for searching 18 target-object categories (produced by 10 humans) are available. In their paper they showed that their IRL algorithm was able to outperform all other algorithms. In addition to this improvement their solution also produced state representations that are more explainable than their counterparts. For self-driving vehicles applying such an algorithm could improve the driving behaviour to more closely match that of a human driver. Applying a model that matches the visual search of a human could improve the current self-driving capabilities of those vehicles.
One of the domains in which IRL can show the most progress is that of robotics. Robotics is often used to automate tasks that are currently done by manual labor. By observing the behaviour of those workers IRL can adapt to perform those tasks relatively quickly. In the paper by Kumar et al. [KZH+23] they showed that by using third-person videos of a task being solved a IRL model can be created that outperforms four other techniques. The main difference between their IRL approach and the other approaches is that they trained their model on a diverse set of videos, instead of a relatively restricted domain of videos. This approach can be very useful for implementing self-driving behaviour in vehicles, since they need to act in many different driving situations. Also the large amount of recorded videos of third-person driving behaviour can be utilized by this technique.

3.1.1 Self-driving using IRL

With the increase of sensory data produced by cars applying IRL to create a self-driving vehicle has become very appealing to researchers. In the research paper by Zou et al. [ZLZ18] an experiment was performed in the simulation software package called Udacity. The approach used in their experiment is similar to the approach used in the experiments for this thesis, however their self-driving behaviour was more complex. They used a convolution neural network to extract features from the driving state and added the steering angle, throttle value, brake value and current speed as additional features observable for each state. From there they applied the IRL algorithm called maximum margin planning. The actions the agents can perform were also more complex. At any state one of following five actions could be performed: drive straight, turn slightly left, turn slightly right, turn hard right or turn hard left. In their experiment they tested the performance of their IRL agent on a curved road. They showed that their IRL approach outperforms the end-to-end learning approach (for autonomous driving) proposed by Bojarski et al. [BTD+16].
While the previous experiment was performed in simulation software, real-world experiments have

also been conducted. One very promising paper by Phan-Minh et al. [PMHC+23] used IRL to train an agent and validated the model in heavy traffic in Las Vegas. It uses their planner (called DriveIRL) which proposes a diverse set of trajectories for which, after applying a safety filter, the MaxentIRL algorithm was used to score the remaining trajectories. They trained their IRL agent on approximately 556 hours of real-world expert data. Their extensive approach of applying IRL to create a real-world self-driving vehicle was able to drive autonomously for 6.9 miles (of a 8.5 miles route) when applying safety filtering and 8.8 miles (of a 11 miles route) when they did not apply the safety filtering. When they applied the safety filtering the only reason for intervening was due to mandatory takeover regions. This paper shows the true potential for applying IRL for creating autonomous vehicles.

4 Experiments

In this chapter the environment in which the experiments will be performed will be described. After that the specific configuration used to run the experiments will be provided. The code and additional instructions on how to acquire the results can be found on Github[2]. Links to video demonstrations will also be available in that repository.

4.1 The environment: CARLA

The environment in which the experiment will be performed is called Carla[3] [DRC+17]. Carla is an open-source simulator for autonomous driving research. Carla is implemented in the game engine Unreal Engine 4. The version of Carla used during the experiment is version 0.9.14. In Carla the behaviour of all the actors (e.g., cars, pedestrians or traffic signals) can be manually specified. This allows for simulating specific traffic scenarios. It is also possible to equip a vehicle with multiple sensors like LIDAR, RGB camera or depth sensors. The physical properties of a vehicle can also be specified. The friction of each individual tire, the brake torque and maximum steering angle are examples of configurable properties of a vehicle in Carla. While many maps are available in the simulation, the map used during the experiments can be seen in Figure 5.

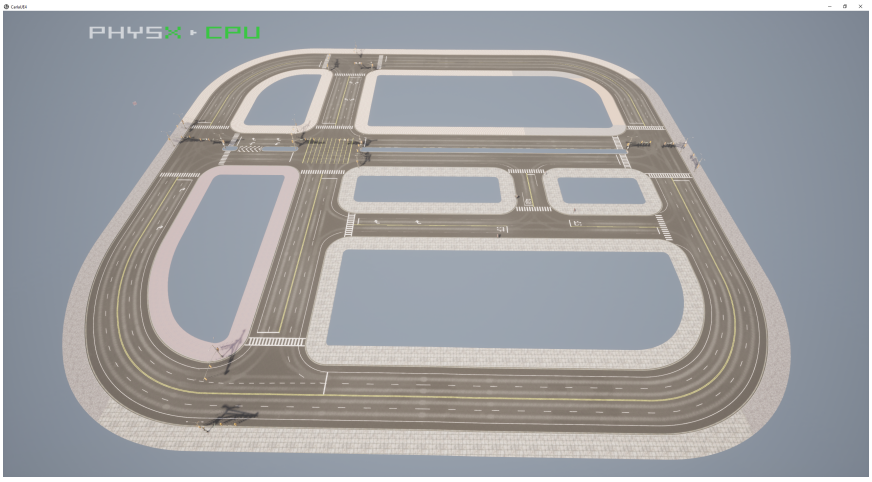

Figure 5: The Carla map (Town10HD_Opt) which is the map on which the experiments are run. During the experiments only the necessary layers are rendered.

4.2 The experiments

The experiments that will be looked at consist of two different scenarios. In the first scenario an expert agent perform a demonstration of passing an intersection. The behaviour that is of interest is that of the interaction between the traffic light and the expert agent. Using the Maxentirl algorithm on the demonstration data a reward function R can be created (see Equation 1). This reward function is then used to create another agent. The performance of the newly created agent

[2]https://github.com/enricobonsu/BachelorThesis
[3]https://carla.org/

is then observed on the same interaction. The trajectory is shown in Figure 6.
The second scenario is largely the same as the first scenario, however now the performance of the agent is observed on an unobserved intersection (see Figure 8). This scenario is used to observe if the behaviour generated by the estimated reward function $\hat{R}_E$ can be applied to new situations.

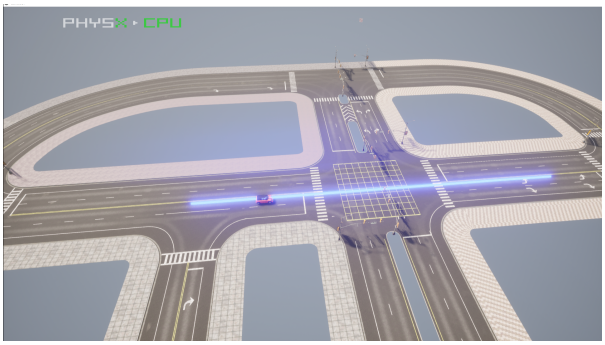

Figure 6: The trajectory that will be followed by the expert agent, starting from the left and following the blue line (scenario 1).

Figure 7: A frame in which the expert agent stops in front of the traffic light, because the light is red.

4.3 The Setup for the Experiments

In order to run the experiments the following steps need to be performed. First an expert agent needs to be created. Once the expert agent is created its behaviour with the environment needs to be observed. This would generate the demonstration data which will be filtered to extract the features from each state. The next part is to obtain the state transition function T. This is obtained by running a custom script. From here it is possible to run the MAXENTIRL algorithm. This will generate the feature weight vector $\boldsymbol{w}$. To observe how an agent would perform using a reward function R based on this feature weight vector $\boldsymbol{w}$ (see Equation 1) a new agent is created with such a reward function R which would act to maximize its reward when interacting with the environment.

4.3.1 Generating optimal behaviour

In CARLA scripts are available that would generate correct driving behaviour. For this CARLA uses two PID-controllers to determine the steering angle and the (de)acceleration needed for the vehicle advance towards its goal. The default script is modified to generate simpler behaviour. The following configuration is set for the expert agent and the agent which uses the estimated reward function $\hat{R}_E$:

- In the environment only one agent exists.
- An action is taken every 0.12 seconds (in the simulated time).
- Only one type of vehicle is used.

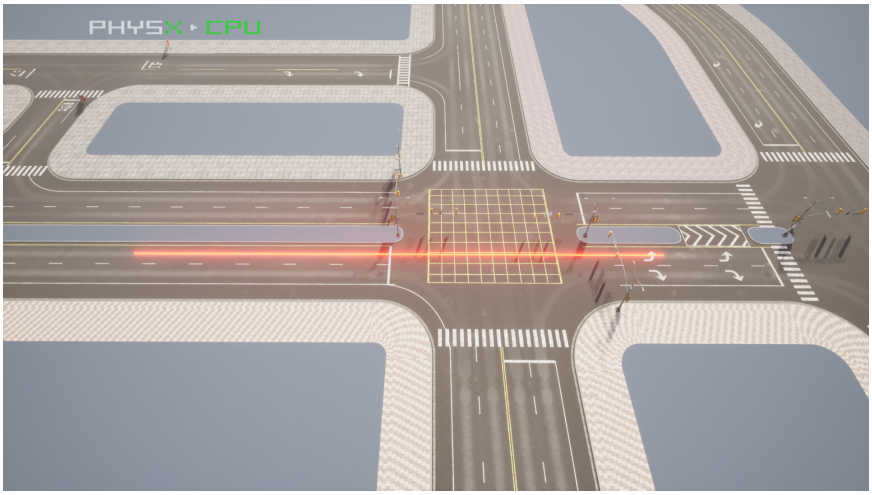

Figure 8: The unobserved trajectory (red line) on which the IRL agent will drive. The agent starts from the left and follows the red line (scenario 2).

- The friction of each tire is set to 0.0 on the vehicle. This allows for consistent state transitions.
- The expert agent moves at a constant velocity of 30.0 km/h.
- The two available actions are either to brake or not.
- Braking would result in an instant stop (velocity 0.0 km/h).
- Performing an action would instantly set the appropriate velocity.
- The start location and destination are fixed for the trajectory performed by expert agent.

4.3.2 Gathering demonstration data

In this thesis the main research application of MAXENTIRL is that of a self-driving vehicle. Because of that the observed data is gathered from the ego of the expert agent's vehicle. The two features that are observed at each step are:

1. Whether or not the vehicle observes a traffic light which is red. This observed value is either 0 or 1.
2. The remaining distance from its destination. This value is generated by dividing the remaining distance towards the agent's destination to the distance between the starting location and the destination location. The value is then rounded to its third decimal place. This would result in a value between 0 and 1 (with 0 and 1 included).

Additionally the action taken in each state is also observed.
In total 200 trajectories were observed, but the first observed trajectory is dropped from the demonstration data due to inconsistencies. This would result in the demonstration consisting of 199 trajectories. In 98 of these 199 trajectories the traffic light was observed red when the expert agent was within distance of the traffic light.

4.3.3 Generation the state transition function

While some state transitions are observed from the demonstration, many transitions likely will never be performed by the expert agent. To obtain the state transition function T a custom script is created to observe the execution of each possible state-action pair 250 times from which then the possible transitions are extracted. The reason why it is needed to observe each state-action pair many times is because the traffic light changes its color arbitrarily (i.e., the environment is non-deterministic), meaning that it can be possible that with fewer observed executions the changing traffic light (and thus changing state) is not observed. This would make the state transition function T incomplete, and thus a large number of repetitions for each state-action pairs is chosen. Due to the property of the traffic light the transition dynamics is non-deterministic. Because of this the transition probability for the traffic light going from red to green is set to 0.01. The transition probability of the traffic light staying the same color is set to 0.99. The transition going from green to red is also set with the same transition probability of 0.01 and staying green with the probability 0.99. The one remaining stochastic transition is that of reaching the traffic light. For this transition the probability of observing the traffic as red or observing it as green is equally likely (0.5 for both).

5 Results

In this chapter first the feature weights generated by the MAXENTIRL algorithm is provided. After this the behaviour produced by the IRL agent is compared to the behaviour of the expert agent.

5.1 Results of the Maxentirl algorithm

By initializing the feature weight vector to 0.2 for both features and initializing the stochastic gradient ascent with an exponential decay learning rate. This is initialized with a learning rate of 0.2 and a decay rate of 0.5. The resulting feature weights are:

- Feature [an observed traffic light is red] weight: 6.25.
- Feature [distance from the destination] weight: −138.11.

How the feature weights change during optimization is visualized in Figure 9 and Figure 10.

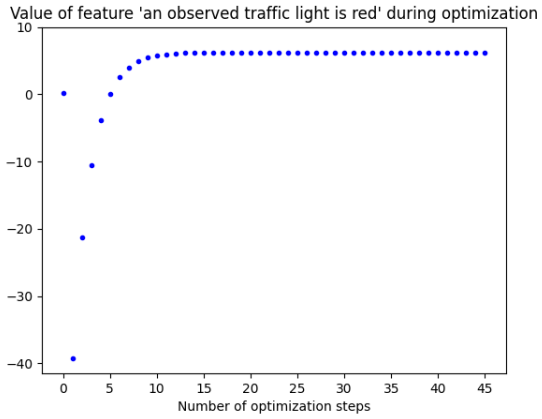

Figure 9: The feature weight for the feature "an observed traffic light is red" during optimization.

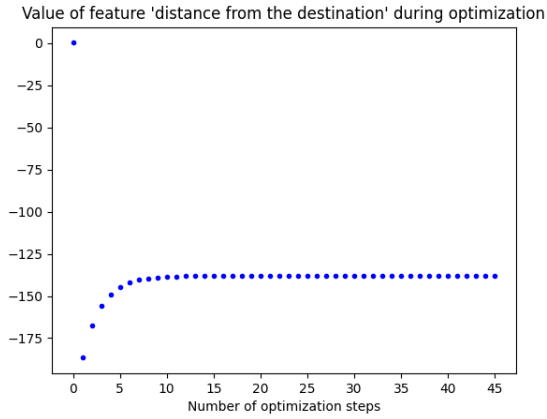

Figure 10: The feature weight for the feature 'distance from the destination" during optimization.

5.2 IRL Agent vs Expert Agent

With the feature vector now available it can easily be concluded how the IRL agent would act in different states. Inserting the feature weight vector $\boldsymbol{w}$ for the correct feature values into Equation 1 allows for calculating the reward value for any state s. From here the IRL agent only has to look at the state transition function T to determine which action is preferred. When the transition is non-deterministic a weighted sum is taken over the possible states it can transition to.
For the two scenarios is can easily be concluded that if in state s and the next state s' the observed traffic light state does not change after transition and the distance towards the goal gets smaller, then the agent prefers to advance to the next state. This is because for both s and s' the value for the feature "traffic light is red" would be the same. Thus only the distance feature would be

considered when deciding the action to perform. When s' is closer to the goal, then transitioning to that state is preferred. This is consistent with the behaviour of the expert agent in the observed scenario.
Transiting to the state that could observe a red traffic light (i.e., the edge of observing the traffic light) is also consistent with the behaviour of the expert agent. Both agents want to transition to that state. The interesting situation happens when the light is red. The expert agent will directly brake and continue only if the light is green. This is not the behaviour the IRL agent produces. Instead of instantly braking the IRL agent continues (while the light is red) until it is at the "edge" of observing the red traffic light and passing the traffic light. Here it will wait until the light turns green which causes the agent to continue. While the behaviour respects the traffic light to not pass until it is green this behaviour is obviously differs from that of the observed expert agent. Here the effect of the principle of maximum entropy is observed. From analysing the demonstration data it is found that the trajectories that had a red light observed would always observe them from the very first state in which observing a red light is possible. This would result in the MAXENTIRL algorithm to find a feature weight vector that always tries to observe a red light and thereafter tries to close the distance between the agent and the destination. With a very high probability (0.99) that transitioning from observing a red light would still result in observing a red light it makes sense that the IRL agent would continue all the way up until the next "no brake" action would result in observing no traffic light (i.e., passing the traffic light).
If the demonstration data would have demonstrated the importance of braking as a result of a red light from different distances from the traffic light, then it can be assumed that the MAXENTIRL algorithm would find the appropriate feature weight vector that would always keep the maximum distance between the agent and the traffic light. This behaviour of moving to the very edge of the traffic light when it is observed red is also observed during the second scenario, where the traffic light is green both agents continue for every state.

6 Conclusions

In this chapter the two research questions will be answered. An answer for the problem statement will also be given. Afterwards the limitations of this work are analysed. To conclude this chapter further research is discussed.

6.1 Research questions revisited

In this section the two research questions of this thesis will be answered. To end this section a conclusion is given for the problem statement of this thesis.

6.1.1 Research questions 1

The first research question was formulated as: can IRL be used to create an agent which is able to cross an intersection correctly if the correct behaviour is observed for that intersection?
With the result of the scenario 1 experiment it can be concluded that the IRL agent is able to correctly cross the same intersection on which the demonstration was observed. This is quite as expected since MAXENTIRL makes sure that the feature weight vector applied on the demonstration trajectories will produce behaviour that respects the feature properties.

6.1.2 Research questions 2

The second research question was formulated as: can IRL be used to create an agent which is able to cross in intersection correctly if the correct behaviour for that intersection has not been observed? Here a powerful example is provided for the application for IRL. By using the same agent that was applied in scenario 1 for scenario 2 generalizability of driving behaviour was observed. The foundation of IRL assumes that there is an underlying (reward) structure based on the features of a state. This makes it easy to generalize, since new states can be evaluated by the features of that state. This meant that the only task that was needed to be performed was to extract the features for these new states and generate a state transition function T for this new state space. When the extraction of the features is done consistently, then the agent will act in this new state space exactly as in the previous state space. This makes it possible to answer this research question by concluding that the IRL agent is able to also cross an unobserved intersection correctly.

6.1.3 Problem statement

The problem statement was formulated as: can IRL be used to simulate driving behaviour?
With both research question concluding that the IRL agent is able to behave correctly when it needs to cross an intersection for observed and unobserved intersections we conclude that IRL can indeed be used to simulate correct driving behaviour. For humans it is assumed that once the skill of driving a vehicle has been obtained, this can be applied in many (previously unobserved) situations. This generalizability has also been observed in the IRL agent in this thesis. While this generalizability has been observed it must be noted that this is due to similar changes in the feature "distance to destination" for the transitions. In both scenarios this distance decreases by 0.09 – 0.011 every time the agent decides to not perform a brake. This value is multiplied by its feature weight and thus plays an essential part in considering which action to take. If the amount of change

decreases enough (e.g., to 0.00001 for every time the agent decides not to perform a brake), then the IRL agent would prefer to never cross a traffic light. To prevent this change in behaviour it is essential to either use different features, or keep the (possible) changes caused by a transition consistent.

6.2 Limitations

While the results provided in this thesis are promising, the concept of a self-driving vehicle has severely be simplified due to time and resource constraints. For the environment the transition probability of the traffic lights were set manually, which might not be accurate for the environment. The friction on the tires of the vehicle were set to 0, meaning there does not exist friction between the vehicle and the road. This is done to obtain consistent state transitions. Also performing a brake would set the velocity of the vehicle to 0 instantly and moving forward will be done with a constant velocity. The environment in which the agent acts is one in which that agent is the only existing vehicle agent. A significant limitation that is set on the behaviour of the agent is that of the steering. The agents are not able to steer the vehicle. They can only decide to perform a brake or not. Observing the state and performing an action is done every 0.12 seconds, which may not match the reaction-time of a typical driver. The experiments are also performed on one specific map.

6.3 Further Research

In order to expand the results to make them viable for real-world applications the most important step to take is to include the steering as part of the possible actions to perform. This will enlarge the action space, but if sufficiently many trajectories are observed and the correct features are selected, then the IRL agent should be able perform both throttling/braking and steering at each time step. Another important aspect that needs to be improved is to consider multiple agents in the environment. Every aspect of driving, whether parking or merging on a highway, is greatly influenced by other drivers. This is also the case when crossing an intersection. The current setup acquires the status of the traffic light from the simulation. An important improvement would be to obtain the status of the traffic light from a sensor like a RGB camera. For further research experimenting with other features would also be very important in order to enhance the capabilities of the IRL agent.

References

[AD21] Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297:103500, 2021.

[AN04] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 1. Association for Computing Machinery, 2004.

[BTD+16] 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.

[BTR09] Oliver Brock, Jeff Trinkle, and Fabio Ramos. *High Performance Outdoor Navigation from Overhead Data using Imitation Learning*, pages 262–269. 2009.

[CN06] Wai-Ki Ching and Michael K. Ng. *Markov Chains: Models, Algorithms and Applications*. International Series in Operations Research and Management Science. Springer, 2006.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[Gra15] Franciszek Grabski. Discrete state space Markov processes. In *Semi-Markov Processes: Applications in System Reliability and Maintenance*, pages 1–17. Elsevier, 2015.

[Jay57] E. T. Jaynes. Information theory and statistical mechanics. *Phys. Rev.*, 106:620–630, 1957.

[KZH+23] Sateesh Kumar, Jonathan Zamora, Nicklas Hansen, Rishabh Jangir, and Xiaolong Wang. Graph inverse reinforcement learning from diverse videos. In *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pages 55–66, 2023.

[Lit01] M.L. Littman. Markov decision processes. In *International Encyclopedia of the Social & Behavioral Sciences*, pages 9240–9242. Elsevier, 2001.

[LPK10] Sergey Levine, Zoran Popovi, and Vladlen Koltun. Feature construction for inverse reinforcement learning. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems Volume 1*, NIPS'10, page 1342–1350. Curran Associates Inc., 2010.

[PMHC+23] Tung Phan-Minh, Forbes Howington, Ting-Sheng Chu, Momchil S. Tomov, Robert E. Beaudoin, Sang Uk Lee, Nanxiang Li, Caglayan Dicle, Samuel Findler, Francisco Suarez-Ruiz, Bo Yang, Sammy Omari, and Eric M. Wolff. DriveIRL: Drive in real life with inverse reinforcement learning. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1544–1550, 2023.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley Sons, Inc., USA, 1994.

[Rus98] Stuart Russell. Learning agents for uncertain environments (extended abstract). In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT' 98, page 101–103, 1998.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[YHC+20] Zhibo Yang, Lihan Huang, Yupei Chen, Zijun Wei, Seoyoung Ahn, Gregory J. Zelinsky, Dimitris Samaras, and Minh Hoai. Predicting goal-directed human attention using inverse reinforcement learning. *CoRR*, abs/2005.14310, 2020.

[ZLZ18] QiJie Zou, Haoyu Li, and Rubo Zhang. Inverse reinforcement learning via neural network in driver behavior modeling. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1245–1250. IEEE, 2018.

[ZMBD08] Brian D. Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence Volume 3*, AAAI'08, page 1433–1438. AAAI Press, 2008.

A List of Acronyms

Notation	Description	Page List
MAXENTIRL	Maximum Entropy Inverse Reinforcement Learning.	3, 9, 11, 12, 14, 16–19, 21–23, 29
IRL	Inverse Reinforcement Learning.	1–3, 6–8, 14–16, 19, 21–24
MDP	Markov Decision Process.	2–4, 6, 7, 12, 13, 29
RL	Reinforcement Learning.	2, 3, 5–7, 15

B List of Symbols

Notation	Description	Symbol	Page List
Action	An action that is legal to be performed within the environment with $a \in A$.	a	3–6, 13, 14, 28–30
Action space	A finite set of actions. When the possible actions depend on a state, then the finite set of actions for state s is denoted as A_s, with $A_s \subseteq A$.	A	3–5, 14, 24, 28–30
Action-value function	A function (under policy π) $Q_\pi \colon S \times A \to \mathbb{R}$ gives the expected return when starting in state s, taking the action a, and following policy π thereafter. A policy is always required in order to determine the value of a state. The value of a *terminal state* is always zero.	Q_π	4–6, 28, 29
Demonstration	A demonstration performed by an agent.	$\mathcal{D}$	7, 12–14, 17, 20
Discount factor	The discount factor determines the present value of future rewards, with $\gamma \in [0, 1]$.	γ	3–6, 28
Estimated reward function	The estimated reward function $\hat{R}_E$ is an estimate of the reward function R of the expert agent E.	$\hat{R}_E$	7, 9, 14, 18, 28
Expected return	A sequence $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{n=0}^{\infty} \gamma^n r_{t+n+1}$ of (expected) rewards received starting from discrete time step t, with discount factor γ.	G_t	3–6, 28
Expert agent	The expert agent for which it is assumed that its interaction with the environment is optimal.	E	7, 9, 11, 12, 14, 17–22, 28, 29
Policy function of the expert agent E	The policy function of the expert agent E.	π_E	7

Notation	Description	Symbol	Page List
Feature vector	The feature vector is the result of the mapping from a state to the k features of a state: $\boldsymbol{\phi}\colon S \to \mathbb{R}^k$.	$\boldsymbol{\phi}$	11–13, 21, 29, 30
Feature weight vector	The feature weight vector that defines the importance of each state feature in MAXENTIRL.	$\boldsymbol{w}$	11–14, 18, 21–23
MDP without reward function	A Markov Decision Process (MDP) without the reward function R known for the expert agent E.	$\mathcal{M}\backslash_{R_E}$	7
Optimal action-value function	An action-value function $Q_{\pi*}$ is considered optimal when for every other action-value function Q_π it holds that $Q_{\pi*} \geq Q_\pi$.	$Q_{\pi*}$	5, 6, 29
Optimal policy	A policy π_* is considered optimal when for every other policy π it holds that $\pi_* \geq \pi$ under the expected return. This means essentially that $\pi_* \geq \pi$ if and only if $V_{\pi^*}(s) \geq V_\pi(s)$ for all $s \in S$.	π_*	5, 6, 29
Optimal state-value function	A state-value function $V_{\pi*}$ is considered optimal when for every other state-value function V_π it holds that $V_{\pi*} \geq V_\pi$.	$V_{\pi*}$	5, 6, 29
Policy function	A function $\pi\colon S \times A \to [0, 1]$ from states to probabilities of selecting each possible action.	π	3–5, 7, 13, 14, 28–30
Reward	A reward obtained at time step t.	r_t	3, 4, 28
Reward function	A function $R\colon S \times A \times S \to \mathbb{R}$ that maps the reward that will be obtained for executing a triplet (s, a, s').	R	3, 5–7, 9, 11, 13, 14, 17, 18, 28, 29
State	A state which is within the state space S.	s	3–6, 11–14, 21, 22, 28–30
State-action pair	The state-action pair which provides the action a taken in state s.	(s, a)	14, 20

Notation	Description	Symbol	Page List
State space	A finite set of states of the environment.	S	3–6, 11, 13, 14, 23, 28–30
State Transition function	A function $T\colon S \times A \times S \to [0, 1]$, giving for each state and action, a probability distribution over states.	T	3, 5, 6, 18, 20, 21, 23, 30
State-value function	A function (under policy π) $V_\pi\colon S \to \mathbb{R}$ gives the expected return when starting in state s and following policy π thereafter. A policy is always required in order to determine the value of a state. The value of a *terminal state* is always zero.	V_π	3–6, 29, 30
Time step	A discrete time step.	t	3–5, 28, 29
Trajectory feature count	The trajectory feature count $\boldsymbol{\phi}(\tau)$ is the sum of every state feature observed during the trajectory τ..	$\boldsymbol{\phi}(\tau)$	12, 13, 30
Trajectory	A sequence $\tau = ((s_0, a_0), (s_1, a_1), \ldots, (s_n, a_n))$ of state-action pairs (with $n \in \mathbb{N}$).	τ	3, 7, 12, 13, 30