UNIVERSITY OF AMSTERDAM

Q-learning with heuristic exploration in Simulated Car Racing

by Daniel Karavolos

A thesis submitted in partial fullfilment of the requirements for the degree of
Master of Science in Artificial Intelligence
at the University of Amsterdam, The Netherlands.

Supervised by dr. Hado van Hasselt, Centrum voor Wiskunde en Informatica

August, 2013

Contents

1 Introduction

This thesis aims to explore how reinforcement learning can be used to efficiently train a driver in a racing game. In particular, we study the effects of guided exploration on the learning time of the agent. As application, we have used the framework of the Simulated Car Racing Championship, which is a plugin to The Open Racing Car Simulator (TORCS). See figure 1.1 for a screenshot.

We will first give short historic overview of the relation between artificial intelligence and games, with an emphasis on reinforcement learning. Then, the Simulated Car Racing Championship will be described, including the software framework and some interesting competitors. This is followed by other related work in simulated car racing, the specific research questions of this study and an outline of this thesis.

1.1 Artificial Intelligence and Games

Since the early days of the computer, it has been used by researchers in the field of artificial intelligence (AI) to develop game-playing algorithms. As early as 1952 Arthur Samuel wrote a checkers program for the IBM 701, one of the first computers. In 1959 Samuel introduced a learning algorithm that tried to optimize its piece advantage, which highly correlates to winning in checkers (Samuel, 1959). The program used an evaluation function to rate the board positions, and a heuristic look-ahead search based on the minimax-algorithm (Shannon, 1950) to determine the best available move. It would store the value of a board position based on the combination of a real move, and the value returned by the heuristic, which was discounted for each ply in the search tree. This is one of the first applications of reinforcement learning (RL). Although it only achieved novice level play, it is considered as a significant achievement in the field of machine learning (Sutton and Barto, 1998).

Over the years, computer games proved to be an ideal testbed for AI algorithms. Games are much simpler than problems in the real world, but complex enough to be a challenge. And since Samuel's work on checkers, the field has grown rapidly. Initially, research focused on traditional board games, such as othello, checkers and chess. These games are turn-based, deterministic and each agent can perceive the complete state of the game at all times. This made them suitable platforms for the symbolic, search-based AI algorithms that were the main area of interest at that time. Soon, the focus was extended towards games with imperfect information and stochasticity, such as poker and backgammon. For an extensive overview, we refer to Schaeffer (2000) and Fürnkranz (2001).

In the 1990s many successes were booked in terms of the level of play that the AI programs achieved. The first was the defeat of World Checkers Champion Marion Tinsley in 1994 by the program Chinook (Schaeffer, 1997). It was the first program to win a human world championship.

Arguably the most famous achievement was the defeat of World Chess Champion Gary Kasparov in an exhibition match in 1997 by Deep Blue(Campbell et al., 2002; Hsu, 2002). Even though the success of Deep Blue depended on specialised hardware and brute-force search rather than actual intelligence, it is considered as a milestone in the history of artifical intelligence.

Another success was the application of temporal difference (TD) learning (Sutton, 1988) to train a neural network to play backgammon. The program was called TD-Gammon, and it achieved a world class level by means of self-play (Tesauro, 1995). Because it learned by playing versus itself, instead of relying on searches through expert knowledge, it was able to come up with strategies that changed the way human experts play backgammon.

During the same period, the video games industry grew dramatically (Williams, 2002) and received increased academic attention. In the field of artificial intelligence, Laird and van Lent (2001) explored the various roles AI could play in video games. They stated that video games would be an interesting application for AI algorithms for a number of reasons, one of them being that the various game genres allow for more than just the creation of strategic opponents. They also proposed that AI could be used for interactive storytelling, sophisticated support characters and automated commentary. Since then, video games have become an increasingly more popular application for AI, and computational intelligence specifically. For an overview of the various applications of AI in games, see (Galway et al., 2008; Miikkulainen et al., 2006; Lucas and Kendall, 2006).

One of the types of AI algorithms that combine well with games is reinforcement learning (RL). The concept of RL is, in short, that an agent acts in an environment and learns to perform certain behavior through the feedback it gets from the result of its actions. Video games usually provide exactly what an RL algorithm needs, namely a number of agents acting in an interactive world. Since video games provide challenging problems that are suitable for RL, and also provide the software framework to work with, it is an interesting field of application. Various types of games have been used for reinforcement learning studies, such as racing games (Barreno and Liccardo, 2003; Abdullahi and Lucas, 2011), fighting games (Graepel et al., 2004), real-time strategy games (Maderia et al., 2006; Ponsen et al., 2006), and first-person shooters (McPartland and Gallagher, 2008). The main focus of these studies was to use RL for game agent control, that is, to have a single autonomous agent learn how to achieve certain goals in a game.

Racing games have been a popular genre of video games for AI applications. In racing games optimal game agent control covers most of the game AI, since this highly correlates with optimal driving behavior. However, computational intelligence can be used for more than just game agent control in racing games. For example, it could be used to model players' driving styles or to generate racing tracks. In the commercial game Forza Motorsport, for instance, a driving agent is trained to imitate the driving style of the player (Microsoft, 2013). This so-called Drivatar can then be pitted against other players online. For an overview of possible applications of computational intelligence in games, see (Togelius et al., 2007).

1.2 Simulated Car Racing

The Simulated Car Racing Championship exists to promote the application of computational intelligence to racing games, or simulated car racing (SCR). It is held during several conferences, such as the IEEE World Congress on Computational Intelligence, IEEE Congress on Evolutionary Computing, and the Symposium on Computational Intelligence in Games (Loiacono et al., 2008, 2010a).

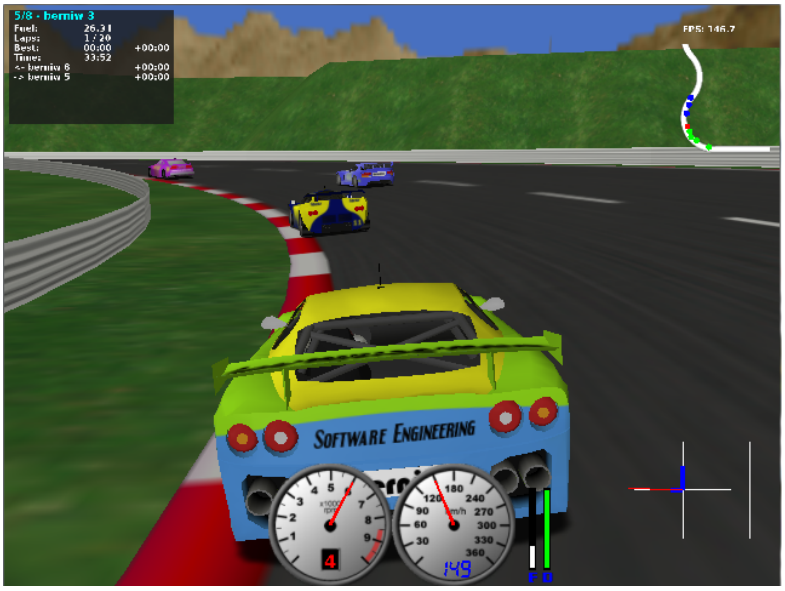


Figure 1.1: A screenshot of TORCS.

SCR Championship Software

The Open Car Racing Simulator[1] is a highly customisable, open source car racing simulator that provides a sophisticated physics engine, 3D graphics, various game modes, and several diverse tracks and car models. Because of this, it has been used in the Simulated Car Racing championship since 2008 (Loiacono et al., 2008, 2010a).

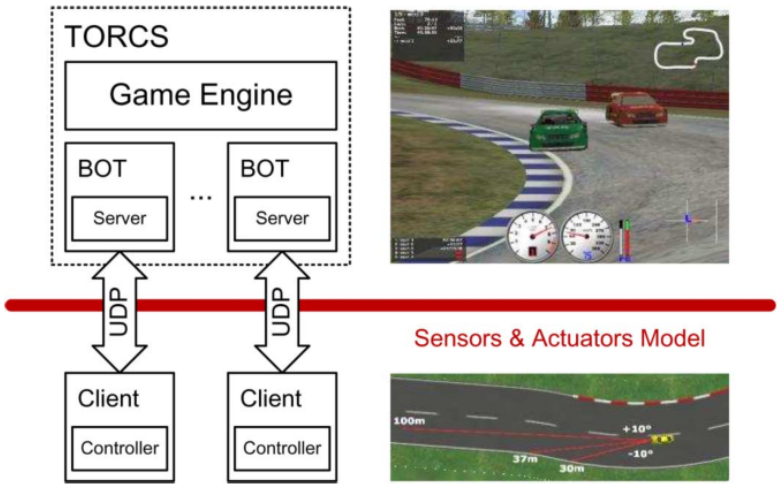
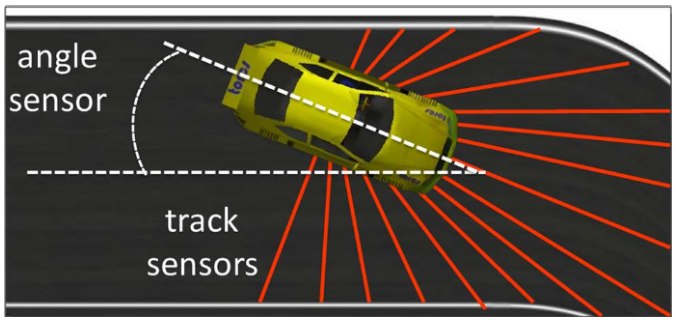


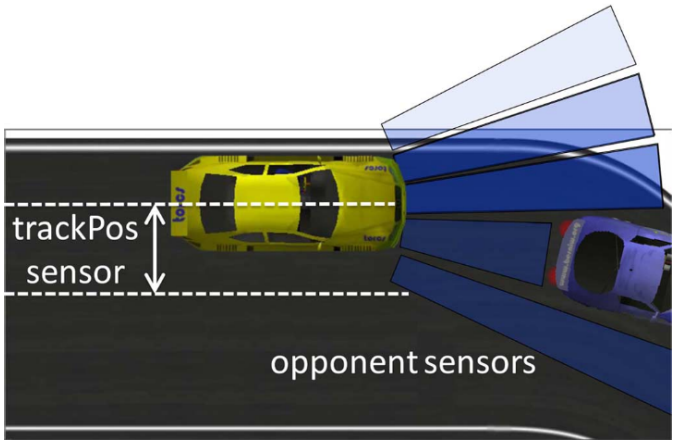
Figure 1.2: The Simulated Car Racing championship architecture of TORCS. From: (Loiacono et al., 2010a)

[1] Available at http://torcs.sourceforge.net. Accessed June 11, 2013.

Normally, the cars in TORCS have access to all information, including the environment and, to a certain degree, other cars. This is not representative of autonomous agents acting in the real world. Therefore, the SCR championship provides the participants with a software interface that changes TORCS into a server-client system (Loiacono et al., 2012), see figure 1.2. The server acts as a proxy for the environment and the client provides the control for a single car. The controllers run as external programs and communicate with a customized version of TORCS through UDP connections. This way, there is no direct access to the information of the game engine. At each control step (or, game tick), the server sends the client the available sensory input. In return, it receives the desired output of the actuators. This separates the controller from the environment, allowing it to be treated as an autonomous agent. The complete list of sensors is given in Table 8.2, and includes the current speed, the raced distance, and range finders to perceive the distance to opponents, or to the side of the track. Figure 1.3 shows a visual representation of the four most important sensors. The actuators include the steering wheel, the gas pedal, the break pedal and the gearbox (see Table 8.1). From now on, when this thesis refers to TORCS, it is meant to refer to TORCS with the SCR Championship plugin included.



(a) A representation of the angle and track sensors.



(b) A representation of the trackPos sensor and the opponent sensors.

Figure 1.3: A visual representation of four types of sensors. From: (Loiacono et al., 2010a)

Evolutionary Computing in Simulated Car Racing

Evolutionary computing (EC) is a subfield of Artificial Intelligence that tries to solve optimization problems by creating a population of solutions that is exposed to the Darwinian principles of evolution. There are several types of EC, genetic algorithms and evolution strategies, among others. The main differences between these types of EC is the way they encode the solution into a genotype and how they generate offspring. For more information about EC, we refer to Eiben and Smith (2003).

Simulated car racing is picked up as application for evolutionary computing, because it allows the use of predefined policies that can be tuned by EC. Also, the SCR championship is mainly held at conferences that concern evolutionary computing. Even though these methods are not the subject of this study, this section describes four successful approaches (Loiacono et al., 2008, 2010a; Onieva et al., 2012), as a frame of reference. This is based on the championships of 2009 and 2010, because later championships have not been described in detail in publications.

Autopia

Autopia won the championship in 2009 and 2010 (Loiacono et al., 2010a; Onieva et al., 2012). Autopia consists of several modules that correlate with driving tasks (Onieva et al., 2012). These modules are: gear control, stuck management, pedal control, steering control, and target speed determination. The gear control module shifts gear at predefined rpm-values. The stuck management module takes over control when the car drives off-track, because in that case the sensor values become unreliable. The pedal control uses a sigmoid function that depends on the target speed to control acceleration and braking. All of the modules are heuristically designed, except for steering control and target speed determination. Those two contain functions with weights attached to the input of several track sensors.

The steering control tries to steer towards the track sensor with the largest reading. The desired angle depends on that sensor and the six sensors around it. The target speed determination is based on a combination of the five frontal track sensors between -20 and 20 degrees. The weights in these two modules are optimized with a genetic algorithm. The performance of each individual is based on the sum of the traveled distance in 80 seconds on four different oval tracks, including a penalty for getting stuck or damaged.

There are two modules that modify the optimal values to adjust the behavior to new situations. The first is the opponent modifier, which adjusts the steering value to avoid collisions and overtake opponents. The second is a learning module, which stores a multiplier for each meter of the track. The accidents on the track during a training lap cause the agent to decrease the target speed for the 250 meters before the position of that accident.

COBOSTAR

COBOSTAR was runner-up in the championship in 2009 and 2010 (Loiacono et al., 2010a; Onieva et al., 2012). The setup is much like Autopia. It uses hand-designed policy functions to map information from the distance sensors to a target speed and angle (Butz et al., 2009). These target values are then modified for particular situations, such as jumps and overtaking opponents. Butz et al. (2009) used the co-variance matrix adaptation evolution strategy (CMA-ES) to optimize the weights in these functions. Just like Autopia, COBOSTAR has a separate hand-tuned strategy for gear shifting, based on rpm-values. The additional modules are crash adaptation,

recovery behavior, acceleration reduction, and jump detection.

Note that, unlike Autopia, COBOSTAR does not have a separate, hand-coded policy for driving off-road. Instead, the calculation of its target speed is changed. Instead of relying on the distance sensors, which become unstable off-road , it uses a function based on its track position and angle relative to the track. This function is also optimized with CMA-ES. Nevertheless, there is a separate function (the acceleration reduction module) that decreases the target speed based on the wheel spin of the rear wheels to avoid wheel slips when the car is off the road. Moreover, there is no separate policy to deal with opponents. To avoid opponents, COBOSTAR creates a model of the relative speed and distance of the opponents within the range of the opponent sensors. The opponents are then treated as moving obstacles, overriding the values of the distance sensors when necessary. Additionally, the target speed is reduced if an opponent is right in front of COBOSTAR (less than 15m).

The additional modules are implemented as a subsumption architecture, overriding the standard sensor-to-motor mapping when they are activated. The parameters in these modules are not optimized with CMA-ES. The recovery behavior consists of an elaborate decision mechanism that shifts to the rear gear if the car is stuck outside the track. The jump detection uses a comparison between wheel rotations and forward and lateral speed to detect a jump and ensures safe landings by forcing the front wheels to be turned towards the direction the car is flying until a landing is detected. The crash adaptation module stores the location and severity of a crash. In successive rounds, it decreases the target speed in a crash adaptation area, both of which depend on the severity of the crash and whether or not another car was involved.

Mr. Racer

Another way to tackle this problem domain is to create a track model and learn how to plan on this model. This method is employed by Mr. Racer (Quadflieg et al., 2010). Quadflieg et al. (2010) devised a measure for the curvature of the track, based on the agent's 19 track sensors. During the warm-up phase, the agent builds a track model of segments based on the encountered curves. Each track segment is categorized into one of six types. The driver can be separated into two modules, a basic controller and a planning module. Note that, unlike the above mentioned drivers, Mr. Racer does not contain a separate module to handle opponents.

The basic controller consists of an optimized pedal control and a heuristic steering control. The pedal control is different for each type of curve and depends only on the current speed. The decision function is represented as a one dimensional linear function with a negative gradient and a range of [1,-1]. An evolution strategy is then used to optimize this function by tuning the zero point and the gradient. The steering control is adapted from Kinnaird-Heether & Reynolds (described in (Loiacono et al., 2008)) and mainly uses the angle of the largest distance sensor as the steering angle. The adaptation consists of increased weights for hairpins and slow bends.

The planning module is used only in sections of the track that contain a succession of two curves with a straight track in between. The planning module uses an evolution strategy to optimize a plan that defines in four stages how to traverse from one track segment to the other.

Later improvements on this involve representing the target speed as a function of the curvature, which is optimized off-line with CMA-ES, and an online adaptation mechanism that is deployed during the warm-up phase to adapt the parameters to the current track (Quadflieg et al., 2011).

Mr. Racer obtained the fifth place in the SCR Championship in 2009 (Loiacono et al., 2010a), though it should be noted that it didn't get any points in the second leg due to software crashes. In 2010, Mr. Racer obtained the fifth place again Onieva et al. (2012), which is right behind Polimi.

Polimi

Cardamone's nameless NEAT-driver (henceforth Polimi, after (Onieva et al., 2012)) uses neuro-evolution of augmenting topologies (NEAT) to evolve a neural network (Cardamone et al., 2009). Gear shifting and recovery behavior were implemented as separate, hand-coded modules.

The neural network in Polimi receives only a small subset of the available sensors as input: the current speed, 6 track sensors (at -90°, -60°, -30°, 30°, 60° and 90°), and one "improved" frontal track sensor, which returns the maximum value of the track sensors at -10°, 0°, and 10°. The network has two output nodes, one for acceleration and braking and one for steering.

Cardamone et al. (2009) evolved 100 networks for 150 generations. The fitness function is a linear combination of the game ticks spent off the road, the average speed and distance raced by the car during the evaluation lap. The performance was measured at the `Wheel-1` lap. Additionally, there is a separate policy for straight sections of the track. When the frontal sensor does not perceive the edge of the track, acceleration is set to 1. This prevents the evolutionary algorithm from wasting time on safe but slow controllers.

Overtaking behavior is evolved as a separate controller. This controller is activated when an opponent is perceived within 40 meters. This overtaking controller uses 8 opponent sensors (four in front, two towards the side and two at the back) in addition to the above mentioned sensors. To evolve the controller, Cardamone et al. (2009) create a situation of 40 seconds at various positions on the track, in which overtaking desirable. The fitness of a controller is a linear combination of the number of game tics that the controller is outside the track, the number of game ticks that a collision is detected, and the difference between the position of the opponent and the controller's final position.

Cardamone et al. (2009) report that Polimi can outperform the SCR Championship winners of 2008. In the SCR Championship of 2009 the controller placed second, and in 2010 it obtained the fourth place (Loiacono et al., 2010a; Onieva et al., 2012).

Reinforcement Learning in Simulated Car Racing

Although evolutionary computing has been the most successful type of computational intelligence that has been applied to SCR, there have been several attempts to use RL to create an agent that learns driving behavior. A few of these attempts will be described below.

Overtaking behavior with tabular Q-learning

Loiacono et al. (2010b) applied tabular Q-learning to learn overtaking behavior in TORCS. They used the built-in bot berniw (one of the best bots available for TORCS) as a base for the agent, and replaced the policy for overtaking behavior by a learning module. In overtaking, they discerned two sub-tasks, trajectory change and brake delay. 'Trajectory change' consists of exploiting the drag effect of an opponent in front to gain speed and changing trajectory to complete the overtaking. This is used to overtake a fast driving opponent on a straight stretch or a long bend. 'Brake delay' is performed in tight bends by steering towards the inside of the bend and delaying the braking action while the opponent slows down, thereby surpassing the opponent.

For 'trajectory change', the agent used four parameters as input, the longitudinal and lateral distance to the opponent, the lateral position of the agent on the track and the speed difference between the agent and the opponent. These continuous dimensions were discretized into 6 - 10 bins. Note that acquiring the values that are relative to the opponent would require some processing within the SCR Championship framework, since that framework only reveals the distance to the opponent according to 36 sensors under various angles (see Table 8.2). This 'trajectory change' module would only control the steering wheel, the gas pedal was left to the base agent. As opposed to a more direct control of the gas pedal, three high level actions were defined: 'move 1 meter to the left', 'keep this position', and 'move 1 meter to the right'.

For 'brake delay', Loiacono et al. (2010b) gave the agent three inputs, the longitudinal distance to the opponent, the distance to the next curve, and the speed difference between the agent and its opponent. These continuous dimensions were also discretized into 6 - 10 bins. The brake delay was built on top of the ABS and speed modules, and had no direct control of the gas pedal. The only output it gave was whether to inhibit braking or leave the decision to the lower level components, which was encoded as 1 and 0.

The reinforcement signal was based on whether the overtake was successful (1) or whether it had crashed (-1). Otherwise, it received a reward of 0. After several thousand episodes, both modules were separately able to improve upon the policy of Berniw. Despite the fact that this study focused on a small sub-task of racing, the results are promising for the use of reinforcement learning in SCR. Especially for the improvement of existing strategies.

Overtaking behavior with neural networks

Pyeatt and Howe (1998) aimed to create a car racing agent that manages its own development. They proposed a two layer control system, in which RL is used to learn when to switch between reactive components. That is, to learn strategic behavior instead of sensory-motor control. These reactive, heuristic components could be incrementally substituted by learning components. In its final form the agent should recognize when a new behavior is needed, and create and train a neural network to perform that behavior.

Their experiments, however, describe a system that uses separate neural networks for each behavior and a simple rule based coordination mechanism. The neural networks are trained with Q-learning and a TD(λ) approach (Barto et al., 1989), which means in this case that the value of an action in a state is updated with the information of up to 15 time steps later. The agent has three acceleration actions and three steering actions. The acceleration actions are: accelerate 10 feet/sec, decelerate with 10 feet/sec, and not adjusting the speed. The steering actions are: steer 0,1 radians to the left, 0,1 radians to the right, and no steering.

Despite the limited actions, both the driving and the overtaking behavior were learned successfully. However, Pyeatt and Howe (1998) conclude that neural networks are far from ideal internal representations in this setting, because they lack local updates. The global updates of the neural network might cause good behavior to be "forgotten" after subsequent updates, which can easily cause the agent to get stuck in local optima. But they also conclude that the straightforward alternative, the traditional table lookup method, does not scale well to this kind of problem.

A neural network controller trained with SARSA(λ

Using another racing simulation (RARS), Barreno and Liccardo (2003) reported limited success with the use of SARSA(λ)(Sutton, 1988) to learn to steer a car on an oval track. They compared

the performance of function approximators with three levels of complexity, tile coding with discrete output (QDummy), tile coding with continuous output (QSmarty), and an artificial neural network.

For their tile coding agents, Barreno and Liccardo (2003) did not exploit the fact that the agent can receive online feedback during each time step. Instead, they treated the problem as an undiscounted, episodic task. Each episode ended when the agent completed one lap, crashed or reversed its direction. The first outcome would result in a fixed positive reward, the other outcomes in a fixed negative reward.

The episode was the same for the neural network. But unlike the tile coding agents, it was given a small positive reward at each time step. Although a small negative reward would be more suitable for a racing problem, they argued that the network was so biased towards steering sideways that they expected that this small positive reward would encourage the network to stay on the track. Their results did not support this, since the network still had the tendency to drive off the track.

As input, the tile coding agents received the agent's speed, its lateral position on the track and its angle relative to the center of the track. Both types of agents used 10 tilings, distributed randomly over the input space using a hashing function on 12000 parameters (Sutton, 1988), as a function approximator. QDummy would output one of three discrete steering actions (left, straight, and right), whereas QSmarty would output a continuous number. The speed of the car was fixed.

QDummy would learn to stay on the track after approximately 10 episodes, and the policy had converged after approximately 30 laps. However, the agent would generally learn to drive on the outside of the curve, which is far from optimal. The high convergence speed might be attributed to the simplicity of the track. Barreno and Liccardo reported that the agent required hundreds of episodes to finish the first lap when it was confronted with a more complex track. They concluded that more input features would be necessary to learn to steer on these more complex tracks.

In another experiment, the fixed reward was changed into a reward that depends on the average speed if the lap was completed, and the traveled distance along the track axis when the agent crashed. It took 80 episodes for the Q-function to converge. QDummy would now learn to drive on 2/3 of the inside of the curve. This is better than before, but still not optimal. Barreno and Liccardo attribute this "cautious" behavior to the need to correct random actions that result from exploration. However, the use of a decreasing exploration rate did not change this.

The results of QSmarty are similar to QDummy, even the convergence rates. It should be noted, though, that QSmarty did seem to have noticeable smoother control. Barreno and Liccardo did not speculate as to why this large difference in action space had so little effect. Perhaps it is due to the fact that they approached this as an undiscounted, episodic task.

Barreno and Liccardo (2003) reported the laborious, frustrating work of tuning the neural network manually. They tried many variations of the number of inputs, the number of hidden nodes, the activation functions of the nodes and even a special momentum function to speed up learning. However, the network never responded with learning the desired steering behavior. Barreno and Liccardo could not come up with an explanation other than that a neural network without extra heuristics might not be a suitable function approximator for this problem.

Finally, they noted that a simple, heuristic controller that consisted of 3 lines of code gave

more satisfying results than the above mentioned reinforcement learning controllers. From this, they concluded that it seems necessary to give an agent more structure to solve continuous optimal control problems, such as simulated car racing.

1.3 Research Questions

Simulated car racing is a complex problem domain with many continuous input dimensions and several continuous output dimensions. This makes it a more interesting challenge for reinforcement learning algorithms than traditional benchmarks, such as the mountain car problem and the pole balancing task (Sutton, 1988). Although the complexity of simulated car racing is still miles away from actual autonomous cars, such as (Thrun et al., 2006), solving the challenges of simulated car racing is a step closer to solving real world problems.

The challenge of creating an autonomous racing car could be described as having a car learn how to race by driving on a race track and giving it feedback. Alternatively, one could create a population of race cars that drive around, and expose them to the Darwinian principles of natural selection and mutation to evolve a competitive car from this population. These descriptions are respectively the concepts of reinforcement learning and evolutionary computing.

Although both descriptions seem like an adequate approach, it has been tackled predominantly by EC. This has probably something to do with the fact that the complexity of this problem invites researchers to use predefined strategies and heuristics to bootstrap their algorithms. Since EC algorithms are more typically used as optimization techniques than RL, this seems the more obvious choice.

There has been a successful attempt to create a car controller from scratch, using an evolutionary algorithm to create an artificial neural network (described in section 1.2). However, creating a car controller from scratch is more typically a reinforcement learning approach and might be more suitable to RL algorithms. Evolutionary computing relies on fitness values, which can only be computed after a complete race. This means that the agent gets feedback on its performance only after a complete race. In contrast, with reinforcement learning the value of a state or a state-action pair is computed, which can either be done during racing or after a complete race. Either way, each time step can be used as feedback for the agent's performance. The data acquired during a race can thus be used more efficiently, which should cause the agent to need less races to acquire a good policy. Indeed, this is what Lucas and Togelius (2007) conclude after comparing a reinforcement learning controller (based on SARSA) with an evolved neural network controller (based on evolution strategies) in a simplified car racing problem. However, they also conclude that the RL algorithm is less reliable, since it quite frequently does not learn anything at all. Thus, the challenge seems to be to produce a more reliable reinforcement learning algorithm.

A problem of RL might be that the agent explores too much irrelevant states of the world. The population point of view of EC allows the algorithm to move away more easily from agents that cannot drive at all. Whereas, due to the frequent, but small updates, a reinforcement learning algorithm might take longer discard a certain type of solution. For example, an agent that crashes in the first curve might quickly be discarded by an evolutionary algorithm, because it does not cover enough distance. However, if the agent were trained with RL, it would first have to learn that crashing is bad and then learn how to prevent it[2].

[2]Though with some algorithms, this might be learned simultaneously.

A solution to this exploration problem could be to guide the exploration of a reinforcement learning agent to actions that seem promising. This could be done by extending the action-selection mechanism with a heuristic that regularly tells the agent what would be a good action to perform next. The primary question that this thesis aims to answer is *What is the effect of the extension of regular heuristic guidance on the performance of Q-learning in simulated car racing?*. In particular, this study will extend the very popular ϵ-greedy mechanism. The RL algorithm that is used in this study is Q-learning (Watkins, 1989). Q-learning is easy to implement and also very popular. However, the concept should apply to any RL algorithm that applies ϵ-greedy for action selection.

Since there seems to be no literature available that describes the use of reinforcement learning to learn a direct mapping from input to output for both steering and accelerating, it remains a question whether it is possible at all to use reinforcement learning for this purpose. Therefore, another aim of this thesis is to show that reinforcement learning can be used to train an agent to drive around a track with minimal prior knowledge.

1.4 Outline

Chapter 2 will describe the theoretical background of reinforcement learning. The problem domain will be briefly introduced, including the traditional methods to solve a reinforcement learning problem and Q-learning. It will describe the choice for tile coding as function approximator and the proposed action selection mechanism with a heuristic component. Chapter 3 will describe simulated car racing as a reinforcement learning problem, including a way to implement it within the SCR Championship framework. Chapter 4 describes the experimental setup and the reasoning behind choice of the parameter values. Chapter 5 describes the results of the experiments. Chapter 6 discusses the results and the experimental setup. Finally, chapter 7 contains the conclusions that can be drawn from this study and directions for future work.

Reinforcement Learning 2

This chapter briefly describes the reinforcement-learning framework and several classical techniques for solving a reinforcement learning problem. Section 2.1 describes what a reinforcement learning problem is, and section 2.2 describes how one could be solved. Section 2.3 describes methods for approximating the value of a state, and specifically tile coding. Section 2.4 presents the proposed action selection mechanism for guiding the exploration with a heuristic. Section 2.5 briefly compares the proposed action selection mechanism with other ways to learn from an existing policy.

2.1 What is Reinforcement Learning?

Reinforcement learning (RL) partly evolved from animal training in biology. It works much like operant conditioning, which is a method of training animals to show active behavior (not to be confused with classic conditioning, which describes the training of reflexes). A famous experiment in the field of operant conditioning is that of Skinner (1932). Skinner trained a caged rat to press a bar by giving it a small food pellet as a reward every time it pressed the bar. One could say that reinforcement learning is the computational equivalent of the biological operant conditioning, since RL also focuses on training active behavior, but then to computer programs.

Like in operant conditioning reinforcement learning involves a decision maker, or agent, learning from interacting with its environment by taking actions and observing their rewards, the famous method of trial-and-error. Since you can't give a computer program food pellets, these rewards are numerical. The agent must accumulate as much reward as possible. Traditionally, these rewards are delayed, i.e. the agent receives no reward until it achieves its goal. But there are settings in which the agent receives a reward every time step, which is also the case in this research.

In a reinforcement-learning problem there is no teacher that tells the agent what is the best response. The agent must learn the effect of its actions by trying them. At the same time, the agent must maximise its reward. This creates a tension between exploration and exploitation. To maximise its reward, the agent must exploit the knowledge it has already gained and select actions that it knows to yield a high reward. But to find the best actions, it must explore the action space and try actions with an unknown effect. For example, in a certain state of the world, the agent might already know a decent response, but there might be an even better action that it has not tried yet. If the agent does not explore its action space, it might never find out that a better response exists.

In short, RL can be seen as having an agent learn to map situations to actions, in which actions are chosen in such a way that the agent maximizes its numerical reward over time. In order to

let computer programs learn from these situations and actions, the environment is formulated as a mathematical framework that is called a Markov Decision Process (MDP) (Bellman, 1957b; Howard, 1960).

Markov Decision Process

In a Markov Decision Process time is modeled as discrete steps, $t = 0, 1, 2, 3, ...$, and transitions from one state of the world do not necessarily deterministically lead to another. This defines it as a so-called *discrete time stochastic control process*.

The state of the world is modeled as a finite set of S states. At each time step t, the process is in some state $s \in S$. The state defines what the agent perceives as the environment in which it acts. This might be raw information, like its location or the angles of its joints, or preprocessed information, such as its velocity or the curvature of the road. There is no limit to the amount of preprocessing, as long as the agent does not receive information that it shouldn't know, like the next card in a closed card deck. Actually, preprocessing raw information to reduce the number of states is part of the art of formulating the reinforcement learning problem. For example, by correcting for all forms of symmetry in tic-tac-toe, the size of the state space can be reduced from 19.638 to 304. Training an agent on such a reduced state space can save a lot of computational resources.

The agent can choose to do any action $a \in A(s)$, where $A(s)$ is the set of actions available in state s. The next time step the environment responds by changing to state s' with a certain probability, denoted by $T(s'|s, a)$, that depends on the previous state s and the chosen action a. Upon changing to this next state, the agent receives a reward for taking action a in state s, denoted by $R(s, a)$. The reward signal is what the agent strives to optimize, and should therefore be carefully defined.

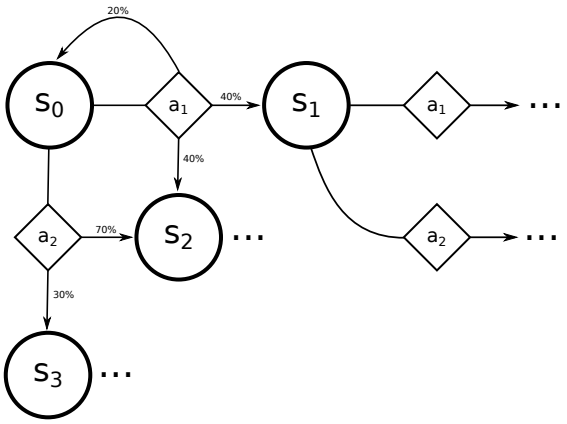


Figure 2.1: An abstract example of a Markov Decision Process. In RL each state transition results in a reward from the environment.

It is tempting to use the reward signal to give the agent prior knowledge about how it should achieve its goals by introducing sub-goals. However, the agent could learn to achieve its sub-goals or otherwise gain sufficient rewards without achieving its actual goal. Such situations inhibit learning and should be avoided.

Like putting a coin in a slot machine might yield different results, in certain MDPs the reward of performing an action in a certain state might differ. However, in this study we only consider deterministic rewards. Which is, in this case, similar to putting a coin in a vending machine.

Note that the transition probability to a certain state s' only depends on the current state s and agent's action a. Which means that given s and a, the probability of ending up in the next state s' is conditionally independent of all other previous states and actions. In other words, this process has the *Markov Property*.

Besides having the Markov property, a MDP assumes that the agent observes all there is to know about the environment, i.e. the environment is fully observable. In that case, state s, action a and transition probability $T(s'|s,a)$ are all that is needed to correctly predict the next state. However, there are problems that include information in the environment that cannot be observed by the agent. For example, TORCS is normally fully observable, since the agent can perceive the whole track and the exact positions of all cars, etc. Whereas in the SCR framework, the agent can only perceive a part of its surroundings. In that case the environment is only partially observable. This process is a generalization of the MDP and is called a Partial Observable Markov Decision Process (POMDP). A POMDP could be modeled with a probability distribution over the set of possible current states. However, in this study we will assume that the agent can correctly predict the next state from the given input and its action, and approach it as an MDP.

2.2 Solving a Reinforcement Learning problem

Now that we have defined a framework, we can focus on obtaining a solution. The solution to an MDP is called a policy, denoted as π, and specifies an action for any state that the agent might reach. Since an MDP is stochastic, each time a policy is executed might lead to a different sequence of states. The quality of a policy is therefore measured by the *expected* total return of that policy. The optimal policy π^* is the policy that yields the highest expected return.

Although a race always ends after a number of laps, it is beneficial to approach racing as an infinite horizon problem. Then we can use stationary policies, i.e. policies that do not depend on the time that a state is visited. However, in an infinite horizon problem the agent might never reach a terminal state, creating an infinite sequence of states with an infinite sum of rewards. In that case, it would be impossible to evaluate a policy.

We can avoid this infinite sum of rewards by discounting them over time. With discounted rewards, the value of an infinite sequence is finite. Given that the rewards are bound by some finite value R_{max} and that $\gamma < 1$, the maximal return of a sequence of states is given by

$$\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = R_{max}/(1-\gamma). \tag{2.1}$$

To define the expected value of a particular policy it is necessary to define the value of being in a certain state. This partly depends on the reward that was obtained by taking an action in that state, but it also depends on the states that the agent can reach from this state. Bellman (1957a) introduced the value function, a way to express the expected value of a state in terms of its reward and the future returns, given a fixed policy π. It became known as the Bellman equation, defined by

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) V^{\pi}(s'). \tag{2.2}$$

When we replace the fixed policy with a max-operator over the action that maximizes the

discounted return of the next state we get the equation for the optimal policy,

$$V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} T(s'|s,a)V^*(s'), \tag{2.3}$$

which is called the Bellman optimality equation.

If we can ensure that we take the optimal action in every state, we can obtain the optimal policy. To do this, we need a definition of the value of taking an action in a certain state, which is called a Q-function (2.4). It differs from eq. 2.3 only in the fact that it starts with action a instead of the maximal action.

$$Q*(s,a) = R(s,a) + \gamma \sum_{s'} T(s'|s,a)V^*(s') \tag{2.4}$$

Since the optimal policy always takes the action with the maximal return, this allows us to define the optimal policy as

$$V^*(s) = \max_a Q*(s,a), \tag{2.5}$$

or

$$\pi^*(s) = \text{argmax}_a Q*(s,a). \tag{2.6}$$

Dynamic Programming

Dynamic programming (DP) is a collective term for a type of optimization algorithms that divides complex problems in a sequence of simpler subproblems. In the field of reinforcement learning, DP can be used to compute optimal policies, given a perfect model of the environment. The most well-known DP algorithms in RL are value iteration and policy iteration. Although DP algorithms have a great computational cost and the assumption of a perfect model, they also have a theoretical value. According to Sutton and Barto (1998) all of the feasible RL algorithms attempt to achieve the same effect as dynamic programming, but with less computation and without the assumption of a perfect model of the environment.

Value Iteration

Bellman (1957b) provided an algorithm, called value iteration, to solve the set of equations of a Markov Decision Process. If there are n possible states, then there are n Bellman Optimality Equations. The n equations contain n unknowns, the values of the states. We would like to solve this set of equations simultaneously to find the state values. However, because 2.3 contains a max-operator, the equations are non-linear. Therefore, we cannot solve the unknowns through linear algebra. But value iteration provides a solution through an iterative approach. It starts with arbitrary values and uses these to compute the right-hand side of 2.3, which is then used to update the left-hand side. This is repeated until it reaches an equilibrium. Let us denote $V_i(s)$ as the value of state s at the ith iteration. The update step, called the Bellman update, then looks like this:

$$V_{i+1}(s) \leftarrow \max_a R(s,a) + \gamma \sum_{s'} T(s'|s,a)V_i(s') \tag{2.7}$$

If the Bellman update is applied infinitely often, the algorithm is guaranteed to reach an equilibrium and the final state values must be solutions to the Bellman equations. In fact, it can be shown that they are also unique solutions, and that the corresponding policy is optimal. However, to obtain an optimal policy, it is not necessary for the state values to converge. It is sufficient for the state value of the converged solution to have the highest value when the max operator is applied.

Policy Iteration

Instead of creating the optimal policy through finding the optimal value function, policy iteration manipulates the agent's policy directly (Howard, 1960).

The algorithm starts with an arbitrary policy. The value of this policy is computed by taking the expected value of the set of linear equations in equation 2.2. Once the value of each state under the current policy is known, it can be computed whether the value of the policy improves if the first action is changed. If it can, the first action is changed into the action that yields the highest expected return under the current policy. This leads to the following update rule for every state

$$\pi'(s) \leftarrow \text{argmax}_a \left[R(s) + \gamma \sum_{s' \in S} T(s, a, s') V_\pi(s') \right] \tag{2.8}$$

This update is guaranteed to strictly improve the policy, thus when no further improvements are possible, the resulting policy is guaranteed to be optimal.

Q-Learning

The two drawbacks of DP are the computational cost and the requirement of a transition model, which might not always be available. There are algorithms that do not need a model of the environment. One of the most famous of these model-free algorithms is Q-learning (Watkins, 1989; Watkins and Dayan, 1992), due to its simplicity. However, this simplicity comes with a cost. Q-learning generally needs a lot of updates to approach the optimal policy.

Q-learning uses the above mentioned Q-function (2.4) to directly map states to actions by computing the values of actions in certain states instead of the state values. By plugging 2.5 into 2.4, we get a Q-function that is independent of state values,

$$Q*(s, a) = R(s, a) + \gamma \sum_{s'} T(s'|a, s) \ \text{max}_a Q(s', a). \tag{2.9}$$

This translates into the following update rule,

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R(s, a) + \gamma \text{max}_a Q(s', a) - Q(s, a) \right], \tag{2.10}$$

where α is the learning rate, which defines to what extend new information overrides old information.

Note that the update-rule depends on the maximal action in the next state, and not on the action that is defined by the used policy. It could update towards action one, but perform action two in that state. This defines Q-learning as an off-policy algorithm.

Also note that Q-learning does not iterate over states, it simply enumerates the Q-values and matches the new input to the existing state-action values. In principle, it could therefore handle continuous states. However, since the computer is a discrete system, some form discretization is needed to store the Q-values. To do this efficiently, one could use a function approximator.

2.3 Function Approximation

Q-learning was originally designed with discrete states and a lookup table of Q-values, which is called the Q-table. This lookup table contains the value of each action for all possible states. If the agent has a actions, and the environment has n dimensions, and each dimension can have m values, then the table would contain $n^m \times a$ entries. For example, the TORCS agent described later in this thesis uses 8 continuous inputs, and each dimension is subdivided into 10

bins. This would mean that the agent can perceive 10^8 possible states. The agent has 2 action dimensions available, which are subdivided into 3 and 5 bins, resulting in 15 discrete actions. The Q-table then contains 15 Q-values for each state, that is $15 \times 10^8 = 1.5$ billion table entries. Learning with a table of this size would take a very long time, especially since each entry must be visited many times before the Q-value becomes accurate. This makes the Q-table an infeasible representation for problems with continuous input dimensions, such as simulated car racing.

It is not necessary, however, to represent the Q-function as a lookup table. It could be approximated by any function approximator, be it linear or non-linear. Watkins (1989) already explored this option when he introduced Q-learning. When using Q-learning with a function approximation there is need for a different update rule. First, note that the Q-function can be approximated by

$$Q_t(s, a) = \vec{\theta}_t^T \vec{\phi}(s, a). \tag{2.11}$$

Here $\vec{\theta}_t$ is a parameter vector at time t, with a fixed number of adaptable real-valued components, and $\vec{\phi}(s, a)$ is a feature vector of state s and action a. The most common way to update this approximated Q-function is to use a gradient descent method (Sutton, 1988; Sutton and Barto, 1998). This method aims to reduce the mean squared error on the observed examples by adjusting the parameter vector $\vec{\theta}_t$ a small amount in the direction that would reduce the error on that example the most. That is, in the direction of the negative gradient of the example's squared error. There are several ways to define the update rule (van Hasselt, 2012). A common definition (Sutton and Barto, 1998) is given by

$$\vec{\theta}_{t+1}^T = \vec{\theta}_t^T + \alpha \, \delta t \, \nabla_{\theta_t} Q(s_t, a), \tag{2.12}$$

where δt is the temporal difference error, defined by

$$\delta t = r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q(s_t, a). \tag{2.13}$$

In the linear case, the gradient of the Q-function with respect to $\vec{\theta}_t$ is simply $\vec{\phi}(s, a)$. The update rule for linear function approximators is thus defined as

$$\vec{\theta}_{t+1}^T = \vec{\theta}_t^T + \alpha \, \delta t \, \vec{\phi}(s_t, a). \tag{2.14}$$

For non-linear function approximation the gradient descent update rule becomes somewhat more complicated, but that is beyond the scope of this thesis. For more information about the most common type of non-linear function approximation, artificial neural networks, the reader is referred to (Rumelhart and McClelland, 1986; Bishop, 1995; Bertsekas and Tsitsiklis, 1996).

Many real world problems require a function approximator to allow an RL algorithm to learn an acceptable policy within a reasonable time, but there is no consensus on which approximation is favorable under most circumstances. Linear function approximations have the advantage that they learn quickly, are easy to implement and easy to analyze. However, they need well chosen state features to be effective and not every problem can be described as a linear combination of state features. In contrast, non-linear function approximators are more robust to the choice of features, but are also more difficult to understand and to implement.

Pyeatt et al. (2001) compared the performance of a lookup table, a neural network, and several forms of adaptive decision trees as function approximators for Q-learning in simulated car racing. They concluded that the lookup table was not a feasible representation for simulated car racing, because the table would be too big to learn anything within reasonable time. Their experiments showed that the decision trees with a decision mechanism based on t-tests had the

best performance, and that the neural network failed to converge within the time limit. Despite being unreliable, the neural network had a decent performance, and whether or not the decision tree had a better performance, depended on the decision mechanism.

Although there has been some success with the combination of neural networks and reinforcement learning (Tesauro, 1995), they seem to perform poorly in the simulated car racing domain (Pyeatt and Howe, 1998; Barreno and Liccardo, 2003). The advantage of a neural network is that it can easily generalize towards new states. Thus, it can handle continuous inputs and solve larger problems than the lookup table. However, there are several reasons why neural networks are not very suitable for this problem. First, it assumes a stationary target function, whereas the targets in Q-learning (the approximated Q-values of future states) continually change during learning. Second, the neural network does not perform local updates, but adjusts the whole network during each update. Thus, the update of a value of one state may also affect the value of another state. This could result in improper credit assignment, causing sequences of updates to cancel each other out or the "forgetting" of Q-values of states that the agent has not visited recently. Therefore, it is easy for the network to become over-trained on a frequently visited part of the state space. Third, the non-linearity of the network gives it great expressive power, but also makes it hard to analyze and has the consequence that it likely converges to a local optimum (Barreno and Liccardo, 2003). However, the neural network is not guaranteed to converge in these kinds of problems at all and may come up with very ineffective driving behavior. Finally, there are many parameters to tune, to which the performance of the network is quite sensitive, while there is no standard setup. For more information about the combination of RL and neural networks, the reader is referred to (Bertsekas and Tsitsiklis, 1996).

The two most common linear and non-linear function approximators, respectively tile coding and artificial neural networks, have been tested in the context of this study. Initial results seemed to confirm the unreliability of neural networks and their sensitiveness to parameter tuning. Therefore the focus of this study was turned towards tile coding. As mentioned before, tile coding has been successful to a certain degree in simulated car racing (Barreno and Liccardo, 2003). Tile coding is easy to analyze, easy to implement and seems to be more suitable to approximate a changing value function.

Tile Coding

So, what is tile coding exactly? Tile coding is a method to choose the state features for linear function approximation, that is, to choose feature vector $\vec{\phi}(s, a)$ in equation 2.11. Tile coding was introduced by Sutton and Barto (1998), who were inspired by the Cerebellar Model Articulation Controller (CMAC) described by Albus (1971, 1981). Tile coding can be used to approximate state values, as in Sutton and Barto (1998), or to approximate a Q-values, as in this study. The latter only differs from the first in the fact that instead of one value for each state, there is a value for each action in each state. Let us therefore consider the simpler case, the approximation of the value function.

Linear function approximation with tile coding is easiest to depict as a linear combination of several overlapping lookup tables with a coarser resolution than the state space (see figure 2.2). More specifically, the state dimensions are partitioned into tiles (or features), and a partitioning of each dimension is called a tiling. The tiles act as receptive fields. That is, when the input value lies within the boundaries of a tile, it is considered active; otherwise it is considered inactive. Thus, at each time step there is only one active tile per tiling. If an active feature is defined as

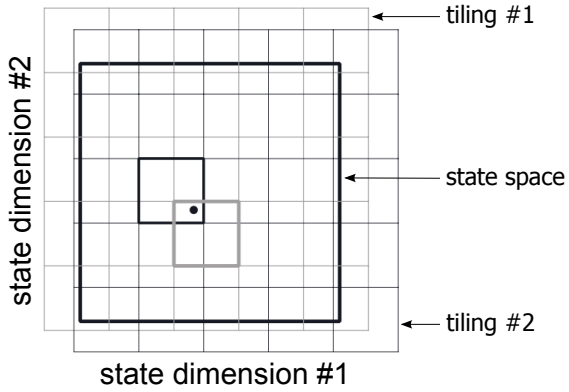


Figure 2.2: The approximation of a value function with tile coding for a two dimensional, continuous problem. The dot marks the actual state. However, note that all dots within the intersection of the two active tiles will be considered as the same state. Adapted from Sutton (1996).

having value 1, and an inactive feature is defined as having value 0, computational costs can be reduced. In that case, one only needs to consider the active tiles when computing the state value or updating the parameter vector. In other words, at each time step the number of considered features is the same as the number of tilings.

Tile coding generalizes the learned state values in two ways. First, all states that result in the same set of active tiles are considered as being the same state. That is, all states that lie within the same intersection of some set of tiles have the same value, whether that particular state was already visited or not. Second, since each parameter corresponding to an active tile is updated separately, the update of parameter vector $\vec{\theta}_t$ affects all the states that lie within the union of the active tiles. Note that for the same reason the learning rate should be adapted. Since each parameter is updated separately, the learning rate α should be divided by m, where m is equal to the number of active tiles. Otherwise, each update step would have m times as much impact as in the tabular case.

Finally, a tiling can be defined as a uniform grid with as many dimensions as the state space, as shown in figure 2.2, but it does not have to be a uniform grid. Depending on the desired form of generalization one could use stripes, log stripes or even more complicated patterns; as long as the tiling covers the entire state space (Sutton and Barto, 1998, Ch. 8.3).

2.4 Action selection

A reinforcement learning agent must learn to accomplish a task by means of trial-and-error. Therefore, it must explore the effects of its actions to learn what is the best action in any given situation. The agent must decide at each time step whether to choose the action with the highest estimated return or to choose another action. The first option *exploits* current knowledge to optimize the short-term return. The second option can be said to *explore* the values of the other actions, since it improves the agent's estimate of a perceived sub-optimal action. Picking the second option might induce a short-term penalty, but it might also allow the agent to find a better action, which can be exploited in the long run. Since only an agent that already follows the optimal policy can choose only exploitative actions, an agent generally has to balance these

options. The problem of finding this balance is known as the 'dilemma of exploration and exploitation' (Sutton and Barto, 1998).

If one is training an actual car, it would be desirable that the car does not crash while training, since it would cost a lot to repair the car. In that case, the balance would be set more towards exploitation. However, a simulation like TORCS can be restarted with the push of a button. This allows for more exploration.

This action selection dilemma gets an extra dimension in simulated car racing, because the speed of the agent changes the cost of an exploratory action. Indeed, the effects of a short, sharp turn to the right on a straight track have less implications when the agent drives 20 km/h than when it drives 200 km/h.

Also, the agent first needs to exploit its knowledge on low speeds before it can explore its actions at high speeds. That is, many exploratory actions at high speeds lead to a crash and before the agent can try a new action at that speed, it must accelerate to that speed without crashing or getting slowed down by exploratory actions at lower speeds.

Another aspect of the balance between exploration and exploitation is whether the agent is judged on its performance while it is still learning, so called *online learning*. In that case the balance would lie more towards exploitation, because it must accumulate rewards while adjusting its policy. However, in this study we will assume *offline learning*. In other words, the agent's training session is separated from its performance evaluation. In that case the agent can explore as much as it wants as, as long as it has learned a good policy at the end of the training period. Still, even when the agent learns offline, it is good to have quite some exploitation to reach interesting parts of the state space.

The most popular action selection mechanism is ϵ-greedy (Watkins, 1989; Sutton and Barto, 1998). Instead of always choosing the action with the highest expected reward, the agent chooses a random action with a small probability ϵ (epsilon), independent of action values. That is,

$$\pi(s_t) = \begin{cases} \text{random action } a \in A(s_t) & \text{if } \rho < \epsilon \\ \text{argmax}_a \, Q(s_t, a) & \text{otherwise} \end{cases} \tag{2.15}$$

where $0 \leq \rho \leq 1$ is a uniform random number drawn at each time step. It is popular because it easy to implement, requires little computational resources, and has only one parameter to tune. Though tuning ϵ requires little knowledges, there are also no solid rules for choosing ϵ, which can make the process of parameter tuning quite tedious. However, it was shown that ϵ-greedy can achieve near optimal results when tuned correctly (Vermorel and Mohri, 2005).

There are many more ways to implement exploration, such as Soft Max (Sutton and Barto, 1998), which depends on the relative Q-values and a temperature parameter that decreases over time, or Value-Difference Based Exploration (Tokic, 2010), which computes an adaptive ϵ for each state based on the temporal difference error. Instead of implementing an explicit exploration/exploitation choice, one could also slowly adjust the Q-values of actions that have not been tried for some time to make those actions more interesting, such as in the dual control algorithm of Dayan and Sejnowski (1996). For more details on exploration and action selection in model-free algorithms, the reader is referred to (Schmidhuber, 1991; Thrun, 1992; Wiering, 1999).

In this study we propose an extension to ϵ-greedy to guide the exploration to interesting parts of the state-action space. We introduce the parameter η (eta), which works the similar to ϵ. However, instead of a random action, this probability is attached to an informed action. This informed action is generated by a simple heuristic that exploits domain knowledge (see section 3.2). The action selection mechanism then looks like this,

$$\pi(s_t) = \begin{cases} \text{informed action } a \in A(s_t) & \text{if } \rho < \eta \\ \text{random action } a \in A(s_t) & \text{if } \rho < \eta + \epsilon \\ \text{argmax}_a \, Q(s_t, a) & \text{otherwise} \end{cases} \tag{2.16}$$

where $0 \leq \rho \leq 1$ is a uniform random number drawn at each time step. Every time step there is a probability η of taking a heuristic action, a probability ϵ of taking a random action, and a probability $1 - \eta - \epsilon$ of taking a greedy action.

The idea behind this extra parameter is that the agent is guided towards interesting parts of the state-action space by exploiting domain knowledge instead of action values or the temporal difference error. Using some of the probability mass for a heuristic action results in more exploration than a greedy action and more exploitation than a random action. It might therfore result in a better balance between exploration and exploitation than ϵ-greedy.

One might question the use of random exploration if there is a heuristic available. Indeed, at the start of the training phase a heuristic should improve the agent's policy more than random exploration. However, the heuristic also limits the exploration of the agent, since the heuristic always recommends the same action in a certain state. It is not guaranteed that, given enough training time, each action will be tried infinitely many times, which is necessary for Q-learning to converge. Moreover, the heuristic in this study is more used as a guide than as a teacher. Therefore, the random exploration is necessary to learn to improve upon the heuristic policy.

For example, say that the heuristic steers towards the center of the track. By occasionally doing that, the agent might learn that steering towards the center of the track is a good thing to do. Steering towards the track center is a good start, however the optimal race line deviates from the track center and by more exploration it should learn over time that certain little deviations from the track center are more rewarding than staying at the center. It would be easier to learn this by trying a random action every once in a while than by depending on the gradually changing Q-values.

2.5 Learning from Examples

The proposed action selection mechanism uses a heuristic to generate an appropriate action for a given state from which the agent learns. This could be seen as generating an example. One could have an agent train on these examples and learn to directly imitate the given policy by learning to choose the same action in any state. This would be a form of *supervised learning* and seems too restrictive for this kind of problem. Since the goal is to imitate the given policy, which is treated as the optimal policy, it is penalized for deviating from this policy. This means that the agent could improve on the existing policy only by accident. However, if the optimal policy was indeed available, there would be no need to train the agent.

Supervised learning does not necessarily give poor results, though. Athanasiadis et al. (2012) were able to train a neural network with an almost equal performance to COBOSTAR, which is quite

a notable achievement. They accomplished this by splitting the training period in four separate sessions and using examples from three increasingly complex agents, and finally COBOSTAR itself. Using examples from simple agents to learn simple behavior and progressively changing to more complex behavior seems to result in better policies than attemps to combine RL with neural networks (Pyeatt and Howe, 1998; Barreno and Liccardo, 2003). However, the setup of supervised learning would never allow that agent to improve upon the policy of COBOSTAR. Unless, ofcourse, the agent would fail to copy the policy of COBOSTAR and get stuck in a local optimum, which by chance resulted in a better racing policy.

In this study, the agent may copy an action from an existing policy, but it draws its own conclusions about whether choosing that action again would improve its policy. If the heuristic is viewed as an agent, it could be seen as a teacher sitting beside the driver and telling it which action it should perform in a certain state. However, after performing this action, this state-action pair is updated as if it were the agent's own choice of action. This is called *implicit imitation learning* (Price and Boutilier, 2003). One could say that every time the RL agent performs a heuristic action, it is imitating the heuristic agent.

Imitation learning is a common practice in the field of robotics and optimal control, such as for learning motor control. It has an overlap with supervised learning, in the sense that the goal is to imitate the policy of a teacher. However, the agent does not learn from its deviation from the desired output, but from the effects the output has in the environment. Imitation learning allows some exploration, which might lead the learning agent to improve upon the policy of the teacher. A typical example is the training of a robot arm to follow a trajectory that was demonstrated by a human (Atkeson and Schaal, 1997). However, there are many applications, resulting in various aliases, such as 'learning by demonstration' and 'apprenticeship learning'. For an overview of imitation learning, the reader is referred to (Schaal, 1999; Bakker and Kuniyoshi, 1996).

Unlike (Price and Boutilier, 2003), most studies in imitation learning involve a human demonstation instead of a demonstration by another agent. Perhaps this is due to the fact that most algorithms need an expert. Since they lack an expert agent, they turn to humans. As opposed to imitation learning, the goal of the proposed action selection mechanism is not to replicate the policy of a teacher. As in (Smart and Kaelbling, 2000), the teacher is simply there to point the agent to interesting parts of the state-action space. As a result, the teacher does not have to be an expert. This is convenient, since there would be no need to train the agent if there was already an expert agent available.

TORCS 3

This chapter describes how to formulate the TORCS environment as a MDP, including implementation details. Section 3.1 describes TORCS in terms of states, actions and rewards. Section 3.2 describes the implementation of a reinforcement learning agent in TORCS.

3.1 TORCS as an MDP

A Markov decision process consists of the following components: states, actions, a reward model and a transition model. The question is now how to map the information in TORCS to these components. The simulation provides a transition model, but it is highly complex and contains many transitions. It is infeasible to use a dynamic programming algorithm to iterate through these transitions or to build a representation of this model. Therefore, we will sample from the existing transition model and use a model-free algorithm instead.

States

One could take the whole sensor model (table 8.2) as the state description. It seems to describe all that the agent needs to know, but it contains a lot of unnecessary information, especially if the agent only has to drive by itself. Table 3.1 contains a concise state description that should be sufficient to model each state of the world during the race. The state dimensions are the speed along the track - or longitudal speed-, the position on the track - or latitudal deviation to the track axis-, the angle with respect to the track axis (normalised from $[-\pi, \pi]$ to [-1,1]) and five distance sensors that measure the distance to the edge of the track. Note that the track may contain a gravel trap or a bank of grass, which means that the edge of the track might be further away than the edge of the actual road. The 20° inputs are not taken directly from sensors 7 and 11, but computed as an average over sensors 6, 7, 8 and 10, 11, 12, respectively, to account for noise. In this study, we are not interested in fuel usage or damage. Therefore, this functionality will be switched off and not incorporated into the state description.

Table 3.1: State description

State dimension	**Corresponding input from the sensor model**
Longitudal speed	speedX
Latitudal deviation	trackPos
Angle w.r.t. the track axis	angle
Distance sensor at $-40°$, $-20°$, $0°$, $20°$, $40°$	track [5, 7, 9, 11, 13]

Note that these dimensions are all continuous, which means that the number of possible states is infinite. Therefore, we will need some kind of function approximation. We will get back to that in section 3.2.

Actions

There are four action dimensions available in TORCS (excluding the meta-action 'restart'): accelerate, brake, gear and steer. Since braking is simply a negative acceleration, we shall view this as the negative side of the same dimension.

The basic controller of the SCR framework (a.k.a. SimpleDriver) shifts gear according to the rpm of the car's engine. Butz et al. (2011) mention that the engine works best if the full spectrum of its gears is used. The values for shifting down were tuned manually. Onieva et al. (2012) also tuned the rpm-values manually and report that minor deviations do not lead to a change in performance. It seems that there is little to be gained from automatically optimizing the policy for shifting gears. If this action would be included in the representation of the agent, it would only further complicate an already difficult problem. Therefore, we choose to use the gear shifting policy of the basic controller, with the values of (Onieva et al., 2012), which are given in table 3.2.

Table 3.2: Gear shifting values

Gear	1	2	3	4	5	6
Shift up	8000	9500	9500	9500	9500	0
Shift Down	0	4000	6300	7000	7300	7300

This means that there are two continuous action dimensions left, acceleration and steering. To apply Q-learning a discretization of these dimensions is needed. Initially the agent was given seven steering actions: -1, -0.5, -0.1, 0, 0.1, 0.5, and 1. This includes actions for small deviations and actions with a larger impact to pass sharp curves. It was thought that it is necessary to represent the whole input space. But preliminary results showed that the agent learned to prefer frequent small actions, which is easier to learn, above well-timed extreme actions. So, to reduce the dimensionality, the actions 1 and -1 were removed. The basic controller seemed to prefer actions of approximately 0.05 and -0.05, but adding those actions did not improve the performance of the RL agent in the initial experiments.

The values for acceleration were tuned manually as well. Since human players tend to balance between full acceleration and no acceleration at all and since the time resolution is quite high (one action per 20ms), it seemed unnecessary to give the agent a high resolution in the acceleration dimension. The agent was given three values: 1, 0 and -1. Adding -0.5 and 0.5 did not seem to change the performance. No further evaluations were done. The resulting discretization is shown in Table 3.3.

Table 3.3: Discrete actions

Steer↓	Accelerate (1)	Neutral (0)	Brake (-1)
0.5 (left)	0	1	2
0.1 (left)	3	4	5
0	6	7	8
-0.1 (right)	9	10	11
-0.5 (right)	12	13	14

Rewards

The reward is defined by the traveled distance between two states (in m). This should be enough for the agent to learn that driving fast is good, but leaving the road - and thus ending an episode - is bad. The episode ends because the off-road behavior is irrelevant for an agent that drives on a track by itself, the optimal policy would never reach those states. Furthermore, it stops the positive reward signal and thus, discourages the agent to go to the edge of the road.

Even though this might be enough to have the agent learn the optimal policy, the required number of updates might be reduced by adding two components to the reward signal. The first is a reward of -1 for leaving the road, serving as an extra penalty for behavior that ends the episode. The second is a reward of -2 for standing still (traveled distance $< 0,01$m), because it is obvious that standing still is undesirable behavior. Together, one could formulate these rules as 'drive', 'drive as fast as possible', and 'but don't drive off the road'. Note that the added components do not affect the behavior of the optimal policy, because optimizing 'drive as fast as possible' excludes actions that lead to standing still or leaving the road.

3.2 Implementation

This section describes the implementation of the above mentioned MDP description and the architecture that was needed to facilitate the reinforcement learning and connect everything to TORCS.

The controller consists of four parts: the client, which facilitates the communication channel with the server, the driver, which processes the input and computes the output to the server, the learning algorithm, which trains a function approximator, and an interface that connects the algorithm and the driver. Figure 3.1 gives an overview of the classes involved.

To facilitate the extension of this controller with other learning algorithms, there is an abstract driver class with the parameters, variables and functions that any driver would need, including for example the static parameters that are needed for the heuristic. When a new algorithm is added, it can be implemented as a new driver that inherits the basics from this abstract class, which can then be tuned to the needs of the algorithm.

Similarly, each algorithm would need a specific interface to connect the learning algorithm with the driver. Again, there is an abstract interface class that describes most of the functions. We just need different instantiations to call different learning algorithms.

This section regularly refers to a Q-table, even though with tile coding there is no actual lookup table of Q-values. As mentioned before, each tile contains a value for an action, and the values of all active tiles sum up to the approximated Q-value. However, these tilings are implemented as one datastructure (described in 3.2). This datastructure shares the purpose of a Q-table, but it works differently, and therefore, has no name. For convenience, this is also referred to as a Q-table.

The Game Loop

To understand how the agent works, it is important to know what the game loop looks like. Each game tick the SCR server asks the driver to return an action by calling its `drive()`.

The drive function consists of two parts: one that checks whether the agent is *stuck* or outside the road and one that handles action selection and learning.

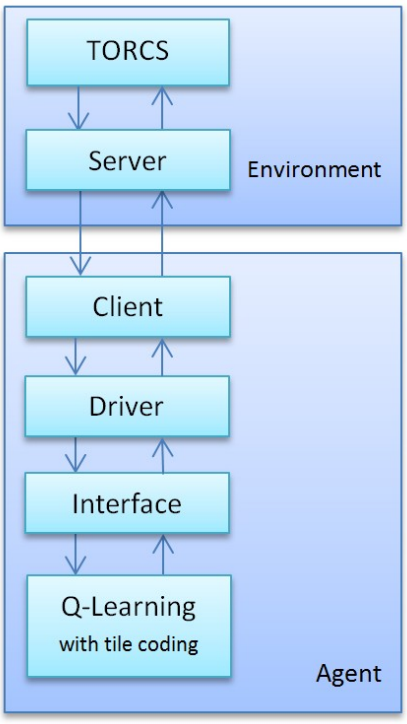


Figure 3.1: Abstraction of the class diagram. Arrows represent the flow of information.

If the car's angle is larger than 45 degrees, it is considered *stuck*. If it is *stuck* for more than 25 game ticks and it drives slower than 10 km/h, the episode ends. Every time the agent is not *stuck*, the stuck timer is reset to zero.

Then, whether the episode ends or not, the learning process is started. First, the reward of the previous state-action pair is computed as described in 3.1, and sent to the learning interface. Next, the current state is converted to the state description of table 3.1 and sent to the learning interface. Finally, the learning interface performs the update.

Since the agent is already processing the current state, the update-function does not only perform an update, but also produces the next action given the current state. The agent uses a random number to choose how to generate an action, based on its Q-table, the heuristic or a random number.

Before the Driver-class receives the action, the action is converted from the discrete number used by Q-learning to a two-dimensional continuous action according to table 3.3. The driver combines this action with the actions that are performed automatically (like switching gear) and sends this combined action to the client, which actually performs the action in the environment.

Before the combined action is sent to the client, the driver checks whether the desired number of updates has been reached. If so, the Q-values belonging to each tile of each tiling are stored in a file, so they can be reloaded later. If there are more experiments, the restart flag of the combined action is activated, otherwise the program ends.

Learning Interface

The primary function of the learning interface is to do action selection and call the update function of the learning algorithm.

Action selection is based on a random number between 0 and 1. If the number is smaller than η, the agent performs an action based on its heuristic. Otherwise, if the random number is smaller than the sum of η and ϵ, it performs a random action. This facilitates the use of the same random number to pick a random action with probability ϵ. If neither condition is fulfilled, the agent performs a greedy action based on its Q-table.

If it is not the first time step, the learning interface calls the algorithm's `update()` with the previous state, previous action, reward and current state, an end of episode flag, the learning rate and the γ. Then, the current state and action are copied to the variables for the previous state and action. Obviously, if it is the first time step there is no previous state and previous action. In that case, the values are copied without update.

The learning interface is also used for saving and loading Q-tables, this is implemented through standard file streams for reading and writing plain text. To make sure that the loaded Q-table is compatible with the current settings the learning interface also writes the number of tiles per state dimension and the number of actions to the file.

Tile Coding

Generating Tilings

During initialisation, the tile coding object reads the tilings from a file. The edges of each input dimension are stored in a separate vector, except for the distance dimensions, which share the same edges. The input file is scanned for the following words: `speed`, `pos`, `angle`, and `dist`. It is assumed that the rest of the line contains the related edges, which is parsed with a separate function. Note that the order in which the dimensions are presented does not matter. However, it is assumed that the order of the edge dimension corresponds with its tiling, that is, the second occurrence of speed edges in the file corresponds with the second tiling. The definition of two tilings in a text file typically looks like this:

```
speed -10 0 25 50 75 100 125 150 200 250 320
speed -10 5 30 55 80 105 130 160 210 260 320

pos -2 -0.90 -0.65 -0.4  -0.15 0.1  0.35 0.60 0.85 2
pos -2 -0.85 -0.60 -0.35 -0.1  0.15 0.40 0.65 0.90 2

angle -2 -0.90 -0.65 -0.4  -0.15 0.1  0.35 0.60 0.85 2
angle -2 -0.85 -0.60 -0.35 -0.1  0.15 0.40 0.65 0.90 2

dist 0 0.025 0.04 0.05 0.15 0.20 0.30 0.40 0.50 0.65 1
dist 0 0.030 0.05 0.10 0.15 0.25 0.35 0.45 0.55 0.70 1
```

Representing Q-values

In this implementation, the tiles with the partial Q-values are stored in a hash map (specifically, the `map` of the C++ Standard Library). A hash map works like a dictionary, so the cost for each lookup is independent of the number of elements in the table. Also, it allows arbitrary insertions of key-value pairs, as long as they are unique. This is ideal for the Q-table, which will contain

a lot of elements, because there is no need to define all states beforehand or to loop through all defined states each state visit. This saves both memory and computation time.

When using tile coding, the Q-value of a state-action pair consists of the values of the contributing tiles. These values are stored separately and summed up when necessary. Therefore, this data structure is not officially a Q-table. However, since it has the same structure and purpose, one could still refer to it as the Q-table.

Generally, objects cannot be used as keys for a hash map, for reasons beyond the scope of this text. However, an object is a convenient way to bundle the information that describes a state. Therefore, when accessing the Q-table, the state-action pairs must be encoded into a variable type that is allowed to be used as a key to the hash map.

The used encoding is a string of nine numbers, which begins with the number of the tiling, followed by the tile numbers of each input dimension. One such string is created for each tiling. Because each tiling classifies a state differently, the values per tiling must be saved separately. After encoding, a state is represented as a vector of strings, with a size equal to the number of tilings.

Each string of the state is connected to the action by means of a `pair` (part of the C++ Standard Library), resulting in a `pair<string,int>`. Like a string, any `pair` can be used as a key. But the advantage is that this representation keeps the distinction between states and state-action pairs. This facilitates the creation of extra debugging information, like the number of visits to a state.

At the start of each execution of `update()`, a separate function is called to generate this representation. To solve the problem of classifying each input, this function calls the same classification function for each input and each tiling, but each time with a different vector of edges of tiles. To prevent errors elsewhere, it should be checked that a state can be classified with each tiling.

Retrieving the Q-value

To retrieve the Q-value of a state-action pair, the values of the active tiles must be summed. This framework uses a separate function for this, with as input parameters the vector of strings of the previous state and the integer that defines the previous action. The function creates a `pair<string,int>` for each string in the vector, which is used to search for the value of the state action pair in the Q-table. All found values are summed and returned. If a state is visited for the first time, the key does not return a value. In that case, a new Q-table entry is created, and by default has the value 0 assigned to it. Computing the largest Q-value of a state or the action with the largest Q-value works similarly, except that instead of picking one action, the function loops through all possible action integers. In fact, it could use the above mentioned function to do the actual looking up and summing. All values are stored in a vector, and either the max or the argmax is returned.

Updating the Q-table

The update of the Q-table happens in the tile coding class. First, the current state and the previous state are converted into a vector of strings. This vector of strings corresponds to the list of active tiles. A partial state-action pair is created for each string in the vector, which is used to access the Q-table. The Q-value of the previous state-action pair is then computed by looking up the partial state-action pairs and summing their values.

If an episode has ended, the temporal difference error is computed as the difference between the reward and the computed Q-value of the previous state. The partial state-action pairs are

used again to access the Q-table. Subsequently, *learning rate* × *TD-error* is added to the value of each active tile.

If an episode has not ended, the maximal Q-value of a state-action pair with the current state is computed, and for each encoded state-action pair the update is applied as in equation 2.14. Note again that each update is done with a learning rate that is divided by the number of tilings, because the effect of the learning rate is multiplied by the number of tilings. This is due to the fact that the value in each active tiling is updated separately, but summed up when computing the Q-value.

Equal Q-values

It is possible for multiple actions to have the same value, for example when a new state is explored and all values are unknown (and all have default value 0). Then, the agent must make a decision based on something other than the value. It could pick an action based on some heuristic, or simply the first action that popped up. This implementation uses a random number to choose between the actions of equal value.

Heuristic Exploration

The agent's exploration is guided by a simple heuristic, SimpleDriver (the basic controller of the SCR framework). This heuristic drives as fast as possible, up to 150 km/h, and tries to minimize its deviation from the track axis. Specifically, the heuristic for steering is given by algorithm 1 and the heuristic for acceleration is given by algorithm 2.

Algorithm 1 SimpleDriver's steering heuristic

```
1:  function getSteer(CarState cs)
2:      read currentAngle, currentTrackPos
3:      steerLock ← 0.785398                          ▷ which corresponds to 1/4 π
4:      steerSensitivityOffset ← 80
5:
6:      %Steering angle is computed by correcting the actual car angle w.r.t. to track
7:      targetAngle ← (currentAngle - currentTrackPos × 0.5)
8:
9:      %At high speeds, reduce the steering command to avoid loosing control
10:     if currentSpeed > steerSensitivityOffset then
11:         return targetAngle / (steerLock × ( currentSpeed - steerSensitivityOffset))
12:     else
13:         return targetAngle / steerLock
14:     end if
15: end function
```

One could use a more sophisticated heuristic, for example a hand-tuned policy or one of the drivers mentioned in section 1.2. However, this gives the agent a lot of information, which is not congruent with the goal of training an agent with very little prior knowledge.

To adapt the heuristic to the action space, the continuous actions must be discretized. Consider steering to the left, if the steer value is larger than 0.33, the agent will steer with 0.5, if the steer value is between 0.33 and 0.02, the agent will steer with 0.1, and if the steer value is between 0.02 and -0.02, the agent will not steer. The same holds for steering to the right. The acceleration

Algorithm 2 SimpleDriver's acceleration heuristic

```
1: function getAccel(CarState cs)
2:     maxSpeedDistance ← 70
3:     maxSpeed ← 150
4:
5:     %Get the reading of distance sensors at +5, 0, and -5 degrees w.r.t. the car axis
6:     rightSensor ← read trackDistance(5)
7:     centralSensor ← read trackDistance(0)
8:     leftSensor ← read trackDistance(-5)
9:
10:    %If the track is straight and far enough from a turn, it goes to max speed
11:    if (centralSensor > maxSpeedDistance) or
12:      (centralSensor ≥ rightSensor and centralSensor ≥ leftSensor) then
13:        targetSpeed ← maxSpeed
14:    else
15:        %If approaching a turn on the right
16:        if rightSensor > leftSensor then
17:            %Compute approximately the "angle" of the turn
18:            h ← centralSensor × sin(5)
19:            b ← rightSensor - centralSensor × cos(5)
20:            sinAngle ← b × b/(h × h + b × b)
21:            %Estimate the target speed depending on turn and on how close it is
22:            targetSpeed ← maxSpeed*(centralSensor * sinAngle/maxSpeedDistance);
23:        %If approaching a turn on the left
24:        else
25:            %Compute approximately the "angle" of the turn
26:            h ← centralSensor × sin(5)
27:            b ← leftSensor - centralSensor × cos(5)
28:            sinAngle ← b × b/(h × h + b × b)
29:            %Estimate the target speed depending on turn and on how close it is
30:            targetSpeed ← maxSpeed × (centralSensor × sinAngle /maxSpeedDistance)
31:        end if
32:    end if
33:
34:    %The accel/brake command is exponentially scaled w.r.t. the difference between target
35:    %speed and current one
36:    return 2 / (1 + e^(currentSpeed - targetSpeed)) − 1;
37: end function
```

dimension is divided into three equal parts. The resulting discretization is shown in table 3.4. Note that the state space of the heuristic is not discretized.

Table 3.4: Discretization of the heuristic. The variables s and a stand for the value of the heuristic in respectively the steering and the acceleration dimension.

Steering $\downarrow$, Acceleration $\rightarrow$	$a >= 0.33$	$0.33 > a >= -0.33$	$-0.33 > a$
$s >= 0.33$	0	1	2
$0.25 > s >= -0.02$	3	4	5
$0.02 > s >= -0.02$	6	7	8
$-0.02 > s >= -0.25$	9	10	11
$-0.33 > s$	12	13	14

The discretization of the action space has an enormous effect on the performance of the heuristic. With continuous actions, the heuristic driver can finish the track of this study in 67,9 seconds. After the discretization, the heuristic driver crashes after 21,9 seconds.

An analysis of the heuristic shows that it generally performs many small actions between 0 and 0.1 to stay around the track axis. But in the curves, it has to steer sharply to remain on track. Since the heuristic is not informed about the coarse, discrete actions, it lacks the timing to perform these actions well.

The heuristic driver could be prevented from crashing if the speed is capped to a maximum of 120 km/h. However, this would limit the speed exploration of the learning agent severely. It would have to rely on ϵ-greedy for the exploration of high speeds, since the heuristic would tell the agent to stop accelerating. When the speed is not capped, the heuristic may not be able to finish a lap, but it can still tell the learning agent to accelerate at speeds higher than 120 km/h.

Experimental Setup 4

The experiments will consist of agents trained with several parameter combinations, which will be compared based on the performance during a separate evaluation run after training. This chapter describes several choices relevant to the setup of the experiments. The general design decisions concerning TORCS will be described first. This is followed by the reasoning behind the chosen values of the edges of the tilings, the model parameters, the number of updates and the algorithm parameters. The model parameters consist of the car, the track and γ. The algorithm parameters consist of the learning rate α, the heuristic rate η and the exploration rate ϵ. The actual experiments are described in Chapter 5.

4.1 Design Decisions

Several design decisions were needed to adapt TORCS to the intended experiments. The most important is the decision to have the agent learn to drive by itself. There are no opponents on the track. Of course, this does not prepare the agent for competitive racing. But we want the conditions to be the same as the warm-up round. Competitive behavior could be learned separately, as in (Loiacono et al., 2010b).

Another important decision is to stop the episode when the agent drives off the road. The main reason for this is the fact that the sensor model becomes unreliable when that happens. This introduces uncertainty about the agent's actual state and could cause the agent to receive impossibly large rewards. Because the measurement of the position becomes unreliable as well, the episode ends when the center of the car passes 95% of either side of the road.

The resolution of one action per 20 ms might seem too fine-grained. It might be difficult for the agent to notice the effects of different actions in a certain state. However, initial tests showed that a resolution of one action per 200 ms is too coarse for an agent to respond to new states at high speeds. Therefore, the action resolution remains 20 ms. A possible way to handle this fine resolution would be to combine normal actions with so-called *options*, actions that are taken over a larger period of time, see (Sutton et al., 1999). However, that is beyond the scope of this thesis.

4.2 Tilings

The implementation of the tilings consists of 5 tilings with $5^4 = 625$ tiles each. These values were chosen to minimize computational costs, while retaining a decent resolution for decision making. The tilings are not canonical, the offset and tile sizes differ for each tiling to get a better resolution at the parts of the input space that seem important after initial experiments.

However, this resolution difference is limited by the fact that all tilings should cover the entire input space with just a few number of tiles per dimension.

The exact values of the edges are described in Appendix B. Note that all tilings cover more than the entire input space. This is a practical consideration, since it might prevent the program from crashing, should some noise in the input occur. One is not bound to define tilings that cover the input space, but having all tilings active in each state simplifies the representation and the updates. Moreover, tile coding is quite robust to the definition of the edges of the tiles, as long as each tiling covers the entire input space (Sutton and Barto, 1998). The tilings could even be defined by a random strategy. However, in this experiment a deterministic strategy was used.

The first tiling of the speed dimension contains the edges -10, 0, 50, 120, 180, and 320. These values reflect speeds in km/h and were chosen based on the idea that the agent should be able to discern between driving forward and driving backward, between city speeds and highway speeds, and between highway speeds and racing speeds. The first and last edge were chosen such that the agent would never exceed these values. The maximal speed of the car is approximately 300 km/h on a track without sharp curves. The other tilings of the speed dimension have regular offsets with regard to the first tiling. These off-sets were chosen such that there is a finer resolution at lower speeds, than at higher speeds.

The position on the track and the angle with respect to the track axis have the same tilings, because they have the same input space (between - π and π radians, scaled to [-1,1]) and both need higher resolution around 0. The tilings were chosen such that there is a lot of overlap at various values around zero. This should allow the agent to fine tune its behavior around the track axis.

The tilings of the distance sensors were chosen such that the edges are distributed almost uniformly along the input space. The values correspond to meters towards the edge of the track. The first tile of the first tiling is kept small on purpose (between -1m and 10m), to have a feature for being close to the edge. The other edges of the first tiling were chosen to have the values 50m, 100m, 150m, and 200m. The edges of the other tilings have an offset of 10m with regard to the previous tiling, except for the first and the last edge, which remain the same for all tilings.

4.3 Parameters

There are several RL parameters that can be tuned in this experiment: the number of tiles, the number of updates, the learning rate, the exploration, gamma, and the rate of the additional heuristic. There are two parameters of TORCS that are relevant to the experiment, the used car model and race track. This section explains the choice for the values of these parameters.

Car & Track

The car model is 'car1-trb1', which is the standard car model of the SCR framework. It has 6 gears and can drive up to ∼320 km/h on a straight track. The agent drives on the track 'E-track5' (see figure 4.1). This track was chosen for its simplicity. It contains both curves to the left as to the right, which makes it a little more challenging than an oval track. The curves are quite smooth, therefore this track does not require an advanced policy for a good lap time. More importantly, it has no large areas outside the track, such as gravel strips, that might give ambiguous information about the edge of the road to the sensors. Such areas can confuse the agent, because they have different dynamics, but cannot be easily discerned from the road. So, for example, the agent would then have to learn from its position sensors that there is a difference

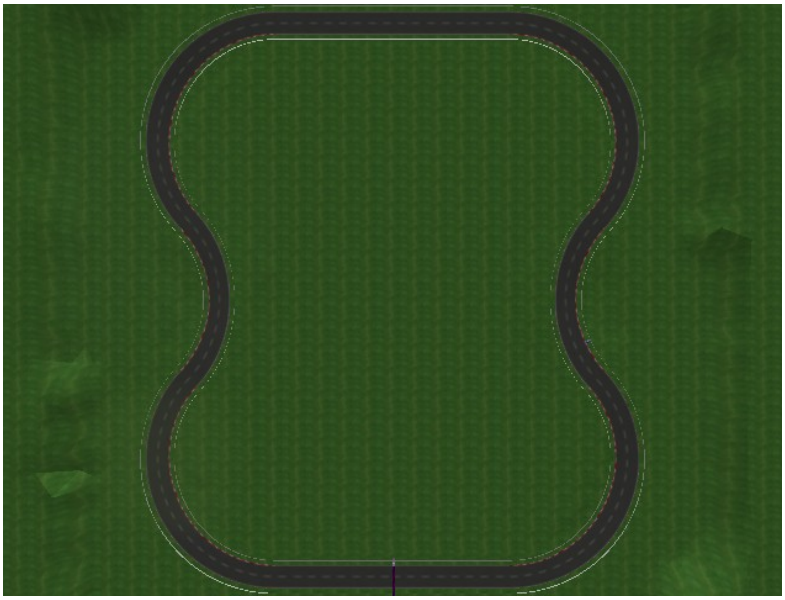


Figure 4.1: A top-down view of E-track5.

between a lot of space in front on a straight part of the track and a lot of space in front in a curve with a gravel strip.

Gamma

Instead of trying to find a good value for γ with experiments, we can estimate a reasonable value and thereby reduce the number of experiments. Let us assume that we want the reward of the next x actions to weigh as much as the return of all states after that. This means that we want

$$\sum_{i=0}^{x-1} \gamma^i = \sum_{i=x}^{\infty} \gamma^i. \tag{4.1}$$

Then we can solve the above mentioned equation as follows,

$$\frac{1-\gamma^x}{1-\gamma} = \frac{\gamma^x}{1-\gamma} \tag{4.2}$$

$$2\gamma^x = 1 \tag{4.3}$$

$$\gamma = 2^{-\frac{1}{x}} \tag{4.4}$$

Manual tests of driving towards curves seemed to indicate that the two seconds after the action are most relevant to the decision. Then, we want x to be $2/0.02 = 100$. Plugging this into 4.4 gives us $\gamma \approx 0.993$, which will be rounded down to 0.99. This value results in a lookahead of 1.38s, or 69 decisions, which is still quite some time for a race car.

Number of Updates

To determine the necessary number of updates, the initial results of the experiments described in section 5.1 were evaluated. The combination of $\eta = 0.2$ and $\alpha = 0.1$ seemed to be part of the

set of combinations that yielded good results. Therefore these values were picked for ten training sessions of two million updates. During these training sessions, the Q-table was stored every 150.000 updates. Each resulting agent drove an offline run of five minutes, which corresponds to 15.000 time steps. The resulting average lap times are shown in figure 4.2. Not every training run produced a successful policy, that is, a policy that could finish a whole lap. The resulting lap times of 300 seconds (the maximum duration of the trial) can be seen as noise, and are therefore removed from the data. Table 4.1 shows the success rates of the runs.

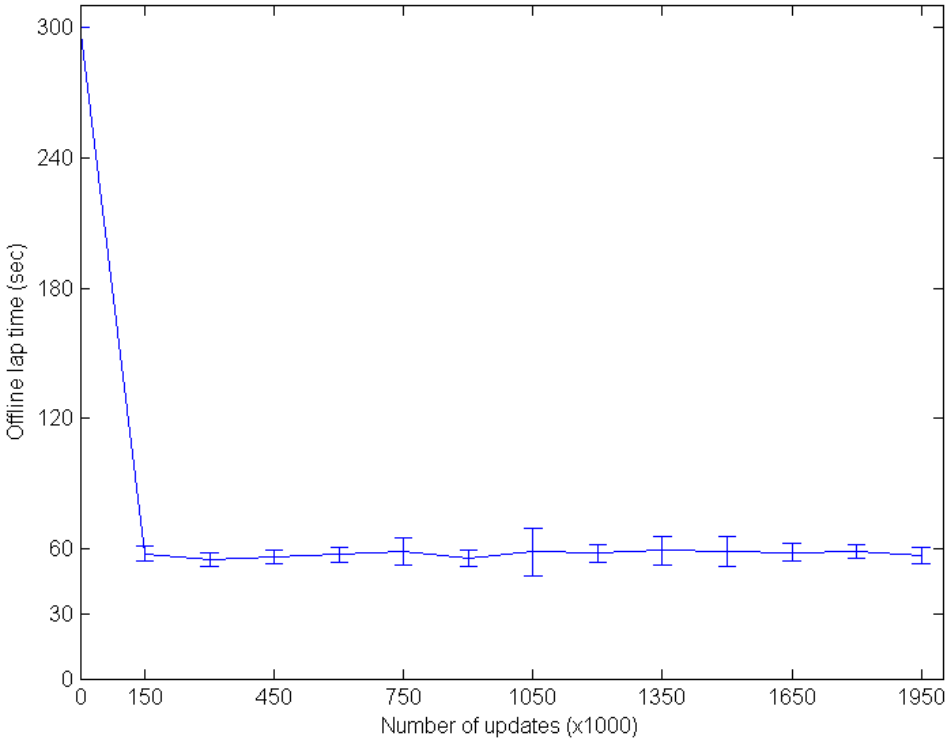


Figure 4.2: Average lap times in a five minute offline run of an agent that learns over time, with parameters $\eta = 0.2, \alpha = 0.1$. The results of agents that were not able to finish a lap are left out. The corresponding success rates are shown in table 4.1.

Table 4.1: The success rates of creating a policy that can finish at least one lap in an offline run, with parameters $\eta = 0.2, \alpha = 0.1$. The corresponding average lap times are shown in figure 4.2.

# of updates (x1.000)	success rate
0	0%
150	20%
300	50%
450	90%
600	100%
750	80%
900	100%
1050	100%
1200	80%
1350	90%
1500	90%
1650	100%
1800	90%
1950	90%

The lap times do not significantly decrease after 150.000 updates, as can be seen from the figure. What does change is the success rate. Where the success rate is only 20% at 150.000 updates, it has already increased to 90% at 450.000 updates. However, 450.000 is very little for a RL-problem of this complexity, and this high success rate might be coincidence due to the small simple size. It does no harm to give the agent more training time, but it might increase the likelihood of a successful policy. So, to have some margin, we have used 750.000 updates to train the agents.

Learning Rate

Sutton and Barto (1998, Ch.8.4) demonstrate the effects of various learning rates on the performance of SARSA(λ) with tile coding, when applied to the mountain cart problem with five tilings. Although both the algorithm and the problem are not the same, the case is similar enough to use as a start for the search for the best learning rate. In that demonstration, it seems that learning rate between 0.05 and 0.2 provides the best performance.

Due to time limits we have chosen to use only two learning rates. To determine the most promising learning rates, we did one sample run of 750.000 updates for each $\alpha = 0.01, 0.05, 0.1, 0.2$, and $\eta = 0, 0.1, 0.2, ..., 1.0$. The performance of the resulting policies after 750.000 updates was determined by the lap times of an offline run of five minutes.

Table 4.2: Lap times of a 5-minute offline run after 750K updates (in s).

	η										
α	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.2	-	70.7	-	58.2	58.8	60.0	54.0	57.7	52.5	-	-
0.1	-	-	59.0	60.1	57.0	57.3	57.5	61.0	-	62.7	-
0.05	-	-	-	56.6	52.9	-	-	-	-	-	-
0.01	-	-	-	-	-	-	-	-	-	-	-

Table 4.2 shows that there is no η for which a combination with $\alpha = 0.01$ provides a policy that can finish a lap. There are two values of η for which $\alpha = 0.05$ provides a policy that can finish a lap. But the most robust learning rates seem to be $\alpha = 0.1$ and $\alpha = 0.2$, because both values provide seven acceptable policies with various values of η.

It could be that a specific learning rate might not be very robust, but have a good peak performance. However, the lap times of $\alpha = 0.05$ do not seem to deviate enough from the other lap times to validate such an argument.

Exploration Rate

The exploration rate, or ϵ, of ϵ-greedy regulates the noise in the action selection mechanism of the algorithm, and can be set freely between 0 and 1. It should be sufficiently large to induce the necessary deviation from the contemporary optimal policy. But it should also be small enough to allow the execution of a good policy. For example, the exploration should be large enough to cause the agent to deviate from the track axis, but small enough to keep it from crashing before it ever reaches a curve.

A standard value for the exploration is $\epsilon = 0.1$. In simulated car racing performing a random action every ten actions might be too much noise, even though one random action only affects a time span of 20 ms. However, results from initial training show that the agent can still complete many laps without crashing. Therefore, this value was chosen without further evaluations.

Heuristic Rate

The aim of this study was to find out whether an additional exploration rate η, that controls a heuristic, would improve the learning process of the agent. The base case of this study is to have no heuristic ($\eta = 0$). It would also be interesting to find out what would happen if the agent would learn only from samples gathered by the heuristic ($\eta = 1$). However, the best results would probably be obtained with a value somewhere in between. Table 4.2 shows no specific region of values for η that seems interesting enough to ignore another region. Therefore we chose a uniform distribution of values. The range of [0,1] with steps of 0.1 would have been a good choice. However, due to time constraints the number of experiments had to be limited and we chose to pick values between 0 and 1 with steps of 0.2.

Results 5

Section 5.1 describes the lap times of the agents trained with various parameter settings in an offline run, after training is completed. Section 5.2 compares the performance of agents trained with and without heuristic during the training process by means of the accumulated reward in an offline run. Section 5.3 places the lap times of the trained agents in context by comparing them to deterministic policies.

Note that the agents that were not able to finish at least one lap are sometimes removed from the data. In that case, the percentage of agents that were able to finish at least one lap is denoted in the table as 'success rate'. To compute the standard error of the mean lap times, the sample size is defined as the number of completed laps of all agents with a certain parameter setting. For example, one particular combination of parameter values that has a sample size of one can still have a standard error because that single agent drove multiple laps, and therefore has multiple lap times.

5.1 Rate of Heuristic Exploration

To test the offline performance, each combination of parameters is used to train ten agents with 750.000 updates. Each agent then drives for five minutes on the track without learning. The resulting lap times are used as a performance measure. The agents that were not able to finish a lap are removed from the data, because it would distort the computation of the mean lap time of the good policies. Instead, each table states the number of agents that were able to finish a lap.

Alpha 0.1

Table 5.1 shows the lap times of the agents trained with $\alpha = 0.1$. First, note that the cases no heuristic ($\eta = 0.0$) and full heuristic ($\eta = 1.0$) did not produce agents that could finish a complete lap. The other lap times all seem to lie between 54 and 58 seconds, with a standard error of approximately 4 seconds. Also note that the highest success rate is 70%, whereas the success rate in the experiments that determined the number of updates (section 4.3) was between 80% and 100% for more than 450K updates. This variation is probably due to the small number of experiments.

The policies trained with $\eta = 0.2$ and $\eta = 0.8$ seem to have the lowest mean lap times, and the highest success rates. But since all sample means are within a standard error of each other, there is no statistical significant difference between these values. From these results, we can conclude that the agent benefits from training with a heuristic, as long as it is not used all the time.

Table 5.1: Average laptime (in seconds) of successful agents during a 5-minute offline run, $\alpha = 0.1$. Each setting was tested with ten agents, the agents that did not finish at least one lap are removed from the data.

η	sample mean	std error	success rate
0.0	-	-	0
0.2	55.3s	4.8s	7
0.4	57.9s	3.3s	5
0.6	57.4s	4.0s	3
0.8	54.8s	4.5s	6
1.0	-	-	0

Alpha 0.2

Table 5.2 shows the lap times of the agents trained with $\alpha = 0.2$. Again, there are no lap times for agents trained with $\eta = 0.0$. However, as opposed to $\alpha = 0.1$, there is an agent trained with $\eta = 1.0$ that can finish a lap. The lap times lie between 57 and 64 seconds. The standard deviation varies for different values of eta. However, none of these differences are statistically significant.

Table 5.2: Average lap time (in seconds) during a 5-minute offline run, $\alpha = 0.2$. Each setting was tested with ten agents, the agents that did not finish at least one lap are removed from the data.

η	sample mean	std error	success rate
0.0	-	-	0
0.2	63.9s	4.0s	3
0.4	60.0s	5.7s	7
0.6	60.4s	5.6s	6
0.8	57.5s	4.0s	7
1.0	57.6s	2.5s	1

When we compare the results of $\alpha = 0.1$ with $\alpha = 0.2$ across all values of η, we can see that the lap times overall are few seconds higher. The first has a mean of 56.1 seconds per sample, the latter a mean of 59.7 seconds per sample. The respective standard errors are 4.2 seconds, and 4.8 seconds. Again, there is no significant difference.

5.2 Heuristic vs. No Heuristic

In the previous section, we observed that the value of the heuristic makes no difference. However, for both learning rates the original Q-learning with ϵ-greedy did not produce a single agent that could finish a lap after this number of updates. Let us look closer at this difference in results by comparing the accumulated reward of the agents trained with and without heuristic, respectively $\eta = 0.2$ and $\eta = 0.0$.

Figure 5.1 shows the average accumulated reward after five minutes of offline racing during the learning phase. Every 50.000 steps the Q-table was stored. Afterwards, each Q-table was loaded into an agent, which drove for five minutes without updating its Q-values. The conditions are the same as before, except that no agent was removed from the data.

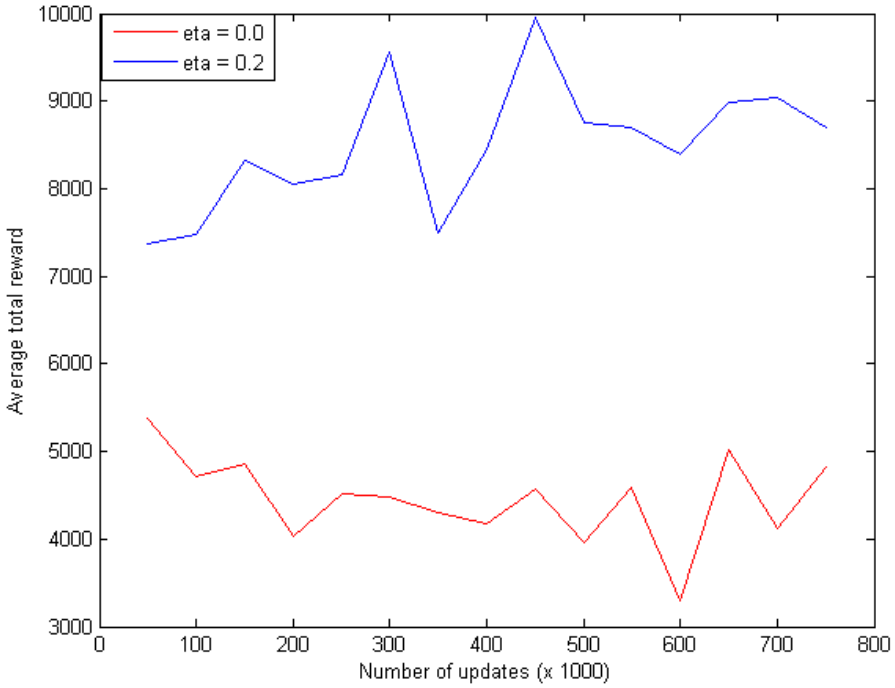


Figure 5.1: Average accumulated reward after 5 minutes of offline racing for agents trained with $\eta = 0$ and $\eta = 0.2$, $N = 10$.

The figure shows a difference of 1993 points between the two conditions after 50.000 updates, which increases to 3890 after 750.000 updates. This difference is quite large, considering the fact that the rewards approximately lie between -1 and 1. More interesting is the fact that the $\eta = 0.2$ condition shows an upward trend, whereas the trend of the $\eta = 0.0$ condition seems to be a concave curve. The latter could even be considered to have a negative trend.

5.3 Reinforcement Learning vs. Other Drivers

In section 5.1 we have seen the mean lap times of the agents trained with RL and a heuristic. But if one were to enter the SCR championship, it is interesting to know how well the heuristic RL agent performs compared to other drivers. Due to time limits, we have not implemented and trained the SCR competitors.

However, we can compare the RL agent with 'SimpleDriver', the driver example of the SCR framework, and the hand-tuned driver 'Berniw', which is written by the creator of TORCS. SimpleDriver is the continuous version of the heuristic, and is described in algorithms 1 and 2. Berniw is not part of the SCR framework, and can access information that the RL agent cannot. It might therefore be an unfair comparison, but Berniw is one of the fastest automated drivers in TORCS, and serves as an upper bound to the agent's performance.

Let us compare the speed of the best RL agent, which is arguably the agent trained with $\eta = 0.8, \alpha = 0.1$. Let us compare the total lap time of each driver after 20 laps, $N = 10$. Table 5.3 shows the total lap times.

Table 5.3: Average time to finish 20 laps for various drivers. Berniw serves as an upper-bound of the agent's performance, and SimpleDriver should be a lower-bound. $N = 10$.

Driver	Duration	Std error
Berniw	636.5s	-
RL agent $\eta = 0.8, \alpha = 0.1$	1054.8s	14.5s
SimpleDriver	1362.1s	-

As expected, the fastest driver is Berniw, with a total lap time of 636.5 seconds. The total lap time of the RL agent and SimpleDriver are 1054.8 seconds and 1362.1 seconds, respectively. Because Berniw and SimpleDriver are deterministic, each trial results in the same total lap time. Therefore, they have no standard error.

The time difference between Berniw and the RL agent is 418.3 seconds and the time difference between the RL agent and SimpleDriver is 307.3 seconds. Respectively, this corresponds to approximately 28 and 21 times the measured standard error of the RL agent. These differences are large enough to conclude that there is a significant difference in the speeds of these drivers without statistical tests.

Discussion 6

The results show that, compared to normal Q-learning with ϵ-greedy, the use of a heuristic to guide the agent towards better states increases the quality of the policy within the frame of 750.000 updates. For both learning rates, the $\eta = 0$ condition cannot complete a lap, whereas each other condition, except for $\eta = 1$, can find a policy that can finish a lap within 64 seconds. Remarkably, the quality of the policies do not seem to differ much for various values of η, since the lap times are approximately the same. What does seem to differ is the probability of finding that policy.

The little difference in quality of the policy for the various values of η could mean that this combination of techniques is quite robust to parameter values. However, it is more likely that some design choices make it difficult to clearly observe the difference between the effects of the various parameter values. For example, the sample size could be too small. With more time, more agents might have been trained, which could have resulted in more measurements. Of course, it might also be the case that other factors are more limiting, such as the chosen track, the values of the algorithm parameters or the number of actions. Such factors might limit the policy space, which perhaps makes it difficult for agents with some value of η to reach their full potential.

Another conclusion that might be drawn after more runs is whether an agent with only heuristic action selection ($\eta = 1$) can systematically learn a feasible policy in this scenario. The results from $\alpha = 0.2$ show that it is possible, but more runs could show whether this one successful run was more than chance. Whatever the outcome, one could question the usefulness of such a solution. There is no practical reason to prefer a training with only heuristic action selection above a training with a balance between heuristic, random, and greedy action selection, since it severely limits the exploration of the agent. Even when using an exceptional heuristic, it cannot hurt to sometimes explore states and actions outside the box of this deterministic strategy.

More runs is not the only way to find more differences between the parameters. From the results of figure 4.2 we can conclude that more updates do not necessarily produce a better policy. If Q-learning finds a policy that does not crash, the resulting policy takes approximately one minute to finish, even after only 150.000 updates. Up to approximately 750.000 updates it only increases the likelihood of finding a policy that does not crash. Apparently, it is more difficult to find a consistent slow policy, than a risky policy that takes less than a minute to finish.

This lack of improvement after more updates probably means that this lap time of approximately 55 seconds is the limit with the chosen restrictions. This might have something to do with the rewards. Perhaps the lookahead of 1.38s is not enough or perhaps the reward function does not give enough incentive to stay on the track in relation to the incentive to drive fast.

It might also be due to the coarse state representation, but it is most likely a result of the

restricted action space. For instance, although the SimpleDriver needs large steering actions for sharp turns at high speeds, it generally uses (continuous) actions in the order of 1% of the steering range. Whereas the RL agent has to choose between 0, 10% or 50% of the steering range. Perhaps this is simply too crude. Expanding the action space would facilitate more subtle actions, and thus better policies. This should increase the difference between good and bad parameters.

Expanding the state or action space, however, would also increase the necessary number of updates to learn a good policy. This is part of a trade-off between representational power and learning speed. This study has a design that facilitates a high learning speed. With more time available for training, it might be worthwhile to examine an agent with more representational power. Theoretically, that should result in a driver with more competitive lap times.

The results of 5.2 seems to indicate that within this time frame, the two conditions seem to be in different stages of development. The heuristic agents already seem to be improving upon a certain policy, while the agents without heuristic are still in the initial exploration phase. Indeed, this is exactly the difference that one would expect when frequently giving an agent a sample of a rewarding policy.

The lap times of Berniw and SimpleDriver show that Berniw is a much better driver. Yet, this study uses the SimpleDriver as a heuristic to guide the agent to the optimal policy. Ignoring practical problems like the fact that SimpleDriver was the only deterministic policy that was readily available, one could argue that it would have been better to use Berniw as a heuristic. Surely, it would guide the agent towards better states. But the premise in this study was to give the agent as little information as possible. Using Berniw instead of SimpleDriver would greatly increase the initial information given to the agent and violate that premise. Not to mention the fact that Berniw can access track information that the agent cannot, using that information would violate the restrictions of the SCR framework.

Conclusions and Future Work 7

In this study we have built an agent that learns how to race by means of reinforcement learning. To do this, the agent used Q-learning, which was combined with tile coding for state representation and ϵ-greedy to guide its exploration. Additionally, ϵ-greedy was extended to incorporate the selection of actions according to a heuristic, which was controlled by a parameter η. Section 7.1 describes the conclusions that can be drawn from this study. This is followed by directions for future research in section 7.2.

7.1 Conclusions

One of the aims of this study was to show that it is possible to train an agent with Q-learning to drive around a track in TORCS without crashing, with little prior knowledge and under the conditions of the warm-up phase of the SCR championship. However, it seems impossible to meet both constraints, because they influence each other. To learn how to drive in five minutes in-game time, which is the length of the warm-up phase of the championship, an agent needs a lot of information about how the world works. Similarly, an agent that has little knowledge about the world needs more than five minutes of experience to learn a good policy. We found the restriction of giving the agent little prior knowledge more interesting than a strictly limited training time. Therefore, we chose to loosen the time constraint of the championship's warm-up phase.

Speed vs. Quality

The required learning time is not only influenced by prior knowledge, but also by the quality of the representation. A more detailed representation generally facilitates a better solution, but also increases the learning time. As table 8.2 (in appendix A) shows, the agent has 67 continuous sensors available. Creating a look-up table with Q-values for all values of all sensors is impossible, thus the number of sensors must be reduced. The restrictions of the warm-up phase make this process easier, because some sensors become superfluous, for example the opponent sensors.

Even with a selection of discretized inputs, it is not an option to create a look-up table. Thus, we needed a function approximator. Preliminary results showed that a neural network was difficult to apply to this problem. First, because it introduces extra parameters that need to be tuned. Second, because it allows only for global updates, whereas local Q-value updates are more useful in this context. Tile coding proved to be a more effective function approximator. It allows for local updates, and is more robust to various parameter values than a neural network.

However, a disadvantage compared to a neural network is that tile coding needs discretization of the state space.

In this study we have chosen to favor a high learning speed above representational power to quickly see whether the principle works at all. The state space was limited to 8 discretized dimensions, to reduce the learning time, and the two continuous output dimensions were reduced to 15 discrete actions. However, the reduction of the state space also influences the differences in the environment that an agent can perceive, and the reduction of the action space influences the nuance of the actions of the agent. The result was that the quality of the policy was more or less constant across all conditions and that more training only increased the probability of finding such a policy, until it was found almost every run. Apparently, under the conditions of this experiment, it is less difficult to develop a slow, safe policy than a fast and risky one.

The Value of a Heuristic

The main research question was '*What is the effect of the extension of regular heuristic guidance on the performance of Q-learning in simulated car racing?*'. Conform with the aim to give the agent as little prior knowledge as possible, this heuristic was kept as simple as possible. The results of the experiments show that with restricted time to learn a policy, an agent can benefit from the frequent use of a heuristic to guide exploration. None of the agents with standard Q-learning and ϵ-greedy were able to learn to drive a whole lap within the given time limit, whereas for any frequency of the heuristic guidance, at least one agent could learn such a policy. From this can be concluded that the learning speed has increased due to the use of a heuristic.

In contrast to what one would expect, this experiment did not show a significant difference between the policies obtained with various frequencies of the use of the heuristic. Perhaps this was due to limited representational power or the limited number of actions available to the agent. A follow up study with an extended state or action space would have to show whether this is indeed the case.

Questioning the Premises

Now that there is an example of an agent that can learn how to race in TORCS by means of reinforcement learning, the initial premises of this study can be questioned. The idea of training an agent from scratch under the conditions of the warm-up phase of the SCR Championship is alluring. The principle is interesting, but in practice there is no reason to do this. Training a robust driver beforehand, and fine-tuning that driver during the actual warm-up phase of a race will most likely produce a better driver. Above that, the agent will not learn competitive behavior under these training conditions. However, in a group race, specific competitive behavior will improve the driver's performance. A solution to that problem would be to train competitive behavior separately, and load it after the warm-up round. However, this would violate the principle of training the agent from scratch. The current setup is too simple to have an agent learn a competitive policy under these premises. Further research is needed to check whether these premises could hold at all.

7.2 Future Work

There is little work available on reinforcement learning applied to simulated car racing, and there remains a lot to explore. Apart from time trials, car racing is a multi-agent domain. It would therefore be wise to include the modeling of, or at least dealing with, opponents to the training process. However, it would be complicated to learn how to race from scratch in a multi-agent set-

ting. Perhaps it is wiser to first create an agent that can drive by itself and add the competitive behavior later. This study has taken the latter approach and though the addition of a heuristic seemed to improve learning under time constraints, it has not yet succeeded in completing this first step. Therefore this section is limited to suggestions concerning single agent racing.

As mentioned before, the agent in this study had little sensory information and actions available. More sensory information means that the agent might be able to find better policies, but also that the agent needs more training samples. Those samples, or the required time to process them, might not always be available. On the other hand, one could reduce the training time by exploiting the symmetry in the environment. For example by updating symmetric opposites of the visited curve during the same time step. Future research could be directed towards finding the balance of the costs and benefits of giving the agent more sensory information in this domain.

Another interesting direction would be to handle the current sensory information more effectively by changing the tile coding parameters. Obviously, one could simply add more tilings, but the agent might benefit more from a better strategy to define the tilings. The tilings could be defined in such a way that they have more precision in the states that are visited often. Perhaps the tilings could even change over time, splitting a tile when it is visited more than a certain threshold (Whiteson et al., 2007). This would allow for more subtleties in the agent's policy.

Also, the value of a tile could be initialized at a value that depends on its neighboring cells, for example equal to its nearest neighbor or to a weighted average of its k-nearest neighbors. It is an acceptable assumption that driving a little faster or deviating a little from the previous position does not change the Q-value a lot. Although knowing those subtleties is exactly the difference between first and second place. Nonetheless, initializing the values under this assumption could save a lot of unnecessary exploratory actions.

General measures in reinforcement learning for finding a better policy include a decreasing learning rate and a decreasing exploration rate. This is applicable to the simulated car racing domain as well. After some training time both of these measures stabilize the behavior that is being learned. Since only little changes are made, the chance of crashing due to exploration is reduced. This would facilitate fine-tuning the policy at high speeds. Another beneficial feature of these measures is that the values initially lead to faster convergence and more exploration respectively. This pushes the agent faster towards a rewarding policy, which should lead to the collection of better samples.

Both of the above mentioned measures could be combined with the directed exploration of a heuristic. A possible pitfall might be that a combination of all these measures might result in a relatively fast convergence to a certain policy, leading the agent to an initially promising, but ultimately, sub-optimal solution. In that case, the parameters would have to be tuned carefully. It might be easier to combine these measures with optimistic initialization of the Q-values instead. This increases exploration of unvisited states, and somewhat counters the high convergence rate of the above mentioned measures.

Similarly, it might be interesting to try a decreasing heuristic rate or bootstrapping the learning algorithm with a fixed amount of heuristic examples (Smart and Kaelbling, 2000). Since the RL agent seems to improve upon the original heuristic, it might not need the examples of the heuristic after some time of training.

Finally, the main assumption in this thesis was that the RL agent should learn to drive almost from scratch, which was the reason to give it a very simple policy as a heuristic. Even though it used a limited, discrete version of that policy, the RL agent learned a faster policy than the original, continuous version. From a racing perspective, it might therefore be interesting to let go of that assumption and use Berniw or one of the SCR Championship winners as a heuristic. Perhaps reinforcement learning could be used to improve upon the policies of those expert drivers as well.

Bibliography

Abdullahi, A. A. and Lucas, S. M. (2011). Temporal difference learning with interpolated n-tuples: Initial results from a simulated car racing environment. In *2011 IEEE Conference on Computational Intelligence and Games (CIG11)*, pages 321–328. IEEE Press.

Albus, J. S. (1971). A theory of cerebellar function. *Mathematical Biosciences*, 10:25–61.

Albus, J. S. (1981). *Brains, Behavior, and Robotics*. Byte Books, Peterborough, NH.

Athanasiadis, C., Galanopoulos, D., and Tefas, A. (2012). Progressive neural network training for the open racing car simulator. In *Proceedings of the IEEE Symposium on Computational Intelligence in Games (CIG '12)*, pages 116–123. IEEE.

Atkeson, C. G. and Schaal, S. (1997). Robot learning from demonstration. In *ICML*, volume 97, pages 12–20.

Bakker, P. and Kuniyoshi, Y. (1996). Robot see, robot do: An overview of robot imitation. In *AISB96 Workshop on Learning in Robots and Animals*, pages 3–11.

Barreno, M. and Liccardo, D. (2003). Reinforcement learning for rars. Technical report, University of California, Berkeley,.

Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H. (1989). Learning and sequential decision making. In *Learning and Computational Neuroscience*, pages 539–602. MIT Press.

Bellman, R. E. (1957a). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Bellman, R. E. (1957b). A markov decision process. *Journal of Mathematical Mechanics*, 6:679–684.

Bertsekas, D. and Tsitsiklis, J. N. (1996). *Neuro-dynamic Programming*. Athena Scientific, Belmont, MA.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Butz, M. V., Linhardt, M. J., and Lönneker, T. D. (2009). Optimized sensory-motor couplings plus strategy extensions for the torcs car racing challenge. In *IEEE Symposium on Computational Intelligence and Games, 2009. CIG '09.*, pages 317–324. IEEE, IEEE Press.

Butz, M. V., Linhardt, M. J., and Lönneker, T. D. (2011). Effective racing on partially observable tracks: Indirectly coupling anticipatory egocentric sensors with motor commands. *IEEE Transactions on Computational Intelligence and AI in games*, 3(1):31–42.

Campbell, M., Hoane, A. J., and Hsu, F.-H. (2002). Deep blue. *Artificial Intelligence*, 134:57–83.

Cardamone, L., Loiacono, D., and Lanzi, P. L. (2009). Evolving competitive car controllers for racing games with neuroevolution. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computing*, pages 1179–1186. ACM.

Dayan, P. and Sejnowski, T. (1996). Exploration bonuses and dual control. *Machine Learning*, 25:5–22.

Eiben, A. E. and Smith, J. E. (2003). *Introduction to Evolutionary Computing*. SpringerVerlag.

Fürnkranz, J. (2001). Machine Learning in Games: A Survey. In Fürnkranz, J. and Kubat, M., editors, *Machines That Learn to Play Games*, pages 11–59, Huntington, NY. Nova Science Publishers.

Galway, L., Charles, D., and Black, M. (2008). Machine learning in digital games: a survey. *Artificial Intelligence Review*, 29:123–161.

Graepel, T., Herbrich, R., and Gold, J. (2004). Learning to fight. In Mehdi, Q., Gough, N., Natkin, S., and Al-Dabass, D., editors, *Proceedings of the 5th International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 193–200. The University of Wolverhampton.

Howard, R. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA.

Hsu, F.-H. (2002). *Behind deep blue: building the computer that defeated the World Chess Champion.* Princeton University Press, Princeton.

Laird, J. and van Lent, M. (2001). Human-level ai's killer application interactive computer games. *AI Magazine*, 22:15–25.

Loiacono, D., Cardamone, L., and Lanzi, P. L. (2012). *SCR Championship - Competition Software Manual*. Retrieved at June 6, 2013, from http://sourceforge.net/projects/cig/files/SCR

Loiacono, D., Lanzi, P. L., Togelius, J., Onieva, E., Pelta, D. A., Butz, M. V., Lönneker, T. D., Cardamone, L., Perez, D., Sáez, Y., Preuss, M., and Quadflieg, J. (2010a). The 2009 simulated car racing championship. *IEEE Transactions on Computational Intelligence and AI in games*, 2(2):131–147.

Loiacono, D., Prete, A., Lanzi, P. L., and Cardamone, L. (2010b). Learning to overtake in torcs using simple reinforcement learning. In *WCCI 2010 IEEE World Congress on Computational Intelligence*, pages 18–23. IEEE Press.

Loiacono, D., Togelius, J., Lanzi, P. L., Kinnaird-Heether, L., Lucas, S. M., Simmerson, M., Perez, D., Reynolds, R. G., and Saez, Y. (2008). The wcci 2008 simulated car racing competition. In *IEEE Symposium On Computational Intelligence and Games, 2008. CIG '08.* IEEE Press.

Lucas, S. M. and Kendall, G. (2006). Evolutionary computation and games. *IEEE Artificial Intelligence Magazine*, February:10–18.

Lucas, S. M. and Togelius, J. (2007). Point-to-point car racing: an initial study of evolution versus temporal difference learning. In *Proceedings of the 2007 IEEE symposium on computational intelligence and games*, pages 260–267. IEEE Press.

Maderia, C., Corruble, V., and Ramalho, G. (2006). Designing a reinforcement learning-based adaptive ai for largescale strategy games. In Laird, J. and Schaeffer, J., editors, *Proceedings of the 2nd artificial intelligence and interactive digital entertainment conference*, pages 121–123, Menlo Park. AAAI.

McPartland, M. and Gallagher, M. (2008). Creating a multi-purpose first person shooter bot with reinforcement learning. In *2008 IEEE Symposium on Computational Intelligence and Games (CIG'08)*, pages 143–150. IEEE Press.

Microsoft (2013). Drivatar^{TM}theory. Retrieved July 24, 2013, from http://research.microsoft.com/en-us/projects/drivatar/theory.aspx.

Miikkulainen, R., Bryant, B. D., Cornelius, R., Karpov, I. V., Stanley, K. O., and Yong, C. H. (2006). Computational intelligence in games. In Yen, G. Y. and Fogel, D. B., editors, *Computational Intelligence: Principles and Practice*. IEEE Computational Intelligence Society, Piscataway, NJ.

Onieva, E., Pelta, D. A., Godoy, J., Milanés, V., and Pérez, J. (2012). An evolutionary tuned driving system for virtual car racing games: The autopia driver. *International Journal of Intelligent Systems*, 27:217–241.

Ponsen, M., Spronck, P., and Tuyls, K. (2006). Hierarchical reinforcement learning with deictic representation in a computer game. In Schobbens, P.-Y., van Hoof, W., and Schwanen, G., editors, *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence*, pages 251–258. University of Namur.

Price, B. and Boutilier, C. (2003). Accelerating reinforcement learning through implicit imitation. *Journal of Artificial Intelligence Research*, 19:569–629.

Pyeatt, L. D. and Howe, A. E. (1998). Learning to race: Experiments with a simulated race car. In *Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference*, pages 357–361.

Pyeatt, L. D., Howe, A. E., et al. (2001). Decision tree function approximation in reinforcement learning. In *Proceedings of the Third International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models*, volume 1/2, pages 70–77.

Quadflieg, J., Preuss, M., Kramer, O., and Rudolph, G. (2010). Learning the track and planning ahead in a car racing controller. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games (CIG'10)*.

Quadflieg, J., Preuss, M., and Rudolph, G. (2011). Driving faster than a human player. In *Applications of Evolutionary Computation*, pages 143–152. Springer.

Rumelhart, D. E. and McClelland, J. L. (1986). *Parallel distributed processing: Explorations in the microstructure of cognition. Volume I.* MIT Press, Cambridge, MA.

Samuel, A. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3(3):211–229.

Schaal, S. (1999). Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer Verlag.

Schaeffer, J. (2000). The games computers (and people) play. *Advances in Computers*, 52:189–266.

Schmidhuber, J. (1991). Curious model-building control systems. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 1130–1135, Singapore. IEEE.

Shannon, C. (1950). Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275.

Skinner, B. (1932). On the rate of formation of a conditioned reflex. *The Journal of General Psychology*, 7(2):274–286.

Smart, W. D. and Kaelbling, L. P. (2000). Practical reinforcement learning in continuous spaces. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 903–910, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Sutton, R. (1988). Learning to predict by the methods of temporal difference. *Machine Learning*, 3:9–44.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 8:1038–1044.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Sutton, R. S., Precup, D., and Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211.

Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68.

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.-E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A., and Mahoney, P. (2006). Stanley: The robot that won the darpa grand challenge. *Journal of Field Robotics*, 23(9):661–692.

Thrun, S. B. (1992). Efficient exploration in reinforcement learning. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA.

Togelius, J., Lucas, S. M., and de Nardi, R. (2007). Computational intelligence in racing games. *Studies in Computational Intelligence*, 27:39–69.

Tokic, M. (2010). Adaptive e-greedy exploration in reinforcement learning based on value differences. In Dillmann, R., Beyerer, J., Hanebeck, U., and Schultz, T., editors, *KI 2010: Advances in Artificial Intelligence*, volume 6359 of *Lecture Notes in Computer Science*, pages 203–210. Springer Berlin Heidelberg.

van Hasselt, H. (2012). *Insights in Reinforcement Learning - Formal analysis and empirical evaluation of temporal-difference learning algorithms*. PhD thesis, University of Utrecht.

Vermorel, J. and Mohri, M. (2005). Multi-armed bandit algorithms and empirical evaluation. In *Proceedings of the 16th European Conference on Machine Learning (ECML'05)*, pages 437–448, Porto, Portugal.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. PhD thesis, University of Cambridge.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

Whiteson, S., Taylor, M. E., Stone, P., et al. (2007). *Adaptive tile coding for value function approximation*. Computer Science Department, University of Texas at Austin.

Wiering, M. (1999). *Explorations in Efficient Reinforcement Learning*. PhD thesis, University of Amsterdam.

Williams, D. (2002). Structure and competition in the u.s. home video game industry. *The Internati onal Journal on Media Management*, 4(1):41–54.

Appendix A. Inputs and outputs of the TORCS client 8

Table 8.1: Description of available effectors (from: Loiacono et al. 2008)

Name	**Description**
accel	Virtual gas pedal (0 means no gas, 1 full gas).
brake	Virtual brake pedal (0 means no brake, 1 full brake).
gear	Gear value
steering	Steering value: -1 and +1 means respectively full left and right, that corresponds to an angle of 0.785398 rad.
meta	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race.

Table 8.2: Description of available sensors (from: Loiacono et al. 2008)

Name	**Description**
angle	Angle between the car direction and the direction of the track axis.
curLapTime	Time elapsed during current lap.
damage	Current damage of the car (the higher is the value the higher is the damage).
distFromStartLine	Distance of the car from the start line along the track line.
distRaced	Distance covered by the car from the beginning of the race.
fuel	Current fuel level.
gear	Current gear: -1 is reverse, 0 is neutral and the gear from 1 to 6.
lastLapTime	Time to complete last lap.
opponents	Vector of 36 sensors that detects the opponent distance in meters (range is [0,100]) within a specific 10 degrees sector: each sensor covers 10 degrees, from $-\pi$ to $+\pi$ around the car.
racePos	Position in the race with respect to other cars.
rpm	Number of rotations per minute of the car engine.
speedX	Speed of the car along the longitudinal axis of the car
speedY	Speed of the car along the transverse axis of the car
track	Vector of 19 range finder sensors: each sensor represents the distance between the track edge and the car. Sensors are oriented every 10 degrees from $-\pi/2$ and $+\pi/2$ in front of the car. Distance are in meters within a range of 100 meters. When the car is outside of the track (i.e. trackPos is less than -1 or greater than 1), these values are not reliable!
trackPos	Distance between the car and the track axis. The value is normalized w.r.t. the track width: it is 0 when the car is on the axis, -1 when the car is on the left edge of the track and +1 when it is on the right edge of the car. Values greater than 1 or smaller than -1 means that the car is outside of the track.
wheelSpinVel	Vector of 4 sensors representing the rotation speed of the wheels.

Appendix B. Edges of the tilings 9

Table 9.1: The edges of the tiles that categorise the speed dimension.

Tiling	Speed edges					
1	-10	0	50	120	180	320
2	-10	10	60	130	200	320
3	-10	20	70	140	220	320
4	-10	30	80	150	240	320
5	-10	40	90	160	260	320

Table 9.2: The edges of the tiles that categorise the position dimension.

Tiling	Position edges					
1	-2	-0.8	-0.3	0.0	0.4	2
2	-2	-0.7	-0.2	0.05	0.5	2
3	-2	-0.6	-0.15	0.1	0.6	2
4	-2	-0.5	-0.1	0.2	0.7	2
5	-2	-0.4	-0.05	0.3	0.9	2

Table 9.3: The edges of the tiles that categorise the angle dimension.

Tiling	Angle edges					
1	-2	-0.8	-0.3	0.0	0.4	2
2	-2	-0.7	-0.2	0.05	0.5	2
3	-2	-0.6	-0.15	0.1	0.6	2
4	-2	-0.5	-0.1	0.2	0.7	2
5	-2	-0.4	-0.05	0.3	0.9	2

Table 9.4: The edges of the tiles that categorise the input from the distance sensors.

Tiling	Distance edges					
1	-1	10	50	100	150	200
2	-1	20	60	110	160	200
3	-1	30	70	120	170	200
4	-1	40	80	130	180	200
5	-1	50	90	140	190	200