UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR ELECTRICAL
ENGINEERING IN MEDICINE

Application and evaluation of Reinforcement Learning methods for the autonomous drifting of a remote controlled car

Anwendung und Evaluation von Reinforcement Learning Verfahren für das autonome Driften eines Modellfahrzeugs

Bachelorarbeit

verfasst am
Institut für Medizinische Elektrotechnik

im Rahmen des Studiengangs
Robotik und Autonome Systeme
der Universität zu Lübeck

vorgelegt von
Fabian Domberg

ausgegeben und betreut von
Prof. Dr. Georg Schildbach

Lübeck, den 9. September 2020

IM FOCUS DAS LEBEN

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Fabian Domberg

Zusammenfassung

Autonome Fahrzeuge sind im letzten Jahrzehnt wesentlich sicherer geworden. Trotz dessen haben sie, ähnlich wie menschliche Fahrer, Probleme mit Aquaplaning und vereisten Straßen. Professionelle Motorsportfahrer gleiten ihr Fahrzeug während des Driftens regelmäßig gezielt seitwärts über die Strecke. Jüngste Forschung zeigt, dass es mittels Machinellem Lernen tatsächlich möglich ist, dieses Verhalten nachzubilden. Diese Arbeit entwickelt eine Reglerarchitektur für autonomes Driften mittels Reinforcement Learning. Ziel ist es, hiermit den Raum der kontrollierbaren Fahrsituationen von autonomen Fahrzeugen zu erweitern. Die Aufgabe des Driftens ist in einfach verständliche Teile aufgeteilt und so definiert, dass eine Anwendung auf beliebige Pfade möglich ist. Der resultierende Regler ist in Simulation in der Lage variierende physikalische Parameter zu handhaben und auf andere Fahrzeugkonfigurationen übertragbar - all das mit nur einem kleinen neuronalen Netz. Seine Generalisierbarkeit wird ebenfalls mittels eines ferngesteuerten Autos evaluiert.

Abstract

Autonomous vehicles have become considerably safer in the past decade. However, much like human drivers, they struggle with aquaplaning and icy roads. During motorsport events, professional drivers regularly slide their vehicles sideways when drifting. Recent research shows that it is indeed possible to replicate this behaviour with Machine Learning. This thesis develops an autonomous drifting controller architecture using Reinforcement Learning. With this, it aims at expanding the realm of controllable driving situations for autonomous vehicles. The drifting problem is separated in easily comprehensible parts and defined such that it is applicable to arbitrary paths. The resulting controller is able to handle varying physical parameters in simulation and can be transferred to different vehicle setups - all while using only a small neural network. Its generalizeability is also evaluated on an RC car.

Contents

1 Introduction

Today's autonomous vehicles usually drive very carefully, trying to avoid situations where they might lose control, such as wet or icy roads. However, these situations are part of everyday driving and are sometimes unavoidable. Some professional drivers have the ability to slide their car sideways in a desired direction once lateral rear-tire traction is lost. Teaching such drifting maneuvers to autonomous vehicles could improve their ability to handle these kinds of situations, thus increasing their safety. Developing controllers that push a vehicle to its stability limits can also generally widen autonomous algorithms' understanding and control of a car. This chapter will introduce related research and highlight the contributions of this work.

1.1 Related work

Autonomous drifting has only recently become a research interest. In [11] and Stanford's MARTY [15] a traditional control theory based approach is used. They therefore meticulously model the vehicle dynamics and derive analytical solutions. Using the resulting equations of motion [15] are able to drift a reference trajectory, reaching a desired velocity and yaw angle. A similar approach is used by [17], also modeling vehicle dynamics and then using optimal control to develop an initial controller in simulation. This controller is then transferred to a remote controlled (RC) car, where it is further improved using Reinforcement Learning and soon manages to sustain a circular drifting motion. Reinforcement Learning in general has recently been very successful in learning complex tasks, for example playing the game of Go [22] or solving a Rubik's cube using a robotic hand [19]. These kinds of research projects are often conducted with the help of realistic physical simulations like in [2]. Both [21] and [20] use Reinforcement Learning and simulated environments to train their drifting controllers. [21] train a car to steadily drift circles and then transfer the learned controller to an RC car. Whereas [20] use a simulation environment to drift racetracks while trying to mimic recorded human data.

1.2 Contributions of this thesis

This work formulates the task of drifting arbitrary paths as a Reinforcement Learning problem and proposes an intuitive reward function. Following this approach does not require solving complex nonlinear equations and does also not require there to be reference data the controller tries to replicate, as this is either non-trivial to find or can only be obtained from recording a professional driver. Only basic measurements like positions and velocities are used, thus not demanding the use of additional sensors and equipment. Also investigated are the ability to reduce training time with Imitation Learning, controller robustness to changes in its environment and transferring learned controllers between different vehicle configurations and an RC car.

2

Methodology

This chapter explains the necessary concepts and introduces the selected software. It looks at the dynamics of drifting, establish the notion of Reinforcement Learning and showcases the simulation software. Software packages related to Machine Learning are presented and a conducted interview with a hobbyist drifter is summarized.

2.1 Drifting

The loss of lateral tire traction usually coincides with the loss of stability and ultimately the loss of control over a vehicle, making it slide sideways and eventually spin out. Intentionally loosing lateral rear-wheel traction and having the car's back break out, while still maintaining control by countersteering, thus making the vehicle move sideways at a certain angle, is called drifting. Countersteering occurs when steering in the opposite direction of where the road is curving, i.e., the direction in which the vehicle's back breaks out in while drifting. Skilled drivers are able to deliberately provoke loss of traction and slide their vehicle along the track in a controlled manner. An important element when talking about traction is the friction coefficient μ. When μ is close to 1.0, i.e. high friction, the tires will almost act like they are glued to the road - having very good traction. As μ gets closer to 0.0, e.g. when driving on ice, friction reduces and tires start to slip when spinning, not getting as much grip. Tires can also lose lateral traction, meaning the vehicle does no longer go where its wheels are pointing, but due to the vehicle's inertia also slides sideways when steering. This is what enables drifting. A drift can be initialized in different ways. Most, if not all, of the techniques require a rear-wheel-drive car, as these allow the loss of lateral rear wheel traction while maintaining grip at the front wheels. There are motorsport events such as the Formula DRIFT where drivers compete for prize money and even competitions in which drifters are rated for their angle, speed and overall drift elegance [16].

Sideslip angle

The sideslip angle β is the angle between where a vehicle is facing and the direction it is actually moving in. It is computed from the vehicle's local velocity in lateral (v_x) and longitudinal (v_y) direction, as depicted in Figure 2.1. The formula utilizes the atan2 function, which ensures β to be within the interval $[-180, 180]$ by taking into account the signs of v_x and v_y when internally using the arctangent. This is necessary because both v_x and v_y can be negative, for example when the vehicle is moving backwards. The sideslip angle is therefore computed as

$$\beta = \text{atan2}\left(v_x, v_y\right). \tag{2.1}$$

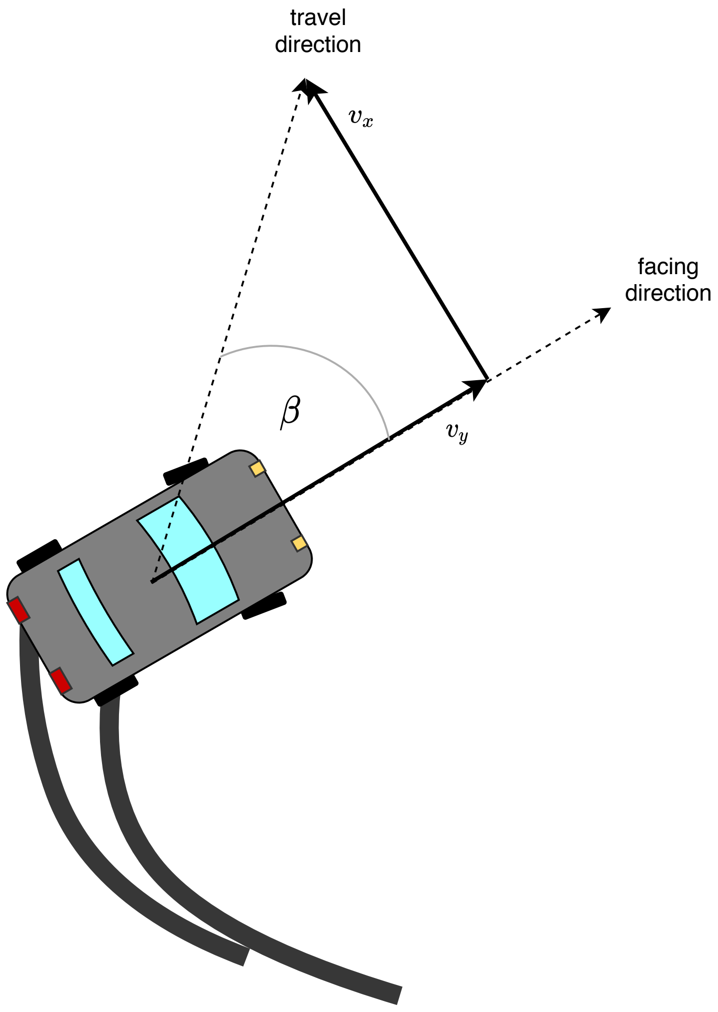


Figure 2.1: Sideslip angle β between a car's longitudinal (v_y) and lateral (v_x) velocity.

When driving around a corner without sliding sideways or its back breaking out, a typical car already produces some amount of sideslip angle. This is due to how cars are designed and how β is defined and therefore does not necessarily imply that $|\beta| > 0$ means the car is sliding sideways or loosing lateral tire traction. Essentially, steering already generates a so-called kinematic sideslip angle β_{kin} proportional to the current steering angle [13]. Typical maximum values range from 15° to 25° for standard cars. The exact value depends on the geometry of a vehicle and can be computed using the arctangent as

$$\beta_{\text{kin}} = \arctan\left(\frac{l_{\text{r}}}{l_{\text{r}} + l_{\text{f}}}\omega_{\text{steer}}\right) \tag{2.2}$$

where l_{f}, l_{r} are the lengths from the center of mass to the front and rear axle
ω_{steer} is the current steering angle.

2.2 Reinforcement Learning

Reinforcement Learning (RL), a subset of Machine Learning, is a method for learning the weights and biases of a neural network to ultimately find the optimal action to take, given an observation [25]. This is usually done by feeding an optimization algorithm observations (inputs) and asking it to produce actions (outputs) which are then rated and a feedback is given. The current state of the network, comprised of its current weights and biases, is referred to as a policy: which action currently produces which output. Updates to the policy are limited by the learning rate, regulating the step size at which weights and biases adjust from one update to another. Contrary to other Machine Learning techniques, e.g., Supervised Learning, the RL algorithm is not initially told what the best action to take given an observation is and does therefore not learn from prerecorded observation-action data pairs. Instead, it is fed back in how desirable its current state is, thus having to deduce itself if the actions taken were correct. This feedback comes in the form the so called reward, which is a number representing how desirable a particular system state is. Which states and behaviors are desirable and which are not is defined by the user within the reward function.

The Reinforcement Learning problem is often modeled as a Markov decision process, whereby the objective is to maximize the expected cumulative reward for any given system state. In this case, the state are the observations that are fed into the neural network. The network essentially tries to predict which of the available actions will produce the most reward not only for the next, but theoretically all, following states. In practice, the cumulative reward impact of future states reduces according to a discount factor, enabling control over how much influence a distant state has on the current decision.

A basic example for Reinforcement Learning is an agent, possibly some sort of robot, in a grid-like environment [23], similar to a chessboard. The agent's possible actions are moving north, south, east and west, while not leaving the grid. It is tasked with reaching a particular cell of the grid. Therefore it is given the current distance to that cell as an observation and a reward once it reaches the cell. Other reward functions could give a reward for getting closer or even give a negative reward for moving away from the goal position. The agent would then have to move around on its own, trying to figure out what actions produce the most reward and eventually learn to move straight towards the target cell.

Being able to learn complex behaviors and decision making just by specifying what the objective is, rather than programming a behavior for every possible observation, is a tremendous advantage of RL. Though, as for most Machine Learning techniques, this also requires a lot of data. Data, in this case, are observations and corresponding actions - in other words: experience. To find what action is the best, you first have to try and take some potentially incorrect actions. This is one of the biggest disadvantages of RL: learning requires making mistakes. In practice it is therefore a good idea to build, experiment with and (pre-)train the system before deploying it to real world hardware.

2.3 Unity Game Engine

For simulation Unity Technologies' Unity Game Engine is used. It is among today's most used videogame development environments, but also regularly used in film (mostly animation), architecture and the automotive industry [9]. Users can quickly create 3D environments by dragging objects around the scene, adding pre-defined behaviors like rigid body physics and even writing custom scripts using C# or JavaScript. Being able to modify build-in systems and creating custom ones if something is not there yet, enables constructing almost any scenario.

Included with the Engine is its own marketplace, the Unity Asset Store, where users can buy and sell 3D models, particle effects, scripts and even extensions for the editor itself. Among them are also a few vehicle packages, mainly aimed at racing game developers, that include 3D cars, road textures and vehicle physics systems.

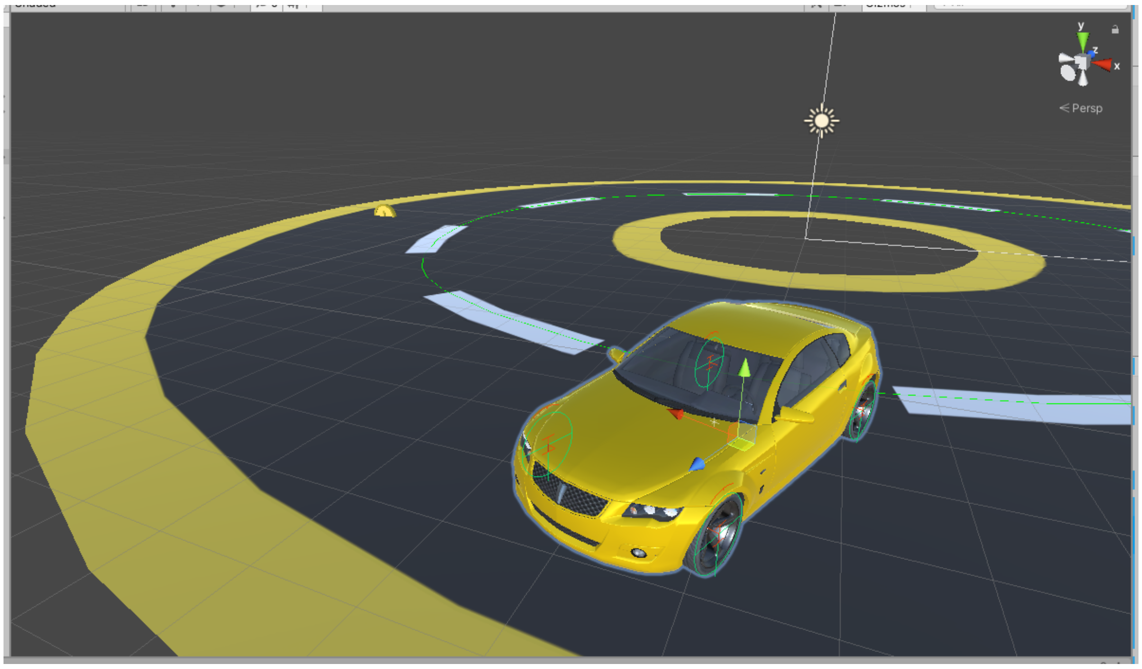
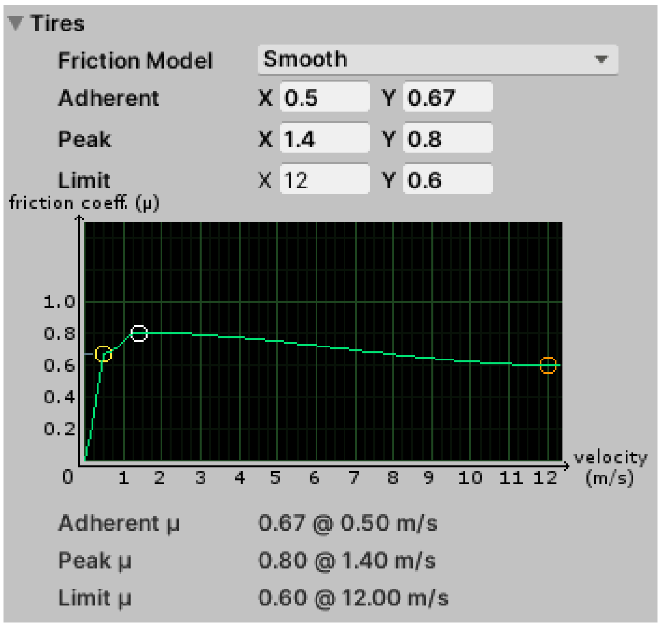


Figure 2.2: Excerpt of a Unity Game Engine screenshot, showing the default vehicle of Edy's Vehicle Physics in its 3D environment.

Edy's Vehicle Physics

Most of the high quality vehicle packages sell at around 60$, sometimes even including their own licensing schema. Edy's vehicle physics however offers a free community edition, which comes with a few restrictions - most notably: having only one vehicle in a scene at a time, only allowing two axle vehicles and not being able to change the underlying code. Apart from these limitations it includes all features available in the full version. Among them are a variety of in-depth settings and parameters regarding the car's configuration, handling and physical behavior. There is for example an option that lets the user decide what axles of the car are driven by the motor, resulting in rear-wheel-, front-wheel- or all-wheel-drive and even custom differential options.

Alongside other detailed and realistic configuration capabilities such as, but not limited to, engine torque curves, transmission types, Ackermann steering, brake ratio and even a fuel consumption model, comes maybe the most important one for the drifting use-case: a tire friction model. Such a model is used to represent the varying friction coefficient μ, which changes depending on the forces acting upon the tire. In this case, μ changes with respect to the vehicle velocity. Edy's Vehicle Physics offers a whole variety of models, ranging from a simple constant coefficient to the Pacejka model, which is regularly used in automotive engineering [6]. Though, for simplicity, a parametric model consisting of only 3 parameters will be used here. The first of them is μ_{adh}, which controls the friction coefficient for low speeds of 0.5 meters per second. Second is μ_{peak}, controlling the height of the curve's peak at speeds of $1.4\frac{m}{s}$. The third parameter μ_{lim} governs the friction for speeds of $12\frac{m}{s}$ and above. A function that includes all of these points is then used to determine intermediate values, as seen in Figure 2.3.



Tires		
Friction Model	Smooth	
Adherent	X 0.5	Y 0.67
Peak	X 1.4	Y 0.8
Limit	X 12	Y 0.6

Adherent μ	0.67 @ 0.50 m/s
Peak μ	0.80 @ 1.40 m/s
Limit μ	0.60 @ 12.00 m/s

Figure 2.3: The parametric tire friction model used. The friction coefficient μ is dependent on the vehicle's velocity.

Unity ML Agents

Unity's Machine Learning Toolkit ML Agents [1] is one of the companies most recent attempts to appeal to users outside of video game development. The open-source extension's objective is to make the game engine's powers available to researchers working with Artificial Intelligence and allow them to easily create environments for their agents to train and be evaluated in. It comes with a few example scenarios and pre-trained agents, showcasing a wide variety of applications ranging from simple balancing tasks over to flying tennis rackets playing each other, cuboids playing football using visual observations and even humanoids trying to walk.

The toolkit comes with the capability of training agents right in the editor, including state-of-the-art algorithms for learning and imitating behaviors. It also provides a Python API which exposes an interface allowing access to observations and taking actions, which can then be hooked up to custom learning algorithms, traditional PID controllers or human inputs. Furthermore developers can export their scene, consistent of the agent and its environment, into an executable file which then does not require the game engine software to run and can be shared with anyone or run on a dedicated machine.

2.4 ROS

The Robot Operation System (ROS) is an open-source software framework that, much like real operating systems, provides services like hardware abstraction and process communication [24]. On top of that, it offers libraries for common robotics problems like pathfinding, computer vision and inverse kinematics. As ROS has been around for many years and has since established its vital part in the robotics community, it is extensive and offers a lot of different functionalities. Only the very basics will be touched upon and used here.

The basic concept goes as follows. There are so called nodes, which can be anything from a temperature sensor to a whole robot, that process information and communicate over a peer-to-peer network. A node can publish and subscribe to so called topics, which can be thought of as channels like in radio communication. An example scenario could be a node that computes the inverse kinematics for a robot arm. This node would subscribe to a topic where some other node (or someone) publishes the arm's desired position. It would then do its calculations and publish the motor commands to a command topic, from where the subscribed individual motors take the information and execute the commands.

2.5 F1tenth

The F1/10 platform is a 1:10 scale open-source autonomous vehicle testbed [18]. It was developed to help researchers deploy and assess their autonomous driving algorithms in the physical world, without first having to engineer a testing vehicle from scratch. It furthermore facilitates others to quickly reproduce and improve upon results.

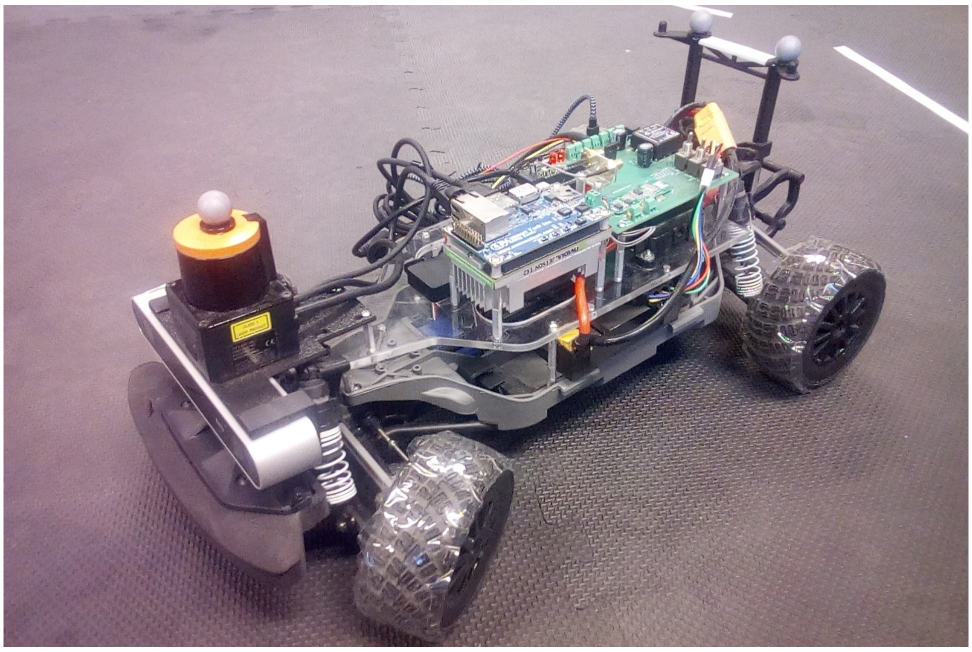


Figure 2.4: The F1tenth RC car used for drifting in the laboratory.

Build upon a TRAXXAS RC car's chassis, utilizing an NVIDIA embedded AI computing board and housing a variety of sensors, such as LiDAR and a depth camera, the testbed adequately resembles a full scale autonomous vehicle. Given its open-source nature and ROS based system architecture, the platform can also be easily expanded upon and modified.

2.6 OpenAI Gym

OpenAI Gym [8] is the group's contribution towards improving comparability for reinforcement learning research, by standardizing how information is passed between the task to be learned and the learning algorithm, illustrated in Figure 2.5. With providing an array of predefined and standardized environments, such as 'cart pole', 'lunar lander' and 'space invaders', the toolkit enables users to focus solely on developing and improving their learning algorithms. This in turn then also allows for quicker benchmarking of different publications trying to solve the same problem, instead of first having to try and recreate the problems implementation for each of them.

On top of that, users can also define their own learning environments by following a simple structure. The most important functionalities a custom Gym environment has to provide are *step* and *reset*. For every iteration the learning algorithm calls *step*, which takes in actions and returns consequential observations. In addition to that, calling *step* with some action also returns what reward this action induced and, when defined, if a certain goal was achieved. Once a target state is reached or the agent somehow failed to achieve its goal, *reset* allows users to revert everything to its initial state and have the agent try again, returning an initial observation.

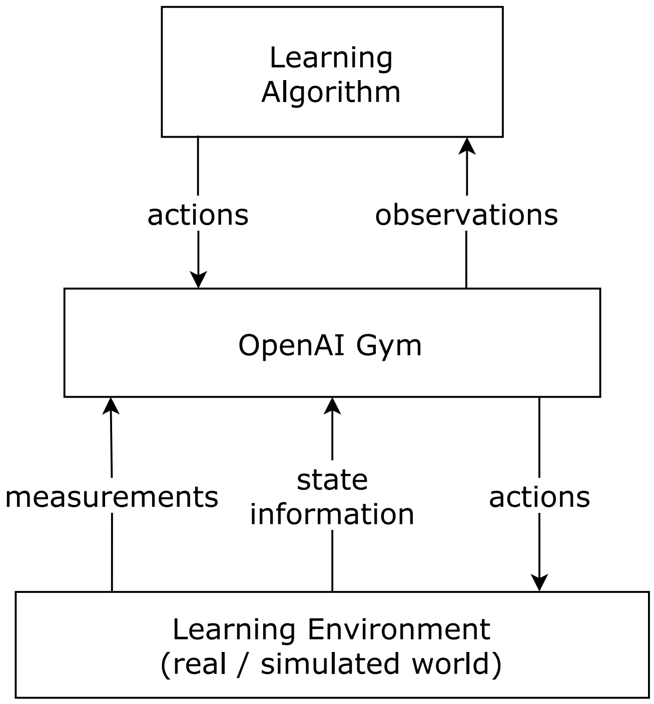


Figure 2.5: The OpenAI Gym acting as an interface between the actual learning environment (world the agent is in) and the underlying learning algorithm, allowing information to pass between them in a standardized way.

How and where exactly the custom Gym retrieves its measurements and state information is not part the specification. The 'Learning Environment' shown in Figure 2.5 is an abstraction of that. In a real world system, e.g., a car, there would be different measurements coming from different sensors all over car or even data from a GPS satellite.

The aforementioned Unity ML Agents uses this framework to offer the interface into custom environments build with the Unity Engine. Essentially enabling every Unity Engine application to be controlled through modern reinforcement learning algorithms, which regularly already make use of the OpenAI Gym interface.

2.7 Stable Baselines and PPO2

Stable Baselines [10] is a collection of TensorFlow implementations for state-of-the-art learning algorithms. It is based on OpenAI's Baselines [4], improving upon code-structure, documentation and usability. Implemented algorithms follow the OpenAI Gym interface, allowing users to train agents with very few lines of code. The learning behavior can be easily tweaked by providing (hyper-)parameters upon initialization, which allows for full control over the model. It also offers already trained agents for a lot of the standard Gym environments such as 'cart-pole' or 'space invaders'.

```
env = gym.make('CartPole-v1')

model = PPO2(env)
model.learn(parameter1, parameter2, ...)

model.save('my_trained_agent')
```

One of the algorithms included is PPO2, an improved version of OpenAI's PPO [14]. The general concept behind these Proximal Policy Optimization techniques is to not deviate from the current policy too much within a learning iteration, allowing for stable and consistent updates. PPO has proven to be successfull for a variety of different benchmark reinforcement learning problems and Unity ML Agents even employs PPO2 as its default learning algorithm.

Besides Reinforcement Learning, Stable Baselines also contains tools for Imitation Learning. This can, amongst other things, be used to initialize a neural networks weights and biases before training. This pretraining is similar to a regression, whereby a function's parameters are optimized such that it fits a set of datapoints as close as possible. In this case, the neural networks weights and biases are optimized such that the network produces a desired output given an input. It therefore is required to record observation-action pairs for the task at hand before training, meaning you need some form of expert or system that can, at least to some extent, already perform the task. With these collected data-pairs, the network's weights and biases are iteratively optimized such that its outputs match the recorded ones for a given input as close as possible, e.g. using a least-squares distance measure.

2.8 Interview with a recreational drifter

To gain a better understanding of how cars behave during a drift, how they can be controlled and to what extent that knowledge applies to simulated and RC vehicles, it was tremendously insightful to talk to someone who has sat behind a wheel for a big part of their life.

Ben 'KameTrick' Craft [3], a video creator with over 15 years of drifting experience (real, simulated and RC) was able to share some of his know-how to help develop a grasp of what is important for drifting. His suggestions and driver's point-of-view helped to shape the inputs and outputs of the controller, pointing out what information a driver utilizes when drifting. He also provided feedback and suggestions regarding vehicle setup, for example pointing out that using the handbrake over the footbrake could yield better drifting performance as the handbrake usually only locks the rear wheels, enabling the back of the car to break out. Though for some tracks, braking might not be necessary at all. He also highlighted the significance of the car's maximum steering angle, regarding maneuverability and staying in control as the sideslip angle increases.

3

Controller architecture

This chapter introduces the underlying controller and neural network architecture and their implementation. The overall objective is split into two components, similar to how drivers are rated in drifting competitions: following a general path and maximizing the sideslip angle. The path representation is explained first, followed by a detailed look at the observation and action space. Using this knowledge, the reward function is derived and the general neural network structure established. In addition to that, the implementation of the acquired controller on the RC car is described.

3.1 Path representation

The path the car has to follow is represented by an array of points in 2D space $(x, y)^T \in \mathbb{R}^2$ which will be referred to as waypoints from now on [20]. These waypoints are spaced approximately one car length apart by default, this waypoint spacing will be denoted as δ. Placed waypoints can now resemble almost any trajectory a car might follow, from a simple circle to a figure-8, all the way to real life racetracks and roads.

3.2 Observation and action space

The following will take a detailed look at what observations and actions are used for drifting, explaining why and in what range of values they operate.

Observation space

The observation space is continuous. To enable controller transfer, generalizability and a stable training process [10], all observations are normalized to be within the interval $[-1, 1]$. This is usually done by dividing an observation by its, sometimes empirically determined, maximum value. The main idea behind choosing the following observations as input for the learning algorithm is to mimic what a human driver uses, while also restricting the use of additional measuring equipment like wheel traction sensors. Also, since a lot of inputs can result in higher computational effort and longer learning periods, the amount of observations is kept to a minimum.

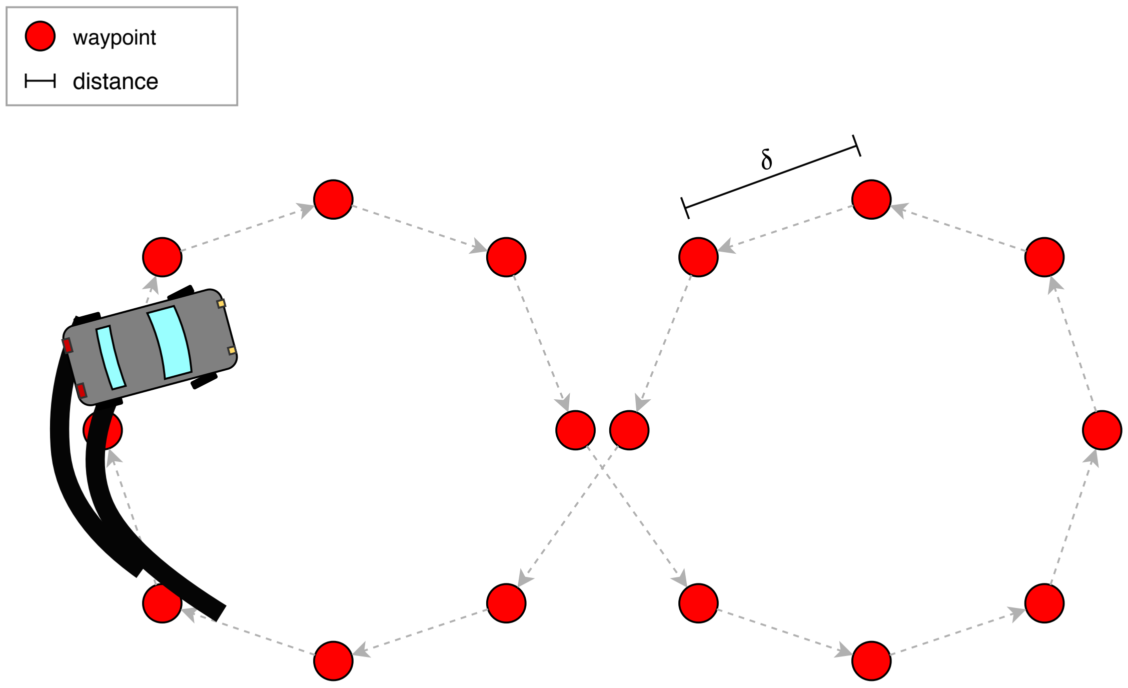


Figure 3.1: Representation of a figure-8 shaped path using evenly spaced waypoints with a distance of δ.

Human drivers rely heavily on feeling the G-forces as they drift. This is modeled by providing the car's **angular velocity** and **sideslip angle** as observations [15, 20, 21]. To achieve a first person perspective of the road, like a human would have, the algorithm is also fed in x and y coordinate of the **next n waypoints** ($n \in \mathbb{N}$), transformed from the world to the car's local coordinate system [7]

$$\begin{pmatrix} \hat{x}_{\text{wp}} \\ \hat{y}_{\text{wp}} \end{pmatrix} = \begin{pmatrix} \cos(\theta_{\text{car}}) & -\sin(\theta_{\text{car}}) \\ \sin(\theta_{\text{car}}) & \cos(\theta_{\text{car}}) \end{pmatrix} \left(\begin{pmatrix} x_{\text{car}} \\ y_{\text{car}} \end{pmatrix} - \begin{pmatrix} x_{\text{wp}} \\ y_{\text{wp}} \end{pmatrix} \right) \tag{3.1}$$

where θ_{car} is the car's rotation about the world's vertical axis (yaw angle)
$x_{\text{wp}}, y_{\text{wp}}$ are the waypoint's world coordinates
$x_{\text{car}}, y_{\text{car}}$ are the car's world coordinates
$\hat{x}_{\text{wp}}, \hat{y}_{\text{wp}}$ are the waypoints's local coordinates.

For $n = 6$ and a waypoint spacing of 5 meters, the car can observe $6 \times 5 = 30$ meters of the road ahead, which is roughly what a driver would take into account. Each coordinate of the n waypoints is normalized through diving it by the distance to the last of these n waypoints. This distance is equal to $n \times \delta$, in other words: the amount of observed waypoints times the distance between the waypoints. For the above example this would mean that every waypoint's coordinate is divided by 30.

Since there usually is a delay between giving full throttle and the engine coming up to speed, the car's revolutions per minute (**RPM**) are also used as an input. Similar to this, there also is the car's inertia, which means that even at full throttle it will take some time until the maximum velocity is reached. Because of this, the car's **local velocity** in x and y direction is also fed in. On top of that, steering takes some time as well, which is why the **current steering angle** is used as an observation. This however is only the case for the simulated car, as the RC car has almost instantaneous steering, allowing the steering angle observation to be omitted.

Action space

The learned controller has two continuous outputs, both in the range $[-1, 1]$. One of them is interpreted as the **steering command**, where -1 means turning the steering wheel all the way to the left, and 1 means turning it to the right and everything in between accordingly. The other one is interpreted as the **throttle command** and, if desired, also the (hand-)brake. Everything greater than 0 is interpreted as throttle, while everything below is used for braking. Though braking is not used by default.

Both outputs are multiplied with their respective maximum to achieve the actual control command. For steering this could be 30°, thus producing steering commands in the range $[-30, 30]$. For a car with a maximum RPM of 9000 this would produce throttle commands in $[0, 9000]$, while braking would be treated similarly.

3.3 Reward function

The backbone of every Reinforcement Learning problem is its reward function. Through it the agent can get an understanding of which behavior is desired and which is not. The reward function used here consists of the two parts mentioned above: following a predefined path and maximizing the sideslip angle. However these parts are not weighted equally, as for drifting it is usually more important to hold an angle than it is to exactly follow the racing line. Therefore the overall reward function can be written as

$$f_{\text{rew}}\left(\vec{W}_{\text{c}}, \vec{W}_{\text{n}}, \vec{W}_{\text{p}}, \vec{C}, \beta\right) = w_1 f_{\text{path}}\left(\vec{W}_{\text{c}}, \vec{W}_{\text{p}}, \vec{C}\right) + w_2 f_{\text{angle}}\left(\vec{W}_{\text{c}}, \vec{W}_{\text{n}}, \vec{W}_{\text{p}}, \beta\right) \tag{3.2}$$

where $\vec{W}_{\text{c}}, \vec{W}_{\text{n}}, \vec{W}_{\text{p}}$ are the current, next and previous waypoint respectively
$\vec{C}$ is the car's world position
β is the car's current sideslip angle.

The weights $w_1, w_2 \in [0, 1]$ are selected such that $w_1 + w_2 = 1$, ensuring f_{rew} itself stays within $[0, 1]$. For drifting, these are chosen as $w_1 = \frac{1}{16}$ and $w_2 = \frac{15}{16}$, heavily favoring the angle part of the reward. This weight distribution has proven to reduce the probability of getting stuck in the local optimum of driving without sliding, as with equal weights already half of the reward could be achieved by only passing though the waypoints, i.e., without actually drifting. This reward function is evaluated only when the car passes the current waypoint.

Path following component

To measure how well the car follows a desired path, each waypoint is interpreted as a line - similar to a finish line - where the original coordinates of the waypoint are the center of the line and σ is half its length. The line length can be adjusted depending on the width of the road, though a good starting value for σ is half the car's length. Start- and endpoint are calculated such that there is a 90° angle between the line and another imaginary line, connecting the previous and current waypoint, as seen in Figure 3.2. Because of this right angle, a waypoint's line will be referred to as its orthogonal line (OL) hereafter.

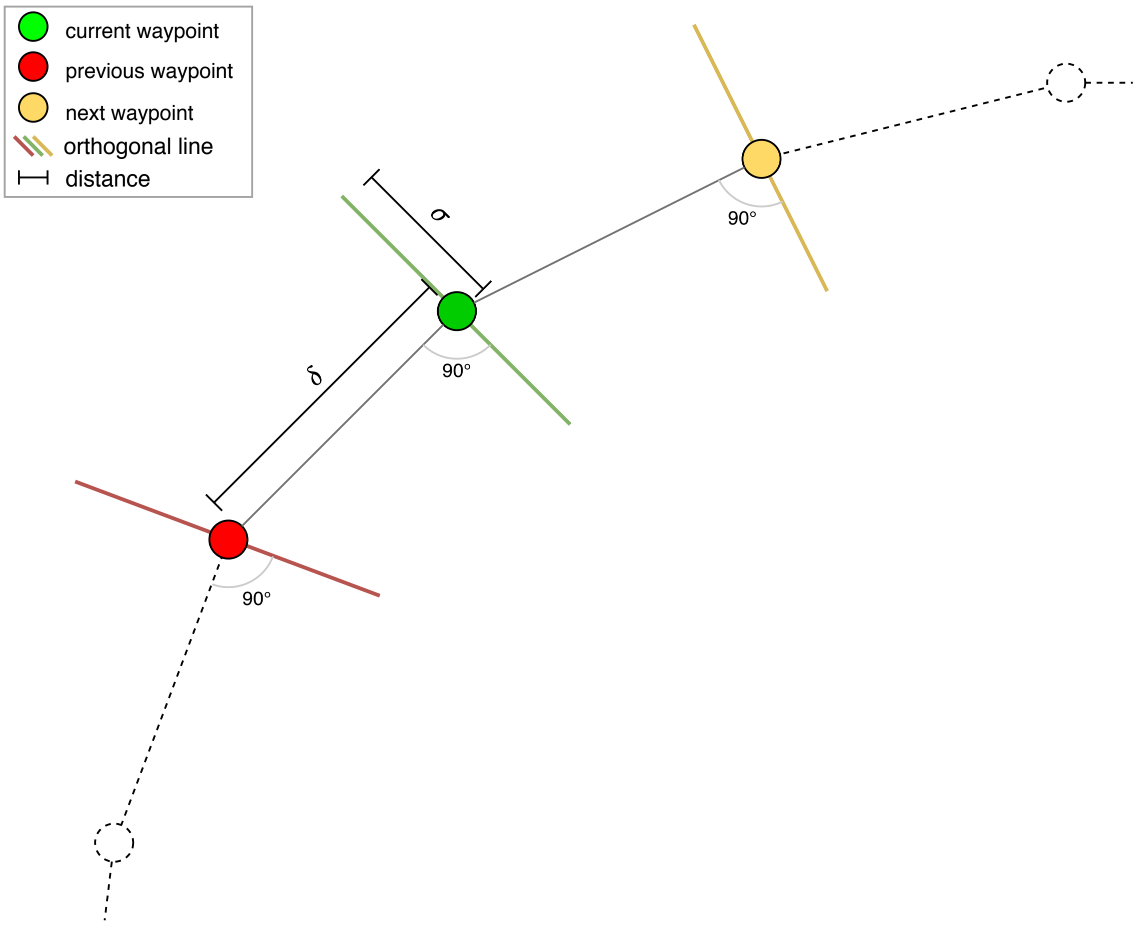


Figure 3.2: Previous, current and next waypoint with their respective orthogonal lines and half its length σ.

The OL is determined by first normalizing the vector $\overrightarrow{PC} := \overrightarrow{W}_c - \overrightarrow{W}_p$ which points from the previous to current waypoint. Second, finding vector $\overrightarrow{U} := \overrightarrow{PC}^{\perp}$, which is perpendicular to $\overrightarrow{PC}$. And third, placing the two ends of the OL opposite to each other along $\overrightarrow{U}$, originating at the current waypoint $\overrightarrow{W}_c$, according to the desired road width σ. The orthogonal line's start $\vec{S}$ and end $\vec{E}$ are therefore determined as

$$\begin{aligned} \vec{S} &= \overrightarrow{W}_c - \sigma\overrightarrow{U} \\ \vec{E} &= \overrightarrow{W}_c + \sigma\overrightarrow{U}. \end{aligned} \tag{3.3}$$

For projecting the car's position $\vec{C}$ onto the OL, the vectors $\overrightarrow{CS} = \vec{S} - \vec{C}$ and $\overrightarrow{SE} = \vec{E} - \vec{S}$ are introduced. Utilizing the dot-product [5], the projected point $\vec{P} := (x_{\text{proj}}, y_{\text{proj}})$ on the OL is computed as

$$\vec{P} = \vec{S} + \frac{\overrightarrow{CS} \cdot \overrightarrow{SE}}{\overrightarrow{SE} \cdot \overrightarrow{SE}} \overrightarrow{SE}. \tag{3.4}$$

Finally, the Euclidean distance between $\vec{C}$ and $\vec{P}$ yields $d_{\text{projection}}$, which is used to determine how close the car is to the current OL and therefore if it passed this waypoint and a reward should be given. To encourage it to stay close to the center of the predefined path, the reward is dependent on the car's distance to the path center. More precisely, the Euclidean distance, denoted as d_{center}, from $\vec{P}$ to the center of the OL, i.e., $\overrightarrow{W}_{\text{c}}$. The further a vehicle deviates from the center, the less reward it is given. When it misses the orthogonal line entirely ($d_{\text{center}} > \sigma$), it is not rewarded and the waypoint does not count as passed.

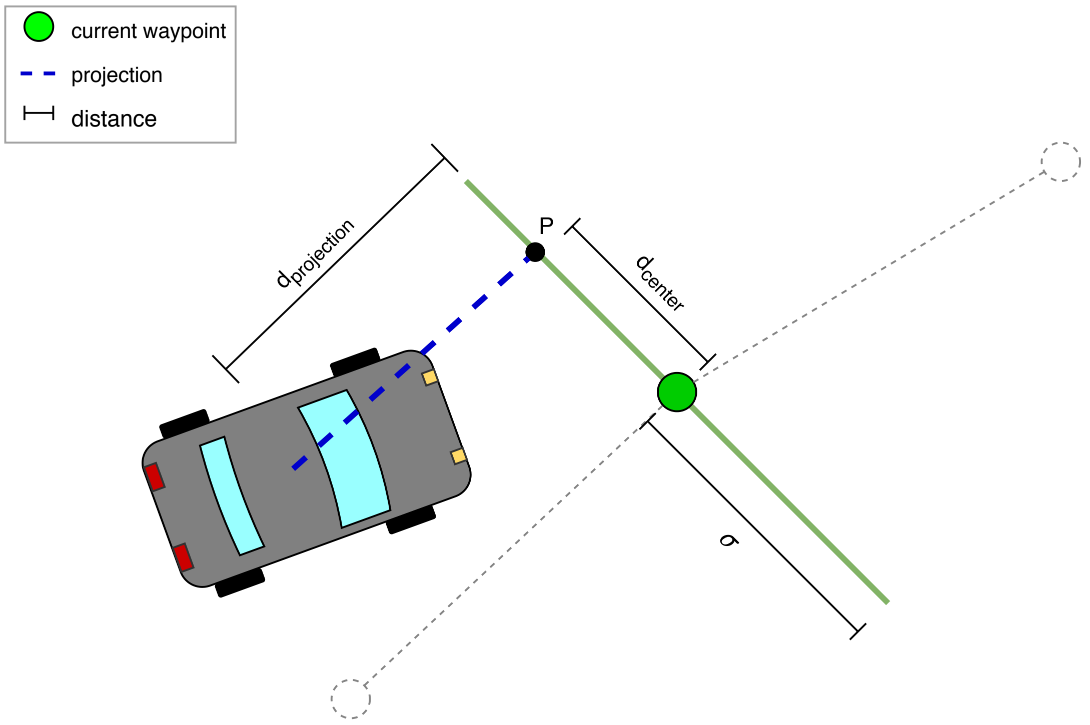


Figure 3.3: The car's position being projected onto the current waypoint's orthogonal line. $d_{\text{projection}}$ being the distance from the car to the OL, d_{center} the distance from the projected point $\vec{P}$ to the waypoint.

The final path following component of the reward function is computed as

$$f_{\text{path}}\left(\vec{W}_c, \vec{W}_p, \vec{C}\right) = \exp\left(-\rho\left(\frac{d_{\text{center}}}{\sigma}\right)^2\right) \tag{3.5}$$

where $d_{\text{center}} = \sqrt{(x_{\text{wp}} - x_{\text{proj}})^2 + (y_{\text{wp}} - y_{\text{proj}})^2}$

$x_{\text{wp}}, y_{\text{wp}}$ are the current waypoint's world coordinates $\left(\vec{W}_c\right)$

$x_{\text{proj}}, y_{\text{proj}}$ are the world coordinates of $\vec{P}$

ρ controls how much of the area under the curve lies within $\pm\,\sigma$.

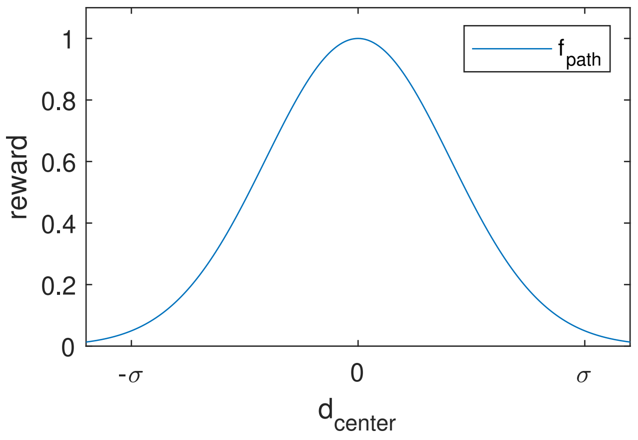


Figure 3.4: Path following component of the reward function (3.5) for $\rho = 3$.

Angle maximization component

The angle maximization component of the reward function follows the simple idea: more sideslip angle equals more reward. A very basic implementation of this is

$$f_{\text{angle}}(\beta) = \frac{1}{180}|\beta| \tag{3.6}$$

Although this can already yield some results, there is a variety of improvements to be made. One of them has to do the sideslip angle β, which per definition is 0° when the car is driving forward in a straight line and ±90° when it is only moving sideways at a 90° angle to where it is facing. The problem here arises when $|\beta| \gg 90°$ because then the car would start to move backwards or spin out of control, which are both not desired here. Therefore (3.6) has to be modified by imposing an upper limit. This limit cannot be chosen uniformly for all cars and all driving conditions, since there are corners and situations where a very high sideslip angle can be of use. Also, cars with higher steering angle can maintain higher β without spinning out of control, as mentioned in section 2.8.

Keeping all of this in mind, a good compromise is limiting the rewarded sideslip to 100° - everything above will be not be rewarded. This then changes (3.6) to

$$f_{\text{angle}}(\beta) = \begin{cases} \frac{1}{100}|\beta| & \text{if } |\beta| \leq 100^\circ \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

Another improvement to be made is regarding the kinematic sideslip angle β_{kin}. The problem with (3.7) is that, due to β_{kin}, a car driving entirely without sliding can achieve a good amount of reward already. Furthermore, one actually sliding, but at a lower angle, will not achieve as much reward. This has been experimentally proven to prompt the learning algorithm to either staying in the local optimum of not sliding for longer, increasing learning time, or deciding against drifting altogether. Since differentiating between what part of the overall sideslip angle is due to do steering and what part originates from actually sliding is difficult, without using wheel slip sensors, a simple solution is to only reward angles greater than the maximum kinematic sideslip angle of the car. A good default here is 20°, but the exact value can be computed using (2.2) by plugging in the maximum steering angle of the vehicle in question. Including this change results in

$$f_{\text{angle}}(\beta) = \begin{cases} \frac{1}{100}|\beta| & \text{if } \beta_{\text{kin}} \leq |\beta| \leq 100^\circ \\ 0 & \text{otherwise.} \end{cases} \tag{3.8}$$

The final improvement aims at further decreasing learning time by eliminating another local optimum. While drifting around a corner, a vehicle's back breaks out towards the outside of the corner. The current reward function however also gives the same reward for breaking out towards the inside of the corner, which is actually counter productive towards drifting. To counteract this, the path curvature factor γ is introduced to the angle reward.

Determining if a vehicle's back is breaking out in the correct direction is achieved by comparing the road curvature's sign with the current sideslip angle's sign. A road segment's curvature is computed from three points: the previous, current and next waypoint. An imaginary line is drawn between the previous and next waypoint, leaving the current one to be on either side of the line. If it is on the left of the line, the path is curving right and vice-versa. The edge-case of it being exactly on the line will be handled separately.

(3.9) is used to determine d_{curve}, which is the signed distance from the current waypoint to the imaginary line between the previous and next waypoint and utilizes a perpendicular vector and the dot-product [12]. This distance provides information about how much this particular road segment is curving and if it is curving left or right and is computed as

$$d_{\text{curve}} = \left(\vec{W}_{\text{c}} - \vec{W}_{\text{p}}\right) \cdot \left(\vec{W}_{\text{n}} - \vec{W}_{\text{p}}\right)^{\perp}. \tag{3.9}$$

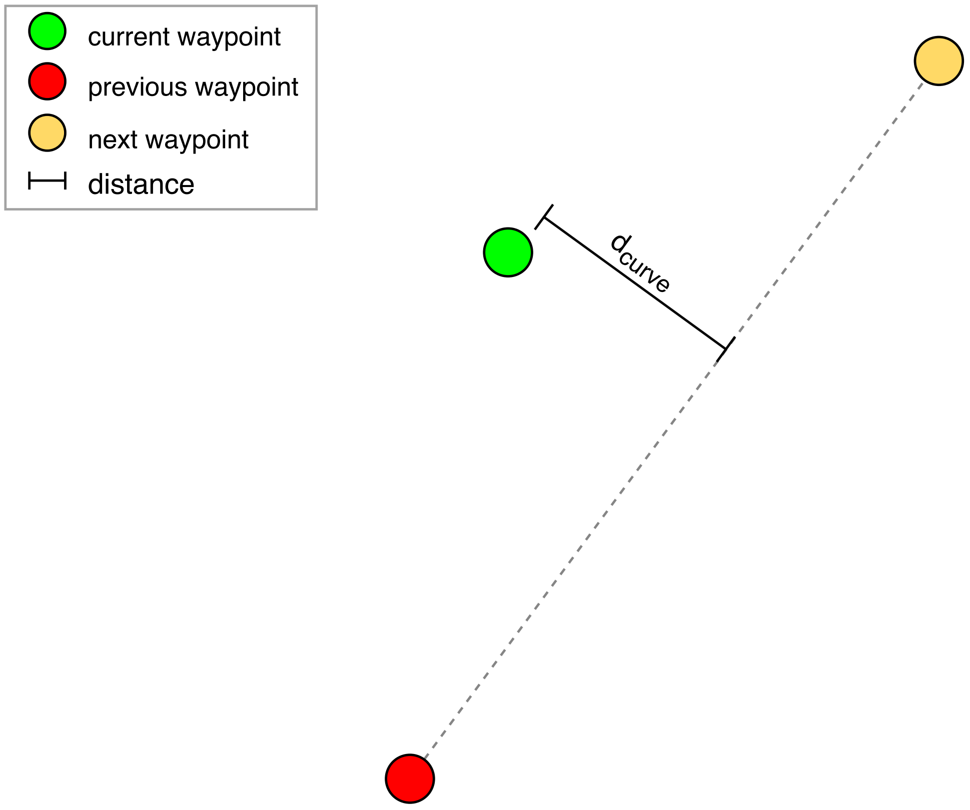


Figure 3.5: Distance between the current waypoint and a line from previous to next waypoint, used for determining the curvature of this path section.

Since the all three waypoints are aligned when the road is straight, this distance can also be zero. In this case, and in the ones where d_{curve} is negligibly small, it does not make sense to impose a correct direction for the car's back to break out in. Because of this reason, γ is kept neutral, which means equal to 1, for straight and almost straight parts of the road. Deciding whether a road segment is straight or not, is achieved by checking if the absolute value of d_{curve} falls below a certain threshold τ. This threshold is dependent on the waypoint's spacing and the specific use-case. For some curvy racetracks there might be road segments that are fairly straight and should not be drifted, even though they have a small amount of curvature. A good default for τ is $\frac{1}{4}$ of the waypoint's spacing δ, though it can, for simplicity, be omitted entirely by setting $\tau = 0$.

To now find if the car's back is breaking out in the correct direction, the signs of d_{curve} and β are compared. If, for example, d_{curve} is positive, meaning the road curves to the right, and β is also positive, γ is set to 1 because the car is breaking out in the *correct* direction. If the road is straight ($|d_{\text{curve}}| < \tau$), γ is immediately set to 1, as there is no correct β. For cases where their signs differ, γ is set to 0, meaning the car's back is breaking out in the wrong direction and will not be rewarded. This can be expressed as

$$\gamma = \begin{cases} 1 & \text{if } \operatorname{sign}(\beta) = \operatorname{sign}(d_{\text{curve}}) \text{ or } d_{\text{curve}} < \tau \\ 0 & \text{otherwise.} \end{cases} \tag{3.10}$$

γ is then multiplied with the sideslip angle reward from (3.8), only giving a reward when breaking out in the correct direction or the road being straight. This makes up the final angle maximization component of the reward function

$$f_{\text{angle}}\left(\overrightarrow{W}_{\text{c}}, \overrightarrow{W}_{\text{n}}, \overrightarrow{W}_{\text{p}}, \beta\right) = \begin{cases} \gamma \frac{1}{100}|\beta| & \text{if } \beta_{\text{kin}} \leq |\beta| \leq 100^\circ \\ 0 & \text{otherwise.} \end{cases} \tag{3.11}$$

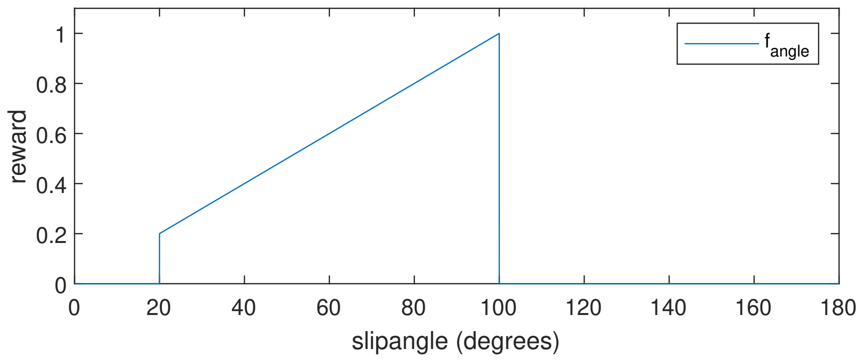


Figure 3.6: Angle maximization component of the reward function (3.11) for $\beta_{\text{kin}} = 20$ and $\gamma = 1$.

3.4 RC car implementation

Because of the RC car used being four-wheel-drive, it first has to be modified to be rear-wheel-drive. This is achieved by taking out the center driveshaft. Since the laboratory floor is made up of rubber mats, meant to hinder loss of tracking in other experiments, the vehicle's rubber tires are wrapped with packaging tape to allow sliding, as seen in Figure 2.4. The car's position, orientation and velocity are measured via an optical tracking system. This information is relayed to a laptop running the learning algorithm and finally steering and motor commands are broadcast to the car. ROS is used to communicate between the different network nodes. Apart from some minor details, everything is implemented on the RC car exactly as it is in the simulation. This allows for a seamless transfer of controllers between the two.

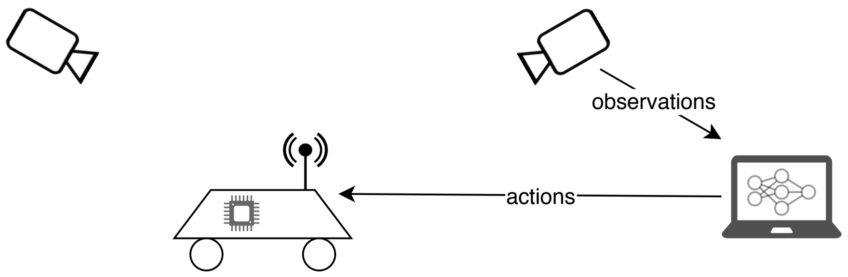


Figure 3.7: Sketch of the RC car implementation.

3.5 Neural network structure

Stable Baseline's PPO2 is employed as the learning algorithm. A network with 3 fully connected layers and 128 neurons each is used, as well as one with 3 layers and 256 neurons for more complex behaviors. These network sizes were determined empirically, slowly incrementing from the Stable Baseline's default 2 fully connected layers with 64 neurons each. The activation function was also left at the default, which is the hyperbolic tangent. A learning rate of 0.003 is used for training, which is conducted in real time with only 1 vehicle at a time on consumer hardware. A high discount factor of 0.999 demands the algorithm to also take future states into account when making decisions. The network is asked to make a decision 10 times per second. During training, a policy update is conducted only every 512 timesteps. This update frequency was also empirically determined, lower values correlated with an unstable training progress.

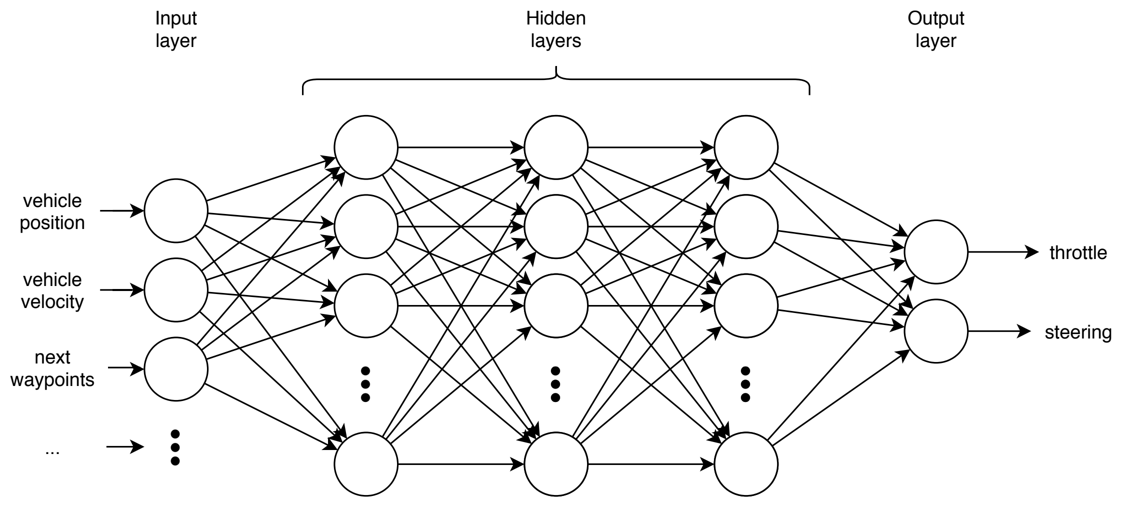


Figure 3.8: Illustration of the neural network architecture used.

4

Results

This chapter evaluates the above described controller architecture. It first looks at circular paths and assess the controllers' robustness to changing physical conditions like maximum steering angle and changing friction values, while also highlighting what effect pretraining has on overall training time. Then, to gauge generalizability and transferability, performance and on a figure-8 and an arbitrary racetrack are shown, while also investigating different size neural networks. At last, results on the RC car are analyzed. The following drifting controllers run at a decision frequency of 10 Hz, which equals 10 timesteps per second. They generally implement the default settings and parameters discussed in chapter 3.

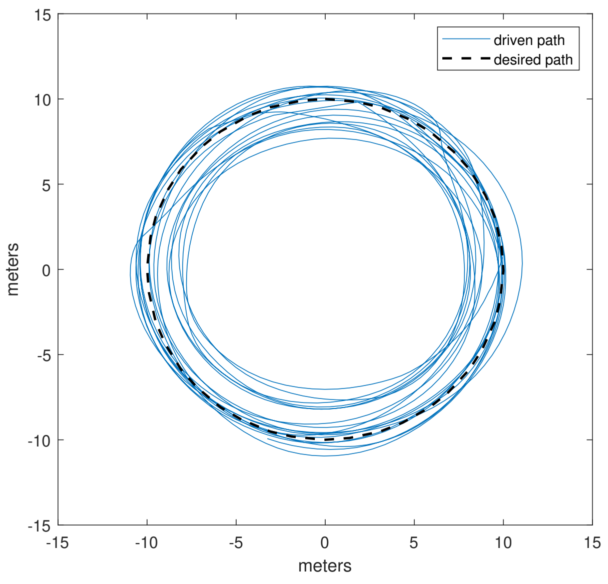
4.1 Simulation

The waypoint spacing for the simulation is set to the vehicle's length ($\delta = 5\text{m}$) and the controller is fed in the next 6 waypoints. The car is equipped with an automatic transmission that is limited to only using first and second gear. Since the steering angle usually cannot be changed instantly on real cars, steering speed is limited to 150 degrees per second. During training, the maximum episode length is 1500 timesteps but the vehicle is reset when it spins out of control or deviates from the desired trajectory too much.

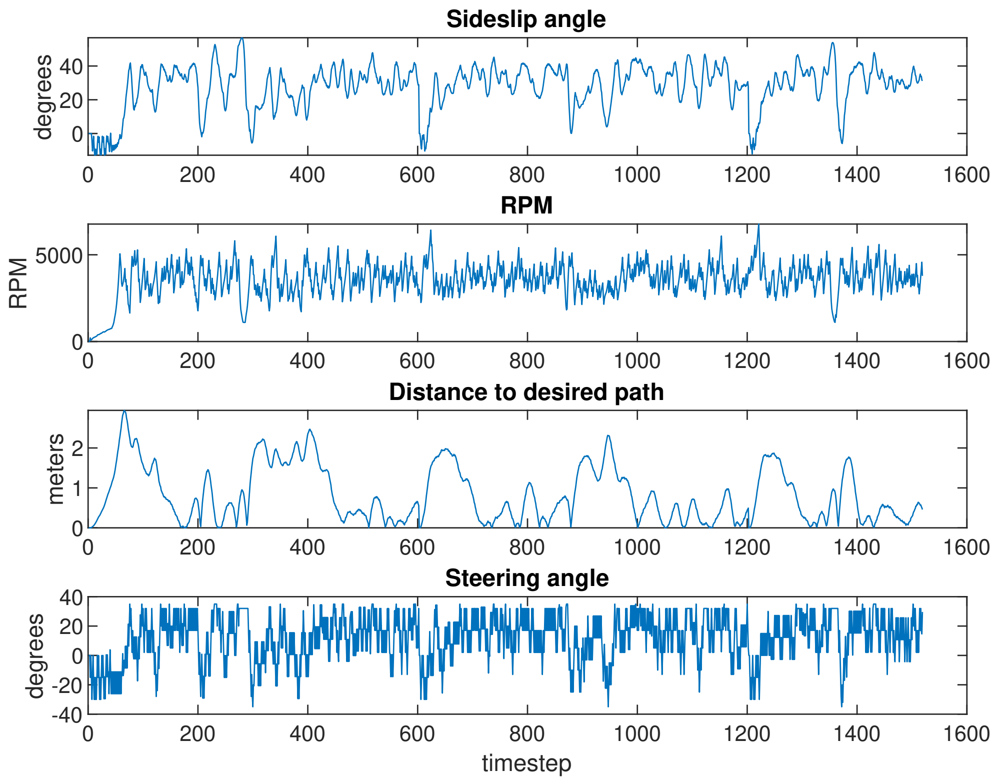
Circles

The baseline controller is able to maintain a sideslip angle between 20° and 40° drifting counterclockwise on a circular path with a constant radius of 10 meters, as shown in Figure 4.1. The car deviates from the desired path by at most 2.5 meters, though often staying within less than 1 meter.

Name	Maximum steering angle	Friction coefficient	Neural net. size	Training	Training path shape
baseline	35°	$\mu_{\text{adh}} = 0.67$ $\mu_{\text{peak}} = 0.8$ $\mu_{\text{lim}} = 0.6$	3×128	from scratch	10m radius circle



(a) Desired path and the actual measured vehicle position.



(b) 2.5 minutes of the measured sideslip angle, RPM, distance to desired path and steering angle.

Figure 4.1: Performance of the baseline controller drifting a 10 meter radius circle.

The vehicle's sideslip angle occasionally surpasses the amount it can compensate with countersteering, which would usually result in loss of control and spinning out. However the controller learned to recognize this and is able to catch the car before spinning out, by removing throttle and accelerating once control is regained. The sideslip angle therefore drops, sometimes returning to normal driving, before quickly coming back up. This behavior can be seen at around timestep 300 and 1400, while also showing how the RPM increases after the sideslip angle drops, i.e., the car accelerating to loose traction and throw its back out further, at timestep 600 and 1200. The steering angle oscillates between 0° and the maximum 35°, meaning the vehicle is steering to the right, i.e., countersteering as it is driving a left turn. Drops in steering angle correlate with drops in sideslip angle as the vehicle tries to stabilize itself.

Figure 4.2 displays the controller's throttle command and resulting RPM, as well as the steering command and resulting steering angle, on an excerpt of 50 timesteps. Both the throttle and steering command output of the controller largely consist of the action space's maximum values, i.e., 0 and 1 for throttle or -1 and 1 for steering. The controller uses the throttle command to keep the RPM in its desired range, mostly by outputting full or no throttle. For steering it behaves similarly, mostly outputting full right or full left. This becomes especially apparent at timesteps 5 through 10 and 15 where it is holding the current steering angle by alternating between extreme steering commands, rather than outputting the actual desired continuous value.

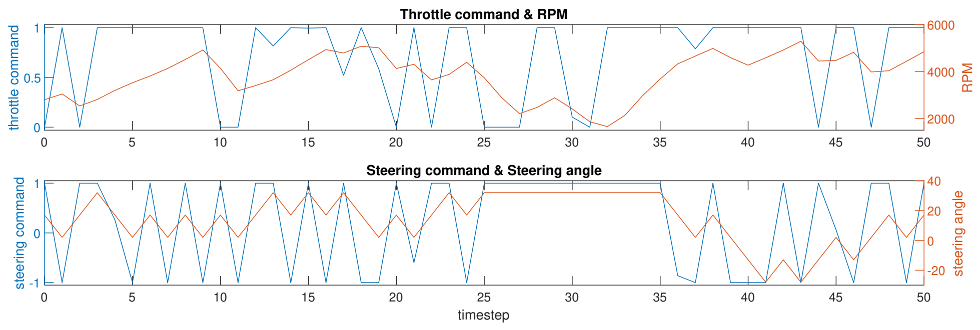


Figure 4.2: 5 seconds of throttle command (blue) vs. resulting RPM (orange) and steering command (blue) vs. resulting steering angle (orange).

Pretraining

Pretraining is employed to inject existing knowledge and skill into a new controller before beginning the regular training progress, aiming to reduce overall training time. This is done by recording observation-action data pairs while the vehicle is being controlled externally, by an existing controller or with human keyboard input from the author, for a few minutes. These datasets are used to pretrain the network with Imitation Learning, as described in section 2.7, for about 10 minutes. This also applies to other occasions where pretraining is used. Figure 4.3 shows the smoothed training progress for a car learning to drift a circular path, given 3 different levels of pretraining. The vehicle setup is exactly the same for all runs, only distinguishable by the pretraining received.

Name	Maximum steering angle	Friction coefficient	Neural net. size	Training	Training path shape
pretrain-none	35°	$\mu_{\mathrm{adh}} = 0.67$ $\mu_{\mathrm{peak}} = 0.8$ $\mu_{\mathrm{lim}} = 0.6$	3×128	**from scratch**	10m radius circle
pretrain-hum	35°	$\mu_{\mathrm{adh}} = 0.67$ $\mu_{\mathrm{peak}} = 0.8$ $\mu_{\mathrm{lim}} = 0.6$	3×128	**pretrained with: human data; + further training**	10m radius circle
pretrain-contr	35°	$\mu_{\mathrm{adh}} = 0.67$ $\mu_{\mathrm{peak}} = 0.8$ $\mu_{\mathrm{lim}} = 0.6$	3×128	**pretrained with: baseline; + further training**	10m radius circle

Learning entirely without pretraining, i.e., with randomly initialized neural network weights and biases, takes about 650k timesteps, which equals 18 hours of realtime training, reaching an average episode reward of 26. Pretraining with human data takes about $\frac{2}{3}$ of that, while with data of an already trained controller, acting as a professional driver, it only takes $\frac{1}{10}$ of the time to reach the same average reward. Training without any pretraining makes very little progress for the first 300k timesteps, as the controller has to learn that actually moving is required to be rewarded. It first learns to drive very careful close to the desired path, as this is immediately rewarded. Between 300k and 400k timesteps though it realizes that sliding sideways results in a higher reward. Pretraining with human example driving results in a much steeper learning curve in the beginning, as the controller already knows what it has to do to achieve a reward. However, the average reward occasionally decreases as the controller also learned to make the same driving mistakes the human did and has to learn to correct those on its own. Given a few minutes of an already trained controller driving, the new controller quickly and directly learns to drift and score a high reward.

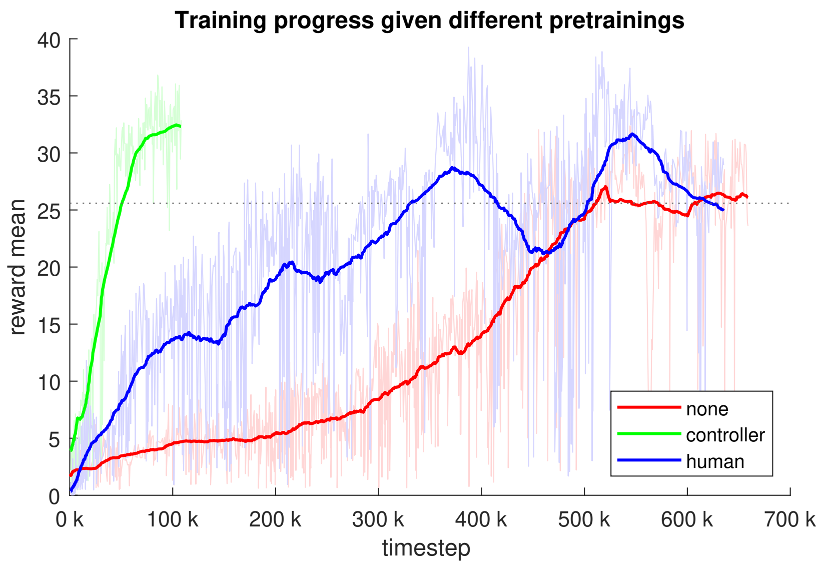


Figure 4.3: Training progress for 3 different levels of pretraining: without (red), human data (blue) and a fully trained controller (green).

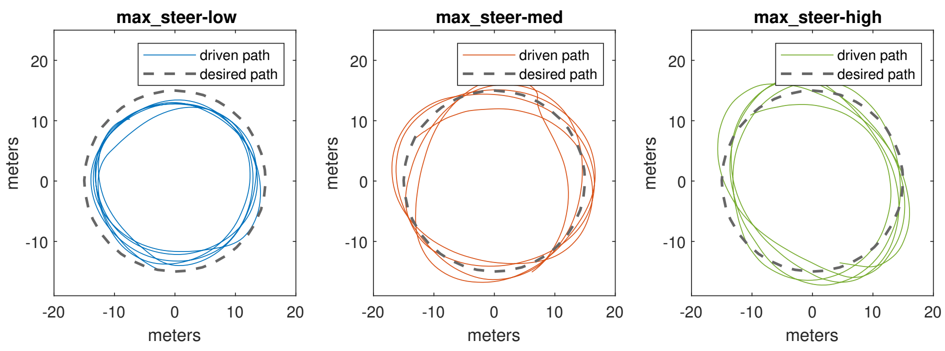
Maximum steering angle

Figure 4.4 portraits 3 individually trained controllers with different maximum steering angles, drifting the same 15 meter radius circle. The three controllers were trained on circles ranging from 10 to 20 meter radii.

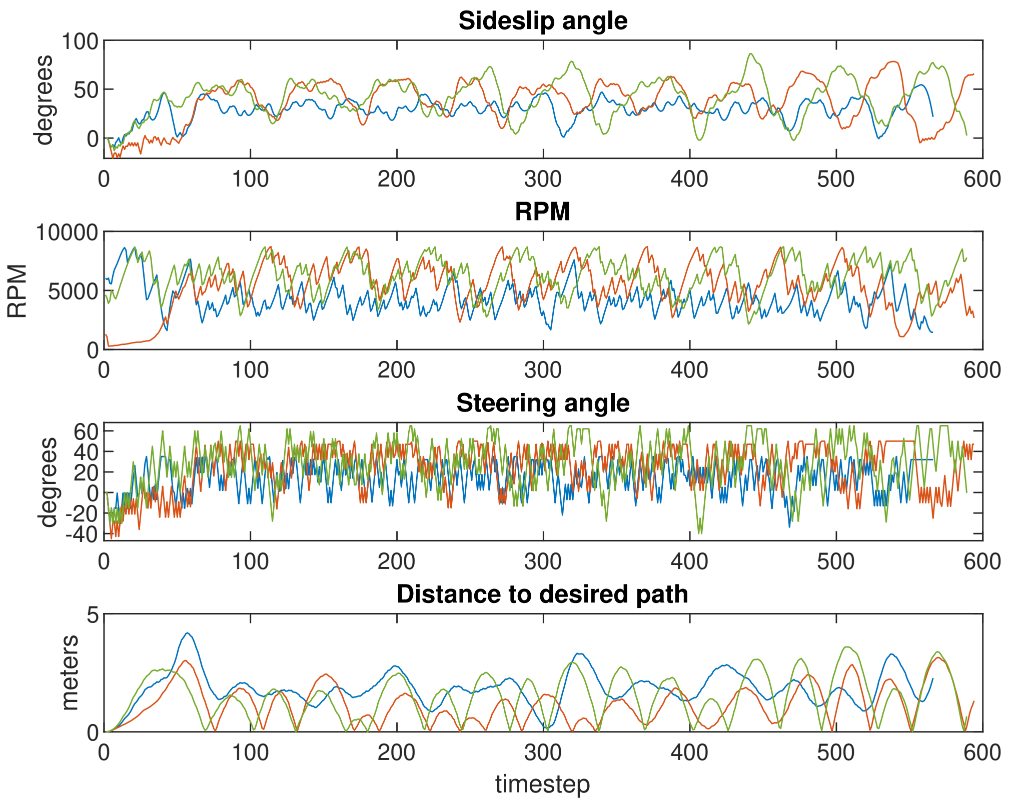
Name	Maximum steering angle	Friction coefficient	Neural net. size	Training	Training path shape
max_steer-low	**35°**	$\mu_{\text{adh}} = 0.67$ $\mu_{\text{peak}} = 0.8$ $\mu_{\text{lim}} = 0.6$	3 × 128	pretrained with: baseline; + further training	10-20m radius circle
max_steer-med	**50°**	$\mu_{\text{adh}} = 0.67$ $\mu_{\text{peak}} = 0.8$ $\mu_{\text{lim}} = 0.6$	3 × 128	pretrained with: baseline; + further training	10-20m radius circle
max_steer-high	**65°**	$\mu_{\text{adh}} = 0.67$ $\mu_{\text{peak}} = 0.8$ $\mu_{\text{lim}} = 0.6$	3 × 128	pretrained with: baseline; + further training	10-20m radius circle

The max_steer-low one is limited to 35° of steering, which is a common value among standard cars. This particular controller stays within the circle a lot and is therefore oscillating close to a distance of 2 meters to the desired path. Its sideslip angle stays within $20° - 30°$, while holding the RPM close to 3500. It does keep its steering angle between 0° and 35°, performing countersteering and correcting when necessary. The second controller (max_steer-med) is given 50° of maximum steering angle and also only deviates from the desired path by at most 2 meters, though oscillating around a distance of 0. Therefore its distance to the path is not as constant as the first controller's, but it generally stays closer to the desired trajectory. Its sideslip fluctuates between 25° and 50°, while its RPM range from 4000 to 8000, which is close to the maximum for this particular car. Steering behaves similar to the first controller, adequately utilizing the maximum available angle while also correcting in between. The max_steer-high controller has 65° of steering angle available and developed the behavior of drifting a slight oval, rather than exactly matching the desired circular path. This can be seen in the distance to desired path as a rather small distance is usually followed by a bigger peak, followed by another small one, and so forth. These peaks correlate to the car being close to the desired path at the narrow parts of its oval and being further from it in the wider spots. This third controller's RPM and sideslip angle do not differ much from the second's, though the sideslip occasionally peaks at up to 60°. It does not use its maximum steering angle as often as the other two controllers and sometimes even has to steer in the opposite direction to correct itself.

All three controllers show some form of oscillating, trying to reach a sideslip angle as high as possible without loosing control over the vehicle. None of them spins out while trying to do so. Even when a loss of control seems imminent, decelerating allows them to stabilize the car, though shortly dropping in sideslip. This can be seen for the 50° steering angle controller (max_steer-med) at timestep 550.



(a) Desired path and the actual measured vehicle position for the 35° (blue), 50° (orange) and 65° (green) steering angle controller.



(b) The controllers' sideslip angle, RPM, steering angle and distance to desired path.

Figure 4.4: 15m radius circle drifting performance of 3 different controllers with different maximum steering angles.

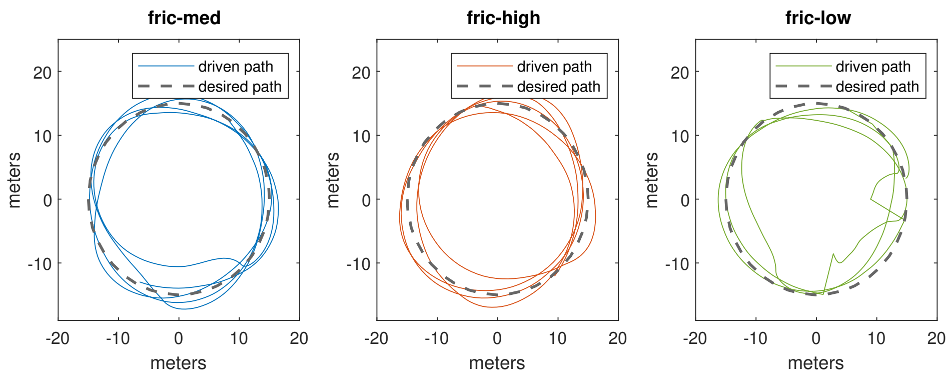
Changing friction values

Figure 4.5 depicts one single controller trying to drift under different friction conditions. It (fric-med) was trained on 10 to 20 meter radius circles, a maximum steering angle of 50° and the baseline limit friction μ_{lim} of 0.8. The other 2 controllers are merely a copy of this one, the only difference being the changed friction value of their tires. This is meant to showcase how a trained controller can adapt, without further training, to deviations from the physical parameters it was originally trained on. It further indicates the controller did learn the actual skill of drifting, rather than only having adapted to these exact physical parameters.

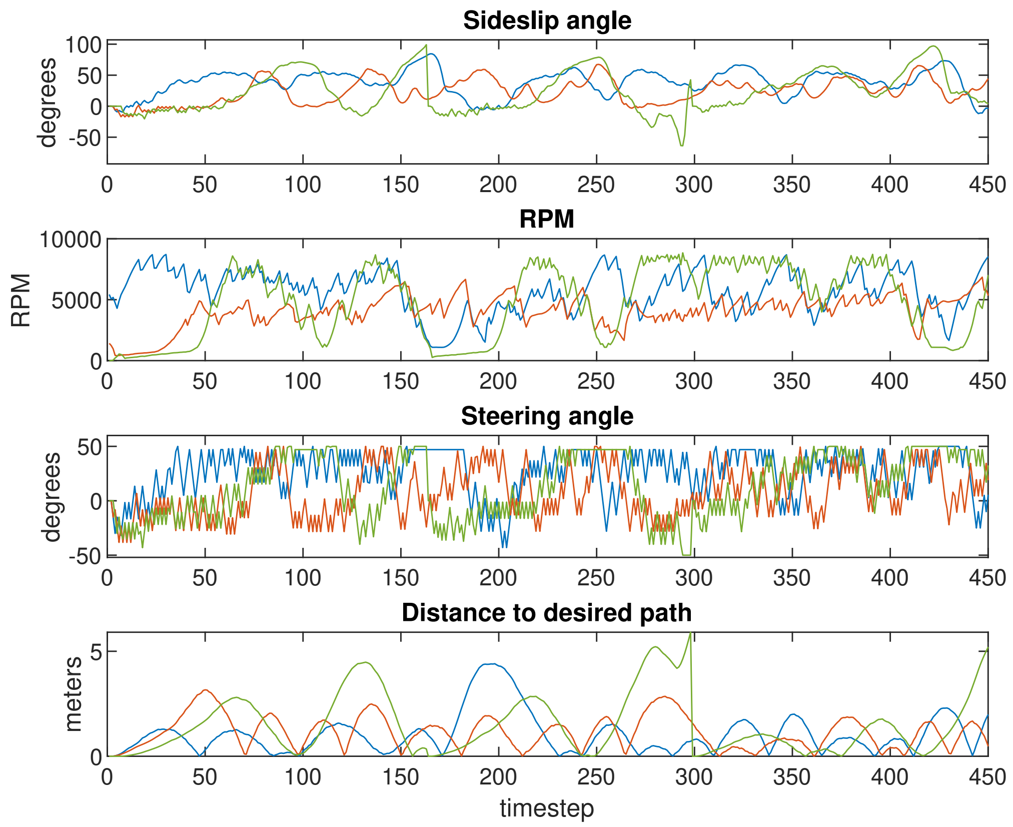
Name	Maximum steering angle	Friction coefficient	Neural net. size	Training	Training path shape
fric-low	50°	$\mu_{\text{adh}} = \mathbf{0.47}$ $\mu_{\text{peak}} = \mathbf{0.6}$ $\mu_{\text{lim}} = \mathbf{0.4}$	3 × 128	copy of: fric-med (no further training)	10-20m radius circle
fric-med	50°	$\mu_{\text{adh}} = \mathbf{0.67}$ $\mu_{\text{peak}} = \mathbf{0.8}$ $\mu_{\text{lim}} = \mathbf{0.6}$	3 × 128	pretrained with: baseline; + further training	10-20m radius circle
fric-high	50°	$\mu_{\text{adh}} = \mathbf{0.87}$ $\mu_{\text{peak}} = \mathbf{1.0}$ $\mu_{\text{lim}} = \mathbf{0.8}$	3 × 128	copy of: fric-med (no further training)	10-20m radius circle

The reference run at $\mu_{\text{lim}} = 0.8$ (fric-med) shows similar behavior to that in section 4.4, holding a sideslip angle of around 50°. Its steering angle is almost always at the maximum of 50°, fully countersteering. Raising the friction to $\mu_{\text{lim}} = 1.0$ (fric-high), along with μ_{adh} and μ_{peak} by the same amount, makes it harder for the controller to hold any sideslip at all, frequently dropping down to 0°. RPM are kept relatively low at only around 4000. As the sideslip drops to 0°, the steering angle falls below 0°, meaning the vehicle does not countersteer and actually drives around the circle without sliding. At $\mu_{\text{lim}} = 0.6$ (fric-low) the controller is barely able to maintain control over the vehicle, even spinning out and having to be reset at timestep 160 and 300. It can be observed to achieve a large sideslip angle of 60° at timestep 100, which it realizes it cannot control and therefore decelerates. While doing so the sideslip drops below 0°, meaning the car's back breaks out in the opposite direction. After successfully regaining control, the controller accelerates back up to max. RPM at timestep 140, though this time gaining too much sideslip and spinning out at timestep 160. Its steering angle also resembles these correctional maneuvers, as it fluctuates between steering left and right.

While not performing as well as under the conditions it was trained on, the deviations in friction are handled relatively well. The vehicle performs correctional maneuvers and tries to stabilize itself. It also adapts its actions, like steering, to the new scenario.



(a) Desired path and the actual measured vehicle position for the $\mu_{\text{lim}} = 0.8$ (blue), $\mu_{\text{lim}} = 1.0$ (orange) and $\mu_{\text{lim}} = 0.6$ (green) run.



(b) The run's sideslip angle, RPM, steering angle and distance to desired path.

Figure 4.5: 15m radius circle drifting performance of a single controller under different friction conditions.

Neural network size

Empirical experiments show that the performance of the 3 layer with 128 neuron neural network on more complicated paths was not as good as on the simple circular trajectories. These controllers are able to learn to follow the desired path, but struggle to hold any angle while doing so. It is therefore necessary to increase the network size.

Name	Maximum steering angle	Friction coefficient	Neural net. size	Training	Training path shape
2×64	50°	$\mu_{\text{adh}} = 0.67$ $\mu_{\text{peak}} = 0.8$ $\mu_{\text{lim}} = 0.6$	**2×64**	pretrained with: baseline + further training	figure-8
3×128	50°	$\mu_{\text{adh}} = 0.67$ $\mu_{\text{peak}} = 0.8$ $\mu_{\text{lim}} = 0.6$	**3×128**	pretrained with: baseline + further training	figure-8
3×256	50°	$\mu_{\text{adh}} = 0.67$ $\mu_{\text{peak}} = 0.8$ $\mu_{\text{lim}} = 0.6$	**3×256**	pretrained with: baseline + further training	figure-8

Figure 4.6 shows 3 different network size controllers being trained to drift a figure-8 shaped path. The neural network with only 2 layers and 64 neurons each can be seen to only achieve a maximum mean reward of 8 before it does not increase any further. During training, the vehicle often spun out of control and generally did not achieve significant amounts of sideslip angle. It only really managed to learn to follow the desired path. The network with 3 layers and 128 neurons already performs better, reaching a maximum mean reward of 12 and. It performed similar to the circular path controllers during training, but struggled with the change of directions on the figure-8, regularly driving off the track. Further increasing network size to 3 layers with 256 neurons yields even better training results, reaching a mean reward of up to 15. This controller's performance is sufficient to drift the figure-8 with a steady sideslip angle and no loss of control, as can be seen in the following sections.

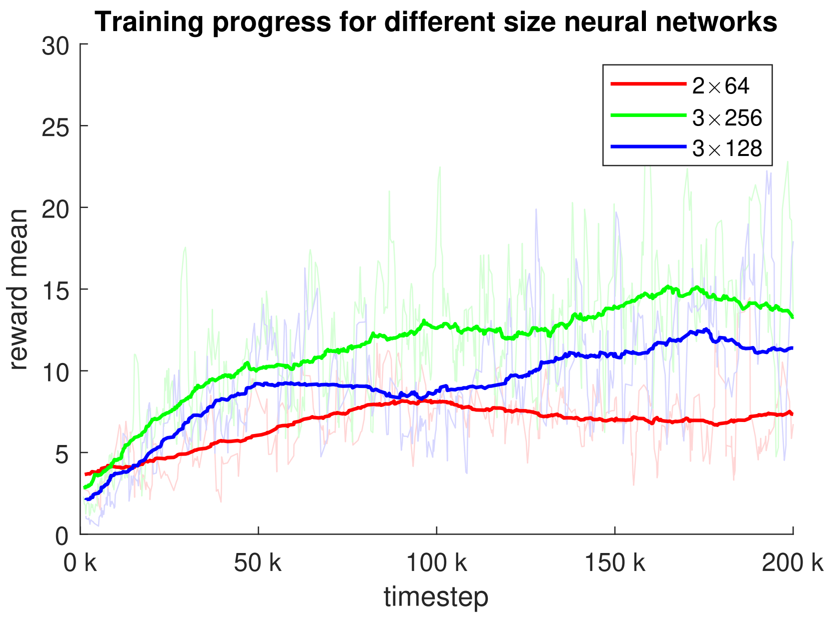


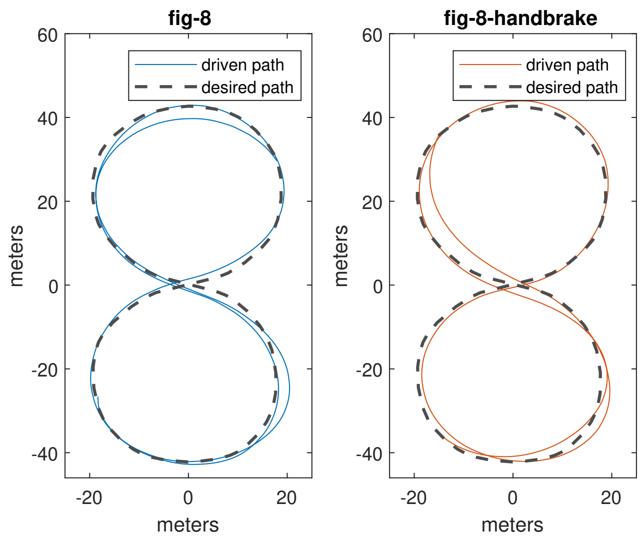
Figure 4.6: Training progress for 3 different neural network sizes (layers × neurons): 2×64, 3×128 and 3×256.

Figure-8

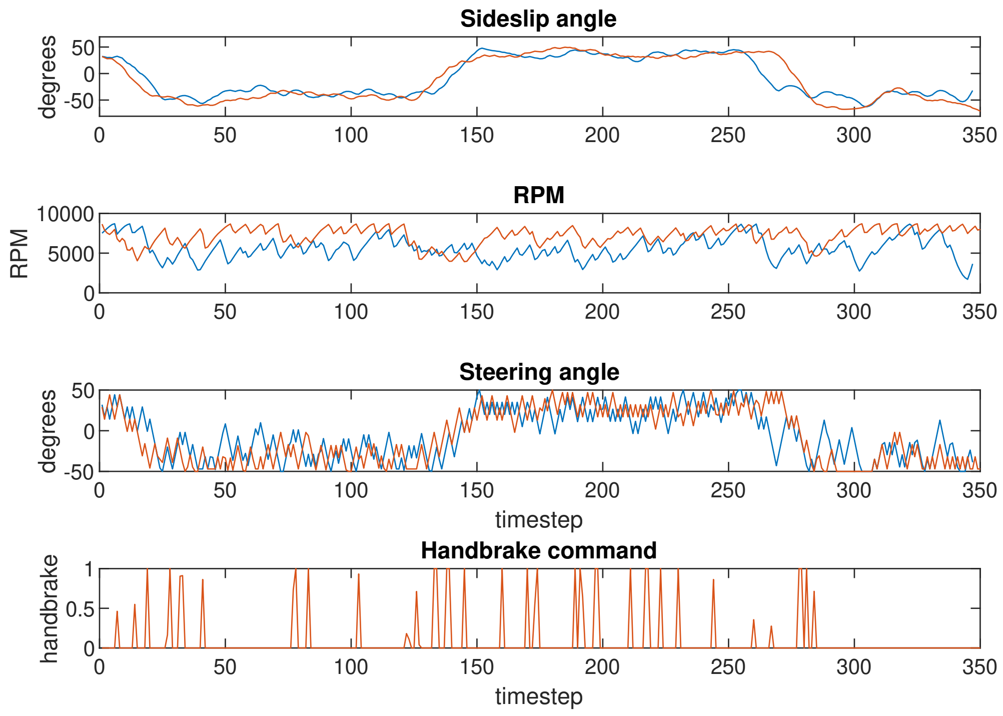
Doubling the amount of neurons per layer of the neural network from 128 to 256 allows the controller to successfully learn to drift a figure-8 shaped trajectory. It was also pretrained with the baseline controller beforehand, enabling a more time efficient learning as it already knows what the objective is.

Name	Maximum steering angle	Friction coefficient	Neural net. size	Training	Training path shape
fig-8 & fig-8-handbrake	50°	$\mu_{\text{adh}} = 0.67$ $\mu_{\text{peak}} = 0.8$ $\mu_{\text{lim}} = 0.6$	**3×256**	pretrained with: baseline; + further training	figure-8

When being reset the vehicle starts at a random point on the path. It is also randomly selected if the track is driven clockwise or counterclockwise. Figure 4.7 compares two controllers drifting the figure-8: with and without the use of the handbrake. There is no significant difference between the two's performance, other than the one using the handbrake maintaining a slightly higher RPM. The handbrake is only tapped every couple of seconds, having next to no impact. Both controllers can be seen to hold a steady sideslip angle of $\pm 50°$, quickly and fluently switching in between. RPM barely drop during the change of directions. This indicates that the car does not decelerate at this point of the track, but smoothly transitions into the other direction. The steering angles' signs are kept equal to those of the sideslip angle, meaning the cars are fully countersteering the entire time. They also stay very close to the desired path, while quickly getting back on track when deviating too much.



(a) Desired path and the actual measured vehicle position for the non-handbrake (blue) and the handbrake (orange) controller.



(b) The controllers' sideslip angle, RPM, steering angle and handbrake command.

Figure 4.7: Figure-8 drifting performance of two controllers, one using the handbrake and one not using it.

Arbitrary paths and transfer learning

Figure 4.9 showcases the fig-8-handbrake controller's performance on a track it has never seen during its training, while also showing this same controller after retraining, where it was able to adapt to this new track. Retraining is done by copying the weights and biases of a controller and initializing the new one's with these before the regular training progress begins.

Name	Maximum steering angle	Friction coefficient	Neural net. size	Training	Training path shape
arb-path	50°	$\mu_{\text{adh}} = 0.67$ $\mu_{\text{peak}} = 0.8$ $\mu_{\text{lim}} = 0.6$	3 × 256	**copy of: fig-8-handbrake; + further training**	arbitrary path

Even without any further training, the fig-8-handbrake controller performs reasonably well - achieving a good amount of sideslip angle, staying close to the desired path and not loosing control over the vehicle. Though it can be observed to drive rather carefully, maintaining a lower RPM and the sideslip frequently dropping to 0°. It also only uses the handbrake somewhat randomly, tapping it slightly every so often. After copying the existing controller and allowing it to further train on the new track, as seen in Figure 4.8, it quickly learns to adapt to the new environment. After this retraining, it is able to drive at higher RPM, while gaining better sideslip angles, which can especially be seen in corners *B*, *C* and *D*. Its overall motion is also smoother, hardly ever dropping the sideslip to 0°, i.e., staying in the drifting state. Handbrake use can now be seen to appear in clusters, actually having an impact on the RPM and correlating with areas where the sideslip's sign switches, like at timesteps 140, 220, 300, 360, 450, 550 and 700. The sign of the steering angle mostly matches the sideslip angle sign for both controllers, indicating the desired countersteering behavior.

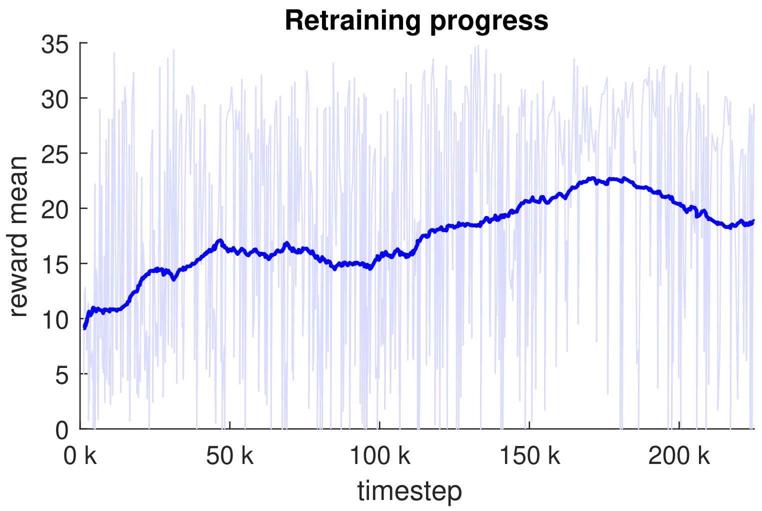
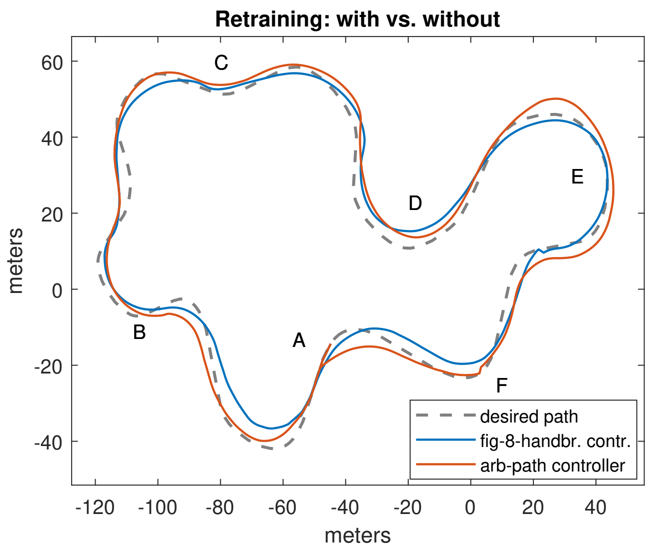
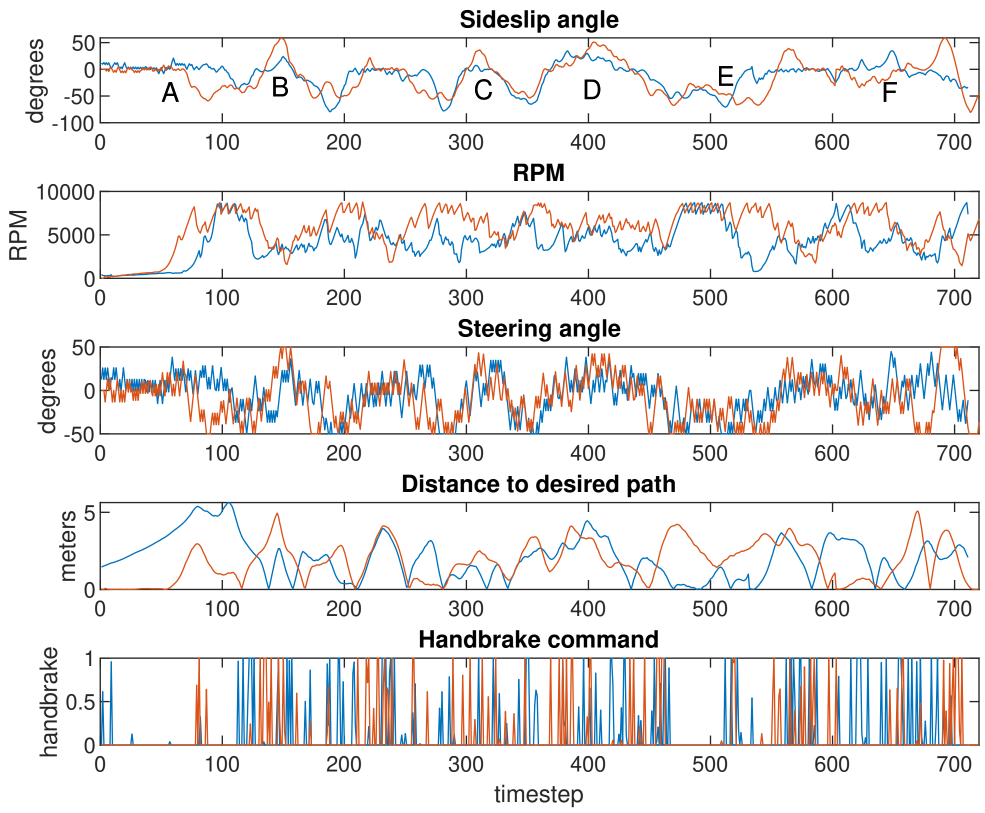


Figure 4.8: Training progress of the figure-8 controller adapting to the arbitrary path.



(a) Desired path and the actual measured vehicle position for the original (blue) and the retrained (orange) fig-8-handbrake controller.



(b) The controllers' sideslip angle, RPM, steering angle, distance to desired path and handbrake command.

Figure 4.9: Arbitrary path drifting performance of the fig-8-handbrake controller before and after adapting to the arbitrary path.

4.2 RC car

To reduce training effort, the simulated car's settings were adapted to resemble the RC car as close as possible and a controller was trained in this modified simulation before being transferred to hardware. It was given another 10k timesteps of training on the real car, adapting to its new environment.

Name	Maximum steering angle	Friction coefficient	Neural net. size	Training	Training path shape
rc-sim	20°	$\mu_{\text{adh}} = 0.6$ $\mu_{\text{peak}} = 0.7$ $\mu_{\text{lim}} = 0.65$	3×128	pretrained with: baseline; + further training	5-15m radius circle (1× to 3× car length)
rc-real	20°	wheels wrapped in packaging tape; driving on rubber mats	3×128	copy of: rc-sim; + further training	1m radius (2.5× car length)

The rc-sim's initial performance on the RC car can be seen in Figure 4.11. The vehicle maintains full throttle and steering to the right for most of the time, resulting in it spinning in small circles around itself. The controller is not able to accurately control the car's position and it eventually circles off the track. After some further 20 minutes of (re-)training the initial controller on the actual RC car, the rc-real controller is able to hold a steady circular motion. It can now be observed to use countersteering more frequently to maintain its position. Because steering is almost instantaneous, the steering command also resembles the actual steering angle. On top of that, it learned to precisely control its position while still moving in a circle. This can be observed when the car misses the orthogonal line of a waypoint: it slowly circles back towards the waypoint, passes it and only then moves on to the next. In addition to that, the controller learned to stay as close to the center as possible, as this is the quickest way of driving through the orthogonal lines.

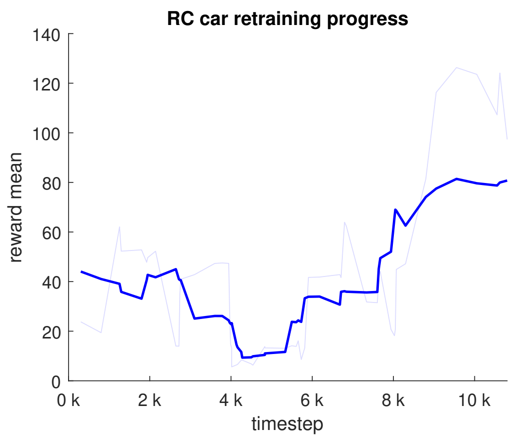
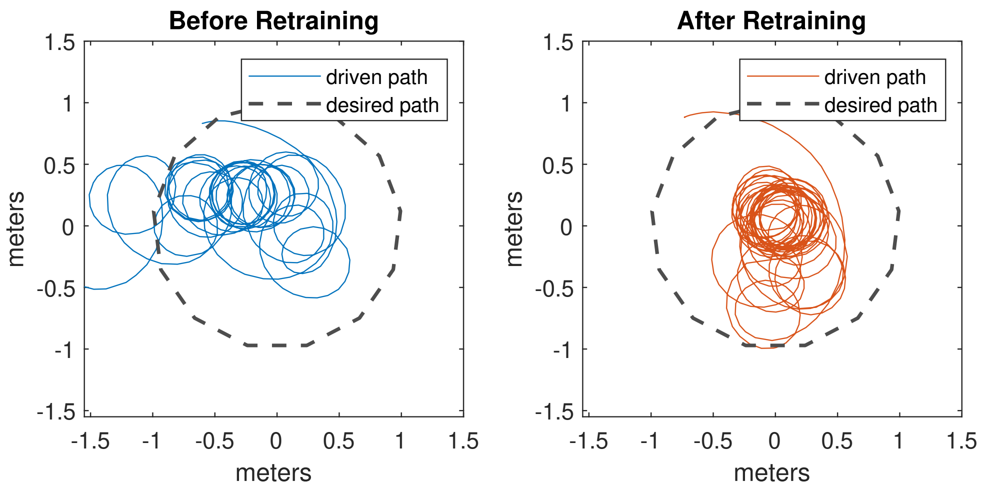
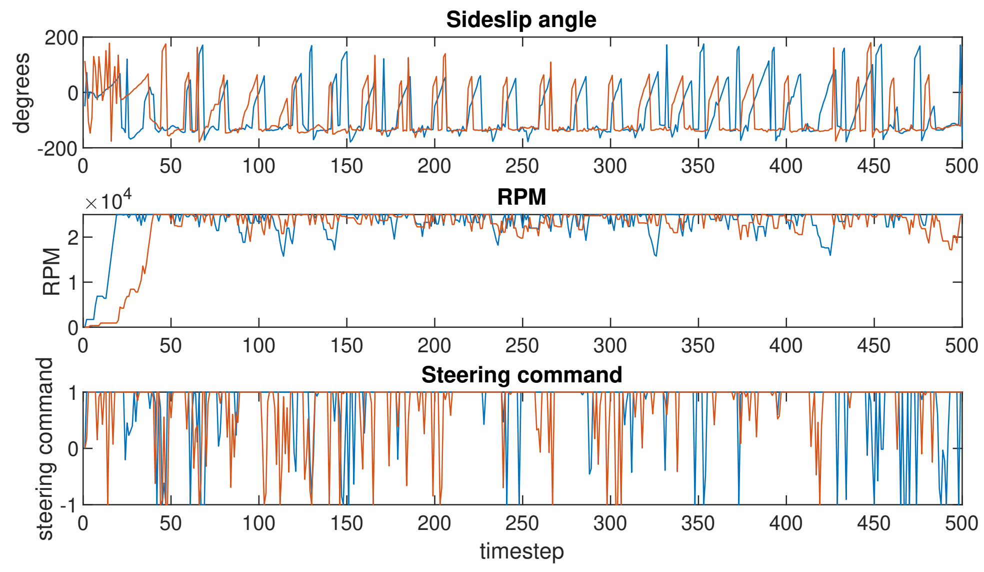


Figure 4.10: Training progress of the simulation trained controller adapting to the laboratory environment.

Both the simulation trained and retrained variant show sudden jumps in their sideslip angle measurement when driving in the laboratory, this is likely due to a measuring error as they appear in regular intervals and the car did not behave like this when observed or in the simulation beforehand.



(a) Desired path and the actual measured vehicle position for the initial (blue) and the retrained (orange) controller.



(b) The controllers' sideslip angle, RPM and steering angle. Because of almost instantaneous reaction, the steering angle and steering command are identical.

Figure 4.11: 1m circle drifting performance of the simulation trained controller in the laboratory, before and after further (re-)training.

5 Discussion

This chapter discusses the above results and proposes how certain aspects of the controller design could be improved. It first suggests changes to the action space. Secondly, different features of the reward function are investigated and new ones offered. Then, the use of braking is discussed. Size and structure of the neural network are also looked at. Finally, the RC car's performance is reviewed and possible improvements introduced.

5.1 Continuous vs. discrete action spaces

As seen in Figure 4.2, the controller almost always exclusively uses the extreme values of its action space. This might be due the human pretraining it received, as here the car was controlled using a keyboard, which only allows the throttle key to be pressed or not. However, even the controllers that were trained from scratch show this exact behavior. A more probable reason is that giving full throttle usually means accelerating quicker, which is very effective in most cases. Though for steering this is not the case, but the behavior is the same - a certain angle is held by alternating between full left and full right, rather than outputting the exact value. Since this behavior occurs anyways, the action space could be made discrete, very likely reducing learning complexity and time. Another solution is forcing smoother outputs by only allowing small steps in the change of throttle and steering commands, e.g., ± 0.1 per timestep. Or alternatively punishing these rapid command changes in the reward function.

5.2 Reward function

Angle maximization component

Since the angle maximization component of the reward function is just linearly rewarding more sideslip angle, the controller can be observed to fall into an oscillating pattern where it pushes the car to the limit of what it can control, then having to slow down and regain control before accelerating back up. This behavior could likely be avoided by introducing an actual target sideslip angle, by changing the angle maximization component of the reward function to a gaussian. This however poses the issue of finding what that target angle should be, as it depends on the curvature of the current road segment and the car's maximum steering angle. While it would be possible to compute the ideal sideslip for a given curvature and vehicle geometry by hand, it does not make sense to do so here, as this is exactly what the computer is supposed to learn on its own. A possible solution to the problem of the oscillating sideslip angle is to take its rate of change, e.g., its time derivative, into consideration when computing the reward.

Path following component

Figure 4.4 shows one of the controllers having developed the behavior of drifting on the inside of the circle, allowing it to pass through the waypoints orthogonal lines quicker than it would driving on the outside. This can also be observed on the RC car. Because of this, the reward function already rewards staying close to the center of the desired path by using (3.5). It might be necessary to further decrease rewards for deviating from the desired trajectory. Or offsetting the peak of the gaussian to not be in the center, such that driving on the outside or inside of a turn is favored. An entirely different approach could be to give a constant reward, no matter where the car passes the waypoint's orthogonal line. This way, the controller could find its own optimum and maybe actually increase the smoothness and elegance of the drift. For circlular paths, where the OLs intersect in the center of the circle, this might not be useful, but for arbitrary racetracks and roads this could be beneficial. Generally, it might be a good idea to allow the controller to find its own optimal cornering technique when the actual optimum is not known. An example for this could be roads that are not meant for racing or simply the lack of data or professional knowledge. On the other hand, racetracks that are build for motorsport already have a well known racing line and professional drivers know how to drive the different corners. In such a setting, this knowledge could be used to shape the path following component of the reward function accordingly. Since the road observations the car makes are already limited to the next few waypoints, basically driving on-sight like a human would, it would also be possible to use visual observations such as a camera image. This could replace the use of waypoints entirely, potentially being used in situations where there is no waypoint data available or it having to be generated while driving.

5.3 Brake usage

As seen in Figure 4.7, the controller does at first not learn to sensibly use the handbrake for the figure-8. However after being transferred to a more complicated track and adapting to it, there is a noticeable correlation between certain road segments and handbrake use. This could be due to the fact that there is no need to brake in a figure-8 setting, while there is for more complicated trajectories. This was also pointed out during the interview in section 2.8, referring to deceleration zones along the track, which there are none of for a simple figure-8. Since the above controllers were not made to drive, and therefore did not perform well, on racetracks that do have deceleration zones, e.g., long straight parts leading into a turn, the reward function would have to be adapted to be able to traverse such paths. This could be done through adding a reward for braking in deceleration zones. This would however require knowledge of where those zones are and also limit the neural network in coming up with its own unbiased solutions. In any case, adding more complexity and potentially even more possible actions to take (e.g., footbrake, clutch, etc.) will inevitably result in more training time and computational effort, but is generally possible.

5.4 Neural network structure

Using the Stable Baseline's default neural network size only allowed the car to learn to drift a circle. Stepping up to slightly bigger networks allows for more diverse tasks to be learned reliably. This architecture was then also able to traverse a more complicated track and even learn to correctly use the handbrake. Other parameters of the learning process, such as learning rate, activation function or PPO2-specific settings were largely left at the provided defaults. Adapting these could yield an even more effective learning progress. A more time efficient learning progress could be achieved by not training in real time, speeding the simulation up and possibly even train multiple instances at once. This however would require appropriate hardware. Given these results and the recent accomplishments in Deep Learning, it is likely that further increasing the size and optimizing parameters of the neural network will also result in increased performance and the ability to learn even more complex behaviors.

5.5 RC car

Even though transferring a controller from simulation to hardware worked well, drastically reducing training time in the laboratory, the RC car's overall drifting performance was not as good. There are multiple reasons for this. First of all it might simply not be possible with the way the car is setup. One limiting factor is its low maximum steering angle of only around 20 degrees. Driving on a rubber floor with offroad tires that are wrapped with packaging tape is another. Even trying to drift by hand using a gamepad proved to be very difficult. Overall, this particular vehicle and this particular environment are not made for drifting, there are however special RC cars that are made for this, which could be used in future projects. It is also possible that the simulated car's settings do not replicate the real RC car well enough. There is a variety of characteristics and parameters that make up vehicle behavior, most of which require more in-depth knowledge of the underlying vehicle dynamics. On top of that, there will always be some discrepancy between simulations and the real world. Generally, adapting the reward function could potentially also improve the RC car's performance, for example by punishing the observed behavior of spinning in circles in one spot.

6 Conclusion

Using Reinforcement Learning to develop autonomous drifting controllers works almost surprisingly well. The simulated car is able to steadily drift different sized circles and a figure-8. It is also able to transfer the learned behavior to new scenarios and drive arbitrary racetracks. The trained controllers robustly handle changes such as different friction, limited steering and modified trajectories. All of this is achieved using a small neural network, a few basic observations like position and velocity and the use of throttle and steering commands. Training time was at most 18 hours in these experiments, though this could be dropped significantly by pretraining with recorded example driving. Especially the reward function is kept simple, basically giving more reward for more sideslip angle, which offers little to no local optima to get stuck in. The overall complexity is therefore rather low and there is much room for expansion. Providing more measurements and allowing the car to use its brakes, paired with a bigger neural network and more training time will almost certainly result in the ability to learn more complex behaviors and achieve even better drifting performance. Transferring trained controllers between simulation and the RC car reduced training time in the field substantially. However, drifting the RC car was not as elegant as in simulation due to the vehicle and its environment not being made for drifting. Generally, these results indicate that it is possible to successfully develop autonomous vehicle controllers using learning based methods. With further improvement and testing, controllers like this could be used as driver assistance systems or even compete autonomously in motorsport competitions.

Further research might focus on expanding the neural network and comparing against other learning algorithms. Deploying the proposed drifting controller to more complex racetracks, possibly expanding to 3D waypoints to handle inclines, and validating its performance under different circumstances could also be insightful. Drifting along other vehicles, e.g., tandem-drifting, would introduce another important part of autonomous driving to the controller: keeping a distance to other road users. Techniques like this can generally be used to research non-optimal or difficult to control driving situations, such as slippery roads or other scenarios where human drivers largely fail. Included with this are cases where vital parts of the vehicle break mid-drive, for example a flat tire on the highway or a cut brake line. High precision stunt driving, like driving a car on 2 wheels, is another potential use-case.

Bibliography

[1] Arthur Juliani, V.-P. B. et al. *Unity: A General Platform for Intelligent Agents*. 2020. eprint: arXiv:1809.02627.

[2] Cerny, A. *Physical driving behavior of vehicles on different ground surfaces and the implementation in Unity3D*. 2015. URL: http://www.alexander-cerny.at/portfolio/wp-content/uploads/2015/12/BA1_CERNY_Alexander.pdf (visited on 08/28/2020).

[3] Craft, B. *KameTrick*. https://www.youtube.com/user/TheKameTrick.

[4] Dhariwal, P., Hesse, et al. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[5] *Dot Products and Projections*. 1996. URL: http://sites.science.oregonstate.edu/math/home/programs/undergrad/CalculusQuestStudyGuides/vcalc/dotprod/dotprod.html (visited on 08/28/2020).

[6] Efstathios Velenis, P. T. et al. Dynamic Tire Friction Models for Combined Longitudinal and Lateral Vehicle Motion. In: *Vehicle System Dynamics, Taylor & Francis* 43 (1):3–29, 2005. URL: https://hal.inria.fr/inria-00000921/document (visited on 08/28/2020).

[7] Gentle, J. E. *Matrix Algebra: Theory, Computations, and Applications in Statistics*. Ninth Edition. Springer, 2007, p. 182. URL: https://books.google.de/books?id=PDjIV0iWa2cC\&pg=PA182\&redir_esc=y#v=onepage\&q\&f=false (visited on 08/28/2020).

[8] Greg Brockman, V. C. et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[9] Haas, J. *A History of the Unity Game Engine*. 2016. URL: https://web.wpi.edu/Pubs/E-project/Available/E-project-030614-143124/unrestricted/Haas_IQP_Final.pdf (visited on 08/28/2020).

[10] Hill, A., Raffin, et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2018.

[11] Hindiyeh, R. Y. *Dynamics and control of drifting in automobiles*. 2013. URL: https://purl.stanford.edu/vz162hz7668 (visited on 08/28/2020).

[12] (https://math.stackexchange.com/users/55608/shard), S. *Calculate on which side of a straight line is a given point located?* Mathematics Stack Exchange. URL:https://-math.stackexchange.com/q/274728 (version: 2020-06-12). eprint: https://math.stackexchange.com/q/274728. URL: https://math.stackexchange.com/q/274728 (visited on 08/28/2020).

[13] Jason Kong, M. P. et al. *Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design*. 2015. URL: https://borrelli.me.berkeley.edu/pdfpub/IV_KinematicMPC_jason.pdf (visited on 08/28/2020).

[14] John Schulman, F. W. et al. *Proximal Policy Optimization Algorithms*. 2017. eprint: arXiv:1707.06347.

[15] Jonathan Y. Goh, T. G. and Gerdes, J. C. *A Controller for Automated Drifting Along Complex Trajectories*. 2018. URL: https://ddl.stanford.edu/sites/g/files/sbiybj9456/f/marty_avec2018_fullpaper.pdf (visited on 08/28/2020).

[16] Layton, J. How Drifting Works. In: 2006. URL: https://auto.howstuffworks.com/auto-racing/motorsports/drifting4.htm (visited on 08/28/2020).

[17] Mark Cutler, J. P. H. *Autonomous Drifting using Simulation-Aided Reinforcement Learning*. 2016. URL: https://weeklyrobotics.com/publications/autonomous_drifting/Cutler16_ICRA_final_submission.pdf (visited on 08/28/2020).

[18] Matthew O'Kelly, V. S. et al. F1/10: An Open-Source Autonomous Cyber-Physical Platform. In: 2019. eprint: arXiv:1901.08567.

[19] OpenAI, M. A. et al. *Learning Dexterous In-Hand Manipulation*. 2019. eprint: arXiv:1808.00177.

[20] P. Cai, X. M. et al. *High-speed Autonomous Drifting with Deep Reinforcement Learning*. 2020. URL: https://arxiv.org/pdf/2001.01377.pdf (visited on 08/28/2020).

[21] S. Bhattacharjee, K. D. K. and Jain, R. *Autonomous Drifting RC Car with Reinforcement Learning*. 2018. URL: https://i.cs.hku.hk/fyp/2017/fyp17014/docs/Final_report.pdf (visited on 08/28/2020).

[22] Silver D. Huang A., M. C. et al. Mastering the game of Go with deep neural networks and tree search. In: *Nature* 529:484–489, 2016. DOI: https://doi.org/10.1038/nature16961.

[23] Smart, B. *Reinforcement Learning: A User's Guide*. 2005. URL: http://www2.econ.iastate.edu/tesfatsi/RLUsersGuide.ICAC2005.pdf (visited on 09/07/2020).

[24] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Kinetic. URL: https://www.ros.org.

[25] Sutton, R. and Barto, A. *Reinforcement Learning: An Introduction*. The MIT Press; Cambridge, Massachusetts; London, England, 2015. URL: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf (visited on 09/07/2020).