First Order Driving Simulator

Wesley Hsieh

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-102
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-102.html

May 12, 2017

First Order Driving Simulator

by

Wesley Hsieh

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ken Goldberg, Chair
Professor Trevor Darrell

Spring 2017

The dissertation of Wesley Hsieh, titled First Order Driving Simulator, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

First Order Driving Simulator

Abstract

First Order Driving Simulator

by

Wesley Hsieh

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Ken Goldberg, Chair

Autonomous driving is a complex task that features high variability in everyday driving conditions due to the presence of vehicles and different road conditions. Many data-centric learning models require exposure to a high number of data points collected in various driving conditions to be robust to this variability. We present the First Order Driving Simulator (FODS), an open-source lightweight driving simulator designed for data collection and benchmarking performance for autonomous driving experiments, with a focus on customizability and speed. The car model is controlled using steering, acceleration, and braking as inputs. The car features the choice between kinematic and dynamic bicycle models with slip and friction, as a first-order approximation to the dynamics of a real car. Users can customize features including the track, vehicle placement, and other initial conditions of the environment, as well as environment interface features such as the state space (images, positions and poses of cars) and action space (discrete or continuous controls, limits). We benchmark our performance against other simulators of varying degrees of complexity, and show that our simulator matches or outperforms their speeds of data collection. We also feature parallelization with Ray [36], a distributed execution framework aimed at making it easy to parallelize existing codebases, which allows for significant speed increases in data collection. Finally, we also perform experiments analyzing the performance of various imitation learning and reinforcement learning algorithms on our simulator environment.

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank my research advisor Professor Ken Goldberg for the opportunity to work in the AUTOLAB. Working in an academic research environment has made the past few years of my undergraduate and graduate career an incredible experience. Professor Ken Goldberg has provided many great ideas and insights on both the direction and presentation of the work I have done throughout my stay here.

I especially thank my mentor Michael Laskey for supporting me throughout my research experience here, from whom I have learned so much from collaborating with him over the past few years. He provided an endless stream of advice on everything from research questions to future careers. I am incredibly grateful for the opportunity to work with him, and his advice has been a major key to the work I have done here.

I would also like to thank the other members of the AUTOLAB for contributing to a great research environment. I am incredibly humbled by the amount of work that is put into all of the various projects there, as well as the availability and willingness of its members to help each other.

I also thank Professor Trevor Darrell for his help on my master's thesis committee, as well as everything I have learned from his computer vision class last year.

Finally, I would like to thank my family for their support throughout the last four years of my undergraduate and graduate career at the University of California, Berkeley.

Abstract

First Order Driving Simulator

by

Wesley Hsieh

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Ken Goldberg, Chair

Autonomous driving is a complex task that features high variability in everyday driving conditions due to the presence of vehicles and different road conditions. Many data-centric learning models require exposure to a high number of data points collected in various driving conditions to be robust to this variability. We present the First Order Driving Simulator (FODS), an open-source lightweight driving simulator designed for data collection and benchmarking performance for autonomous driving experiments, with a focus on customizability and speed. The car model is controlled using steering, acceleration, and braking as inputs. The car features the choice between kinematic and dynamic bicycle models with slip and friction, as a first-order approximation to the dynamics of a real car. Users can customize features including the track, vehicle placement, and other initial conditions of the environment, as well as environment interface features such as the state space (images, positions and poses of cars) and action space (discrete or continuous controls, limits). We benchmark our performance against other simulators of varying degrees of complexity, and show that our simulator matches or outperforms their speeds of data collection. We also feature parallelization with Ray [36], a distributed execution framework aimed at making it easy to parallelize existing codebases, which allows for significant speed increases in data collection. Finally, we also perform experiments analyzing the performance of various imitation learning and reinforcement learning algorithms on our simulator environment.

Chapter 1

Introduction

Autonomous driving has increasingly become a popular area of research especially in the past few years. Many major companies including Google, Tesla, NVIDIA, and Uber have already dedicated teams of researchers into development of self driving cars. Smaller companies and startups built around autonomous driving and related products have also sprung up recently.

Autonomous driving requires precision in perception and robustness to varying conditions of the environment. Due to the high-speed nature of driving, where critical decisions are often required within seconds, controllers for autonomous cars must also be able to plan and execute in real-time. The environment itself is partially observed; sensor observations of the environment are subject to noise and occlusion from other objects in the environment [30]. In addition to learning how to navigate the desired path safely, controllers face a large degree of variability in everyday driving conditions due to the presence of pedestrians, vehicles, and other occupants of the road [48], making it necessary to re-plan based on changes in the environment for safe trajectory planning and execution. Additionally, safety is an especially important consideration of designing a self-driving car controller due to the large costs of a collision with another vehicle or object on the road. Driving tasks also have a large time horizon, making frequent planning and re-planning extremely important to handle changes in the observed environment.

There have been many data-centric methods that have been proposed to train controllers for autonomous driving. All of these data-driven methods aim to acquire enough data and experience to become robust to various changes in the environment that could occur during day-to-day driving. Imitation learning is a class of methods that involves training an autonomous agent to imitate a supervisor that is known to be proficient at the task at hand. In the case of autonomous driving, this agent is often a human in the real world or a classical control model that has access to the dynamics of the simulator environment. Examples of imitation learning methods applied to driving environments include DAgger [42], SHIV [29], SafeDAgger [54], and Dart [28]. Reinforcement learning methods have also been proposed and applied to driving environments; reinforcement learning is a class of methods that involves learning from experience, where an agent's actions are given feedback indirectly through a reward signal. The agent aims to balance exploration of new policies with exploitation of the currently known best policy. Examples of reinforcement learning methods applied to driving environments include as Q-Learning [49], Vanilla Policy Gradient [50],

Trust Region Policy Optimization [45], Deep Deterministic Policy Gradient [31], and A3C [34].

Because of the many safety and robustness requirements for autonomous driving, policies must be able to re-plan quickly and adapt to various environments. Training a data-driven model requires a large number of data samples to be robust to the environment due to the high dimensionality of image observations and high variance in the state space [9].

Collecting training data in the real world often features the assistance of human operators for legal requirements and to ensure safety and quality of feedback [4], making data collection costly.

Training in simulation presents an avenue for safely collecting data without the large consequences of making a mistake, which is especially useful for training in the early stages of a model where it is the most prone to crashing. Simulation also allows faster learning of safe driving behaviors in the presence of other vehicles, which can be fine-tuned to adapt to real-world situations [16].

There exist many simulators that are currently being used in autonomous driving experiments. Simulators are used to determine the effectiveness of a new method specifically for autonomous driving and control. Driving simulators have also been used as a general benchmarking environment to evaluate the effectiveness of a new general-purpose algorithms. Many of these experiments require large quantities of training data to perform, making data collection an expensive operation. TORCS, an open-source racing simulator, has been used in many research experiments [34, 10, 37, 33, 32]. Similarly, Grand Theft Auto has also been used as a data source for experiments related to the driving domain [17, 41]. Research groups and labs have also independently developed their own driving simulators of varying complexities for their own experiments.

We present a lightweight driving simulator designed primarily for autonomous driving experiments, with a focus on customizability and speed of integration with existing learning pipelines. We design the simulator as a first-order approximation of a driving environment to be used as a benchmarking tool to quickly evaluate the effectiveness of a new method for autonomous driving, especially with consideration to the safety and data efficiency of the output policy. To simplify the state space of the environment, we feature a 2D birds-eye view of the simulator centered around the car, with simple graphics for performance. The car features the choice between point, kinematic, and dynamic bicycle models with slip and friction, as a first-order approximation to car dynamics. Its input controls include steering, acceleration, and braking, and can be input from an external program or manually through a keyboard or Xbox controller. To facilitate ease of integration with external modules, the simulator implements the OpenAI Gym interface [8], a popular environment interface for learning problems.

Using the configuration module, many features of the simulator itself can be tweaked to fit the desired task. On startup, the environment reads from this configuration file to set up its features including the track, car placement as well as environment features such as state space (images, positions and poses of cars) and action space (discrete or continuous, limits).

We also feature integration with Ray [36], a distributed execution framework aimed at making it easy to parallelize existing codebases, including data collection for machine learning and reinforcement learning applications. Experiments using this framework can significantly improve the rate of data collection, which is often a bottleneck for learning experiments. Using a parallelization

framework like Ray is extremely important for reducing the costs of data collection especially for data-intensive learning methods.

The goal of this simulator is to be able to quickly configure the environment to fit the desired driving task, then quickly integrate with an existing learning model to start gathering data for training. We perform experiments benchmarking performance against other driving simulators. We also provide example code for integration with imitation learning and reinforcement learning algorithms.

Chapter 2

Related Work

2.1 Benchmarking

Our work is related to data-centric end-to-end systems developed for autonomous driving. ALVINN [39] was one of the first systems that leveraged neural networks to train a policy to map from images to controls for driving; a similar approach was performed by Chen et al. using a more convolutional neural network model [13]. Xu et al. created an end-to-end system for learning vehicle motion models from video datasets [52]. Bojarski et al. used supervised learning to train end-to-end convolutional neural networks for driving in the real world [6], as well as provide explanations and insight into what the system learns [7].

Our work is also related to vision benchmarks and tasks that are necessary for autonomous driving. There exist benchmarks in the real world evaluate the ability of an autonomous driving system to perceive important features of the surrounding environment. Dubbelman et al. created a dataset to benchmark stereo based motion estimation [15]. Geiger et al. created a dataset to benchmark various perception tasks including stereo, optical flow, visual odometry, and 3-D object detection [19]. Fritsch et al. created a dataset to benchmark road area and lane detection in urban areas [18].

Driving simulators have also been used to provide synthetic data for various tasks. Richter et al. created a synthetic dataset generated from Grand Theft Auto V to supplement real-world data for training semantic segmentation systems [41]. TORCS [51], an open-source driving simulator, has been used to learn mid-level cues for autonomous driving that generalized to real images [12], and to benchmark imitation learning [10, 54], reinforcement learning [33], and genetic algorithms [37].

2.2 Driving Simulators

There are many existing simulators with varying degrees of complexity and different target audiences. We compare to other simulators used for autonomous driving projects. All of these

Figure 2.1: TORCS

feature similar control schemes, with steering, acceleration, and braking as the primary control inputs to the car.

TORCS [51] is an open-source 3-D racing car simulator that allows implementation of controllers to race against computer opponents. The project is more complex than FODS and includes dynamics such as gears, fuel, damage, wheel velocities as well as other vehicles. Direct communication with the game is possible with Java and C++, while there exists third-party Python interfaces [53] for communicating with the game through a client-server interface. There also exists a third-party Python library that also implements the Gym interface for ease of integration with external learning pipelines. The viewpoint of the simulator is from the driver's point of view. In comparison to FODS, this simulator is more complex and features more customizability.

DeepGTAV [43] is an interface for communicating with an instance of Grand Theft Auto, a popular 3-D open-world sandbox game with a driving component. The game itself is proprietary, and requires purchase of a license to use. It communicates with the game through a client-server interface in Python, which transmits messages in JSON format. The simulator is in 3-D and the viewpoint is from behind the car. The environment includes realistic graphics and can include other cars. In comparison to FODS, this simulator is proprietary and more complex, and the environment itself is designed for gaming purposes rather than for driving experiments.

Udacity features an open-source 3-D driving simulator rendered in Unity for their online course on using deep learning to train an autonomous driving agent. The viewpoint for collecting data is from behind the car, while labeled images are generated from a first-person drivers perspective of the road. The project features two built-in tracks, as well as a custom track creation module. In comparison to FODS, this simulator has less customizability, more complex graphics, and does not feature other cars.

OpenAI Gym [8] features many built-in environments for evaluation of learning algorithms.

Figure 2.2: GTA V

Figure 2.3: Udacity's Driving Simulator

Their driving environment is an open-source 2-D driving simulator implemented with Box2D [11]. The state space is a top-down view of the environment, with the camera centered around the car. Slip and friction dynamics are implemented, and skid marks are rendered when the car slips. In comparison with FODS, this simulator has less customizability and does not feature other cars on the track.

Many research groups and labs have also independently developed their own driving simulators of varying complexities for their own experiments. Sadigh et al. created their own 2-D driving simulator as an environment for evaluating inverse reinforcement learning algorithms with a focus on interactions between different vehicles [44]. The viewpoint is a 2-D birds-eye view centered around the car and its surrounding environment. The dynamics model is specified symbolically

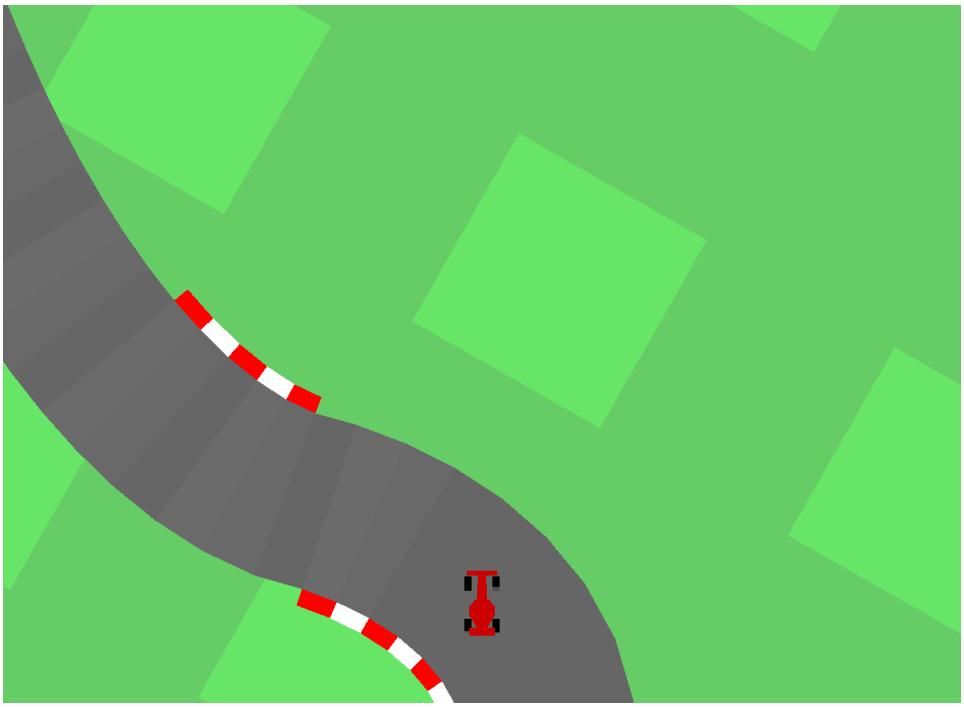

Figure 2.4: OpenAI Gym: CarRacing-v0 Environment

through Theano [5], which facilitates usage of classical control and other algorithms that require knowledge of the dynamics.

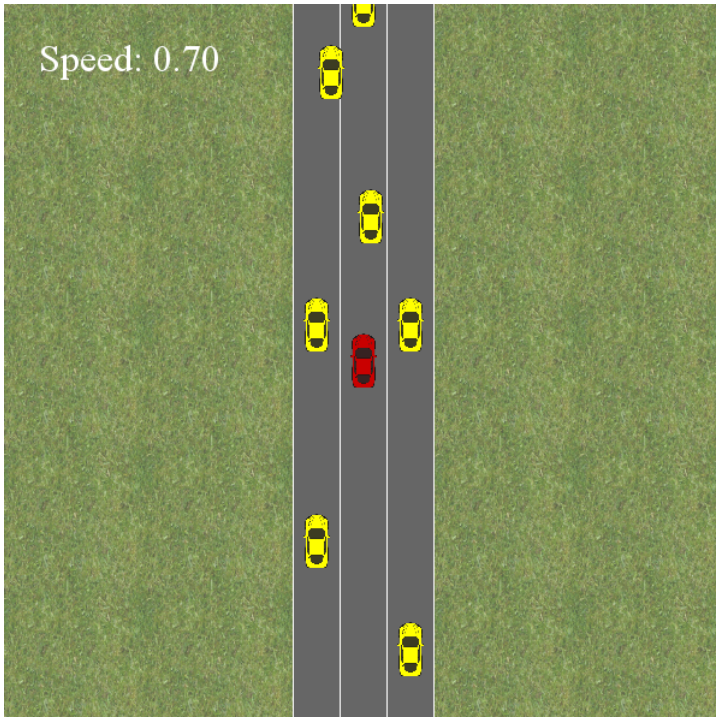

Figure 2.5: Driving Interactions: Simulator

Chapter 3

Dynamics

There are many existing car dynamics models with differing degrees of complexity. We opted to use relatively simpler dynamics for the car to facilitate performance and simplicity while maintaining a first order approximation of a car. The car features the choice between point, kinematic, and dynamic bicycle models with slip and friction, as a first order approximation. Both models use steering and acceleration and braking as input controls. The models are discretized using SciPy's [24] ordinary differential equation integrator and is sampled at $t_d = 100$ ms. For simplicity and simulator performance reasons, we opt not to use complete full vehicle dynamics models [2, 35, 40].

3.1 Point Model

The point model is the most simple dynamics model available for the simulator. The continuous differential equations that describe the point bicycle models are as follows.

$$\dot{x} = v\cos(\delta_f) \tag{3.1}$$

$$\dot{y} = v\sin(\delta_f) \tag{3.2}$$

$$\dot{\psi} = \delta_f \tag{3.3}$$

$$\dot{v} = a \tag{3.4}$$

x and y are the coordinates of the center of mass in an inertial frame (X,Y). ψ is the inertial heading and v is the speed of the vehicle. a is the acceleration of the center of mass in the same direction as the velocity. The control inputs are the front and rear steering angles δ_f, δ_r, and a. Since in most vehicles the rear wheels cannot be steered, we assume $\delta_r = 0$.

3.2 Kinematic Bicycle Model

The inertial position coordinates and heading angle in the kinematic bicycle model are defined in the same manner as those in the point model. The continuous differential equations that describe the kinematic bicycle models are as follows [40, 27].

$$\dot{x} = v\cos(\psi + \beta) \tag{3.5}$$

$$\dot{y} = v\sin(\psi + \beta) \tag{3.6}$$

$$\dot{\psi} = \frac{v}{l_r}\sin(\beta) \tag{3.7}$$

$$\dot{v} = a \tag{3.8}$$

$$\beta = \tan^{-1}(\frac{l_r}{l_f + l_r}\tan(\delta_f)) \tag{3.9}$$

l_f and l_r represent the distance from the center of the mass of the vehicle to the front and rear axles, respectively. β is the angle of the current velocity of the center of mass with respect to the longitudinal axis of the car.

3.3 Dynamic Bicycle Model

The inertial position coordinates and heading angle in the dynamic bicycle model are defined in the same manner as those in the kinematic bicycle model. The continuous differential equations that describe the kinematic bicycle models are as follows.

$$\ddot{x} = \dot{\psi}\dot{y} + a_x \tag{3.10}$$

$$\dot{y} = -\dot{\psi}\dot{x} + \frac{2}{m}(F_{c,f}\cos\delta_f + F_{c,r}) \tag{3.11}$$

$$\ddot{\psi} = \frac{2}{I_z}(l_f F_{c,f} - l_r F_{c,r}) \tag{3.12}$$

$$\dot{X} = \dot{x}\cos\psi - \dot{y}\sin\psi \tag{3.13}$$

$$\dot{Y} = \dot{x}\sin\psi + \dot{y}\cos\psi \tag{3.14}$$

$\dot{x}$ and $\dot{y}$ denote the longitudinal and lateral speeds in the body frame, respectively and $\dot{\psi}$ denotes the yaw rate. m and I_z denote the vehicles mass and yaw inertia, respectively. $F_{c,f}$ and $F_{c,r}$ denote the lateral tire forces at the front and rear wheels, respectively, in coordinate frames aligned with the wheels.

For the linear tire model, $F_{c,i}$ is defined as

$$F_{c,i} = -C_{\alpha_i}\alpha_i \tag{3.15}$$

where $i \in \{f, r\}$, α_i is the tire slip angle and C_{α_i} is the tire cornering stiffness.

Table 3.1: Road Friction Coefficients

Road Type	μ
Dry	0.9
Wet	0.6
Snow	0.2
Ice	0.05

We estimate the cornering stiffness as follows [47].

$$c_f = \mu \cdot m \cdot \frac{l_r}{l_f + l_r} \cdot \frac{\dot{a_f}}{\dot{\delta}_f - \frac{a_f}{v} + r} \tag{3.16}$$

This estimate is restricted to $\dot{\delta}_f - \frac{a_f}{v} + r \neq 0$. Under the assumption of the linearized single track model equations, this term is identical to $\dot{\alpha}_f$. We therefore use the following estimate.

$$c_f = \mu \cdot m \cdot \frac{l_r}{l_f + l_r} \tag{3.17}$$

We use the value of $I_z \approx 2500$, which is a typical value for a smaller car [22].
We also list the different coefficients of friction based on the road type [20].

Chapter 4

System Features

4.1 System Architecture

The project is written in Python with the PyGame library for visualization [46]. The project implements the OpenAI Gym environment interface for ease of integration with learning experiments [8]. Sample code for interaction between a generic environment and agent is as follows. The standardized interface allows easy integration with different learners.

```
# assume agent is intialized
env = DrivingEnv()
done = False
while not done:
    action = agent.get_action(env)
    observation, reward, done, info = env.step(action)
```

4.2 Customization

One of the goals of this project is to provide a highly customizable benchmarking environment for different learning algorithms. Using the command-line based configuration module, many features of the simulator itself can be tweaked to fit the desired task; the configurations are stored in an external file in JSON format. On startup, the environment loads from this configuration file to set up its features including initial conditions and environment interfaces. These files can be saved and loaded from the configuration file for ease of editing later on.

Many of the features of the environment are customizable to fit the needs of the experiment. Learning algorithms are often benchmarked to determine their model's stability and robustness to unseen states; we offer configuration options to provide control over the variance in the encountered state space. One feature is track placement including positions and orientations, which allows testing of the policy's generalization error when navigating previously unseen tracks. The road conditions of the track can be configured to include dry, wet, snowy, and icy terrain to test

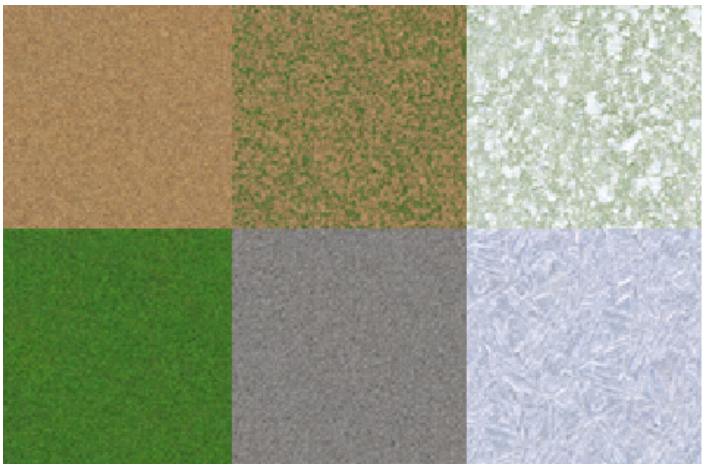

Figure 4.1: Examples of different terrain, with different coefficients of friction.

robustness to different vehicle handlings; similarly, the dynamics model of the main car can be changed between the point, kinematic, and dynamic bicycle models to offer control over the vehicle's handling. The number of computer-controlled cars, as well as their positions, velocities, orientations, and color can be changed to test the robustness of the policy to avoiding cars at varying parts of the map. We also feature control over the randomization of car properties, including positions withing specified bounding boxes, distances between car placements, and starting angles, which allows for control over the large variance in the initial state distribution.

Additionally, many properties of the simulator itself can be customized to fit the needs of the experimental pipeline and system. The simulator can be configured to set the rendering screen size as well as enabling or disabling rendering, which allows for significant performance increases for applications that do not require image data. Similarly, we can configure the output state space, including rendered RGB color images, or positions, velocities, and orientations of cars. The size of the output image can also be downsampled to be smaller than the original rendered image using OpenCV [23]. These options allow the state space to match the data formats needed by the model, as well as allow for performance increases for experiments without rendering. We also feature customization of the action space, including choice of discrete or continuous controls. The control limits of the action space can also be modified, as well as the step size and size of the action space in the discrete case. Steering or acceleration and braking controls can also be disabled or enabled. These features allow control over the action space for the needs of the experiment. The time horizon of the environment can also be set, after which the environment automatically returns True for its "done" output when taking a step, to indicate that the environment should be reset; this allows for performing experiments with varying time lengths. Finally, the sampling rate for logging of the states and actions encountered can be set, as well as its respective output location.

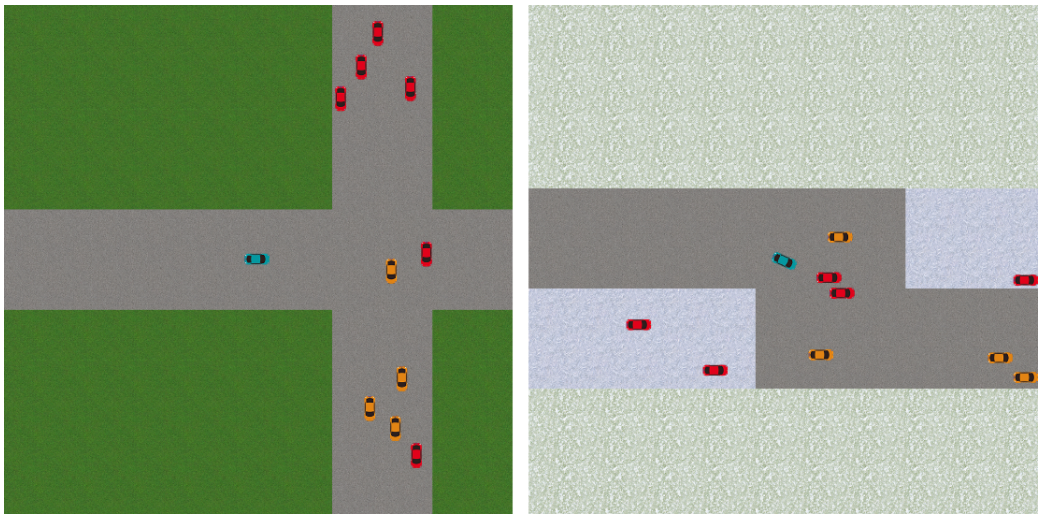

Figure 4.2: Examples of different track configurations.

Adding Features

Additional configuration features can be easily added by modifying the configuration module to add an additional entry and prompt into the existing list of features. Afterwards, the corresponding entry will appear to the environment on startup when it loads the configuration file. The environment then can handle the entry as desired.

Advanced Customization

Additional features that are more complex and intrinsic to the environment, such as custom reward functions, additional car dynamics models, or terrain models, can be implemented by creation of custom classes that inherit from the corresponding environment, car, or terrain class, and overwriting the corresponding functions. Under the hood, most objects that the environment interacts with has its own corresponding ”step” function for updates, which can be modified to handle most changes.

4.3 Parallelization

We feature integration with Ray [36], a distributed execution framework aimed at making it easy to parallelize existing codebases. including data collection for machine learning and reinforcement learning applications. Experiments using this framework can significantly improve the rate of data collection, which is often a bottleneck for learning experiments. We provide an example of integration of Ray with imitation learning experiments, where we iteratively parallelize our data collection over multiple CPUs each with their own individual instance of the simulator, pool our simulated trajectories and rewards from the workers, then update our model with the data and train using a GPU.

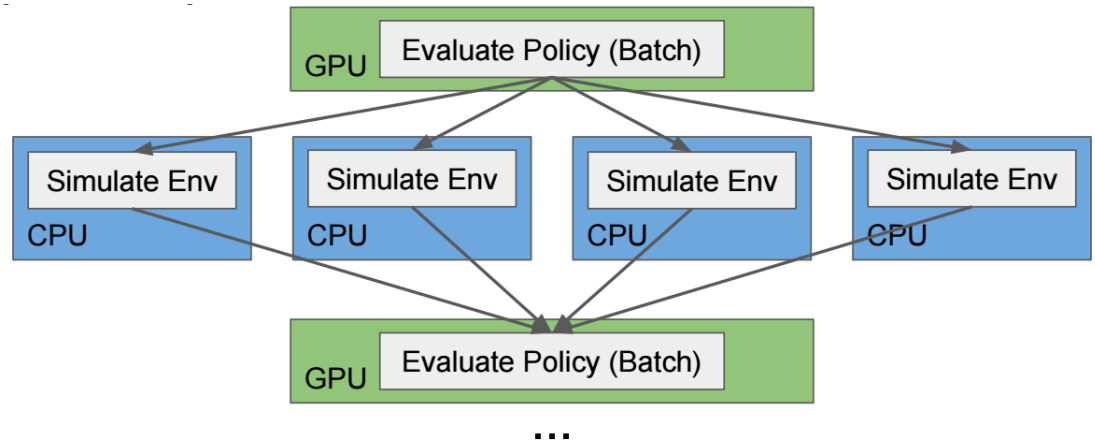

Figure 4.3: Architecture of Ray processing pipeline.

Chapter 5

Benchmarking

5.1 Simulator Comparison

We benchmark our performance against other simulators of varying degrees of complexity, and show that our simulator matches or outperforms their speeds of data collection. All trials are performed on a 12-processor Intel Core i7 CPU. We benchmark by measuring the amount of time the simulator takes to reach 1000 steps over $n = 10$ trials. We also attempt to disable rendering and report the corresponding times; this option has not been found to be possible especially for more complex simulators that couple rendering with the environment update.

5.2 Results

We find that the First Order Driving Simulator outperforms the other simulators that we benchmark against. We significantly outperform the 3-D simulators with over three times the speed of the next best simulator, as expected due to their higher detail and complexity in rendering. We also outperform the 2-D simulators with slightly under two times the speed of the next-best simulator.

Table 5.1: Steps (x1000) per Minute, Rendering

Simulator	mean	stdev
FODS	**3.48**	**0.0372**
Udacity	0.99	0.0623
TORCS	0.29	0.0004
Driving Interactions	0.56	0.0164
OpenAI Gym (CarRacing-v0)	1.84	0.0526

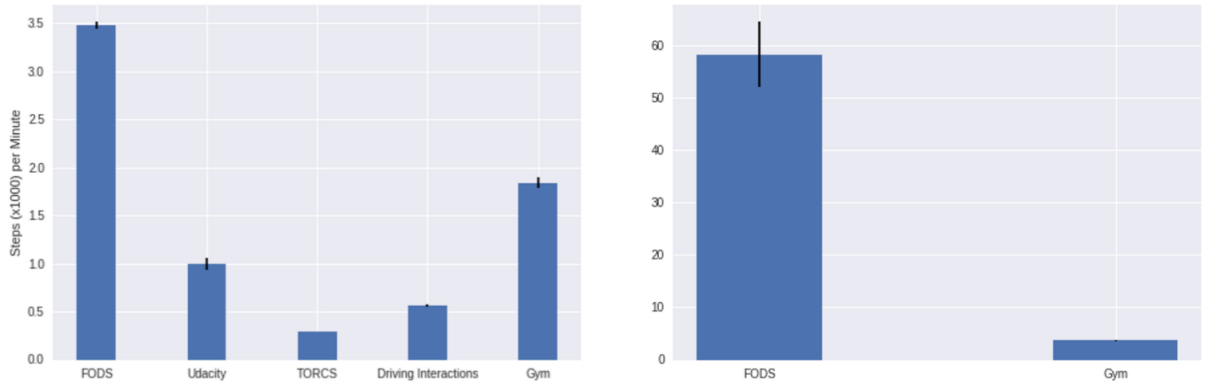

Figure 5.1: Simulator performance benchmarking results. Left image is benchmarking with rendering. Right image is benchmarking with rendering disabled.

Table 5.2: Steps (x1000) per Minute, No Rendering

Simulator	mean	stdev
FODS	**58.3**	**6.321**
Udacity	-	-
TORCS	-	-
Driving Interactions	-	-
OpenAI Gym (CarRacing-v0)	3.71	0.1017

5.3 Parallelization

We also benchmark the effects of parallelization on the performance of FODS. We benchmark by measuring the amount of time the simulator takes to reach 10000 steps over $n = 10$ trials, while varying the number of cores allotted to the simulator. We report results for rendering enabled and disabled.

5.4 Results

We find that increasing the number of processors up to the maximum limit of 12 improves the performance of the simulator, but there are diminishing returns due to additional overhead.

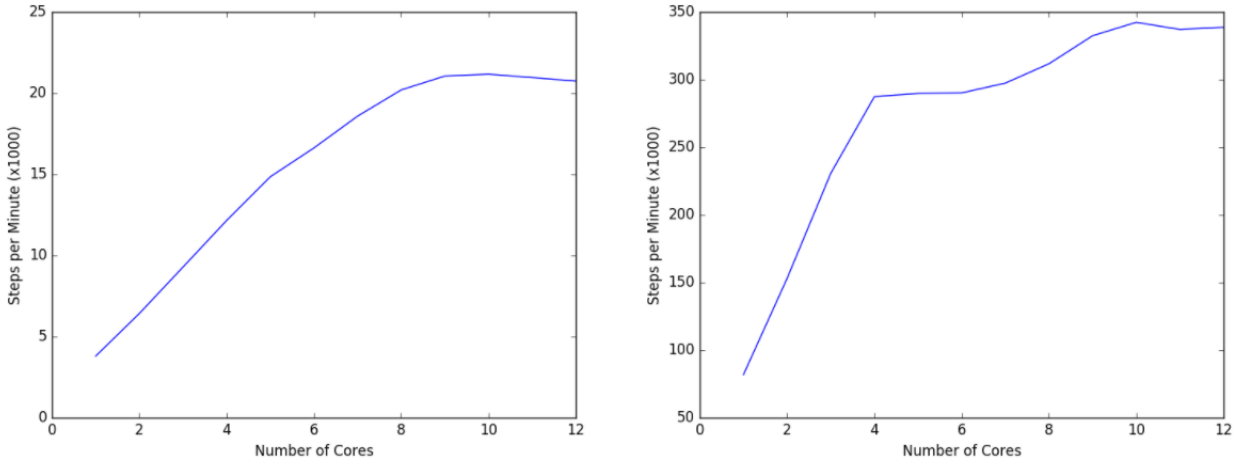

Figure 5.2: Simulator parallelization benchmarking results. Left image is benchmarking with rendering. Right image is benchmarking with rendering disabled.

Chapter 6

Experiments

We benchmark the performance of imitation learning and reinforcement learning methods on this environment.

6.1 Configuration

We include the particular set of configurations used for these experiments. The agent's car has initial state variance uniform over [-30, 30] for starting angles. There are five other cars uniformly scattered within a grid that spans up to 100 time steps forward in the road from the car's current location. The agent only has control over its steering and is forced to accelerate up to its maximum speed at twice the speed of the other cars, forcing it to learn to steer around the other cars while avoiding travelling off the road. The road is also narrow enough to make it impossible to turn around.

6.2 Evaluation

We evaluate performance of a learning algorithm based on the average number of steps the agent travels before the current trajectory is terminated. The current trajectory is marked as "done" and terminated immediately when either the main car collides with another car or runs off the track, or if the maximum time horizon of 100 time steps is reached.

6.3 Imitation Learning

Algorithms

One set of experiments we perform is analyzing the performance of different imitation learning algorithms on this environment. We evaluate DAgger [42], Dart [28], and ordinary supervised learning, as well as variants on these algorithms.

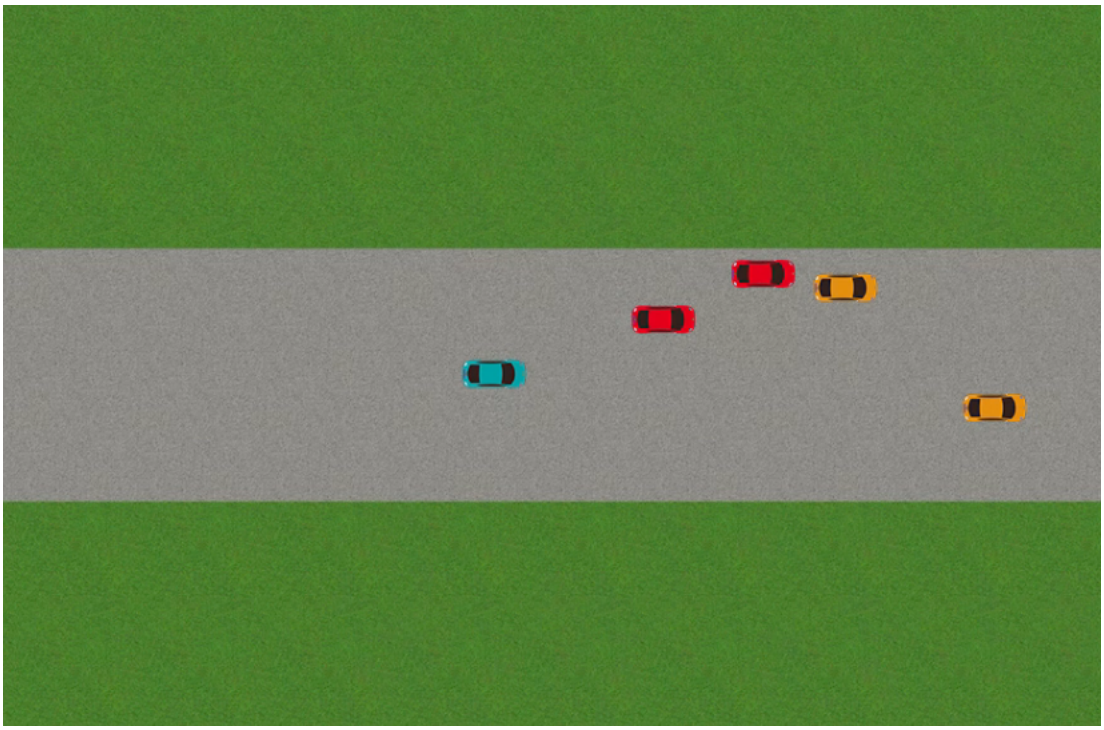

Figure 6.1: Environment used for Experiments

Configurations

For this set of experiments, the state space of the simulator is gray-scale 8-bit images of the rendered image, in the set $S = [0,255]^{300\times300}$. Our neural net architecture features a convolutional layer with 5 filters of dimension 7x7 and a fully connected hidden layer of dimension 60. Each layer is separated by ReLU non-linearities. The images are centered around the agent's car. All experiments feature a cost-based search planner as a supervisor to provide demonstrations, which has access to the lower dimensional internal state space of the simulator. The cost function is weighted to promote navigating around cars while keeping closer to the center of the road. The supervisor is able to achieve a reward of approximately 70 on average on this particular environment. We provide the results of these experiments, plotting average episode reward against the number of demonstrations provided by the supervisor.

Results

On this particular environment, we find that Dart-0.5 performs the best with an average score of approximately 60. The other three imitation learning methods achieve lower scores around 40-50 by the end of the experiment. All of these imitation learning methods outperform ordinary supervised learning, which achieves a score of approximately 20.

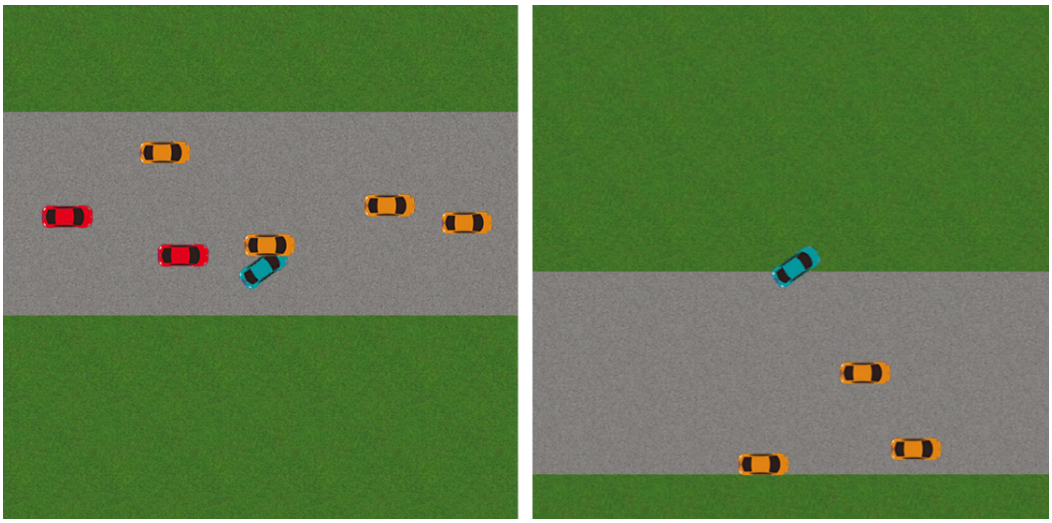

Figure 6.2: Examples of states considered as crashes in the experiment. Left image illustrates an example of colliding with a car. Right image illustrates an example of running off the main road.

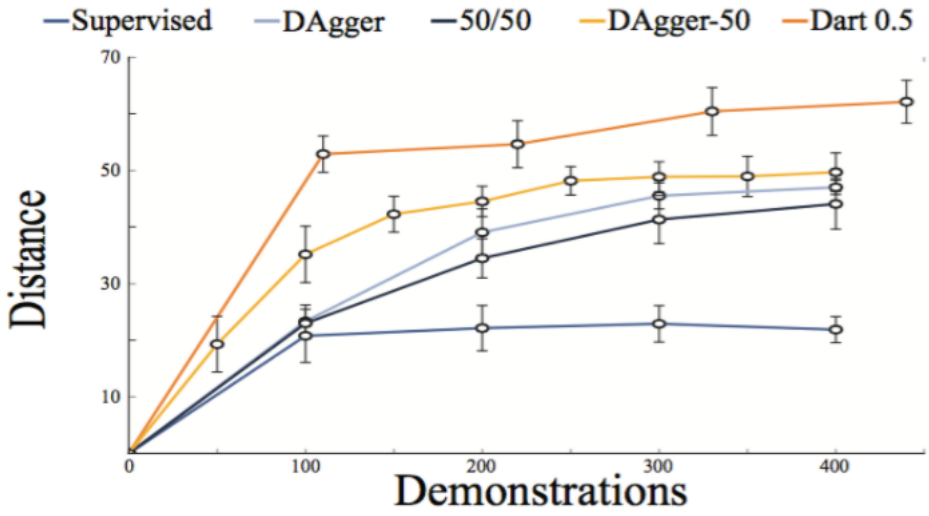

Figure 6.3: Performance of Imitation Learning Algorithms

6.4 Reinforcement Learning

Algorithms

We also perform another set of experiments evaluating the performance of different reinforcement learning algorithms on this environment. We evaluate REINFORCE [50], Trust Region Policy Optimization [45], Truncated Natural Policy Gradient [25, 3], and Reward Weighted Regression [38, 26].

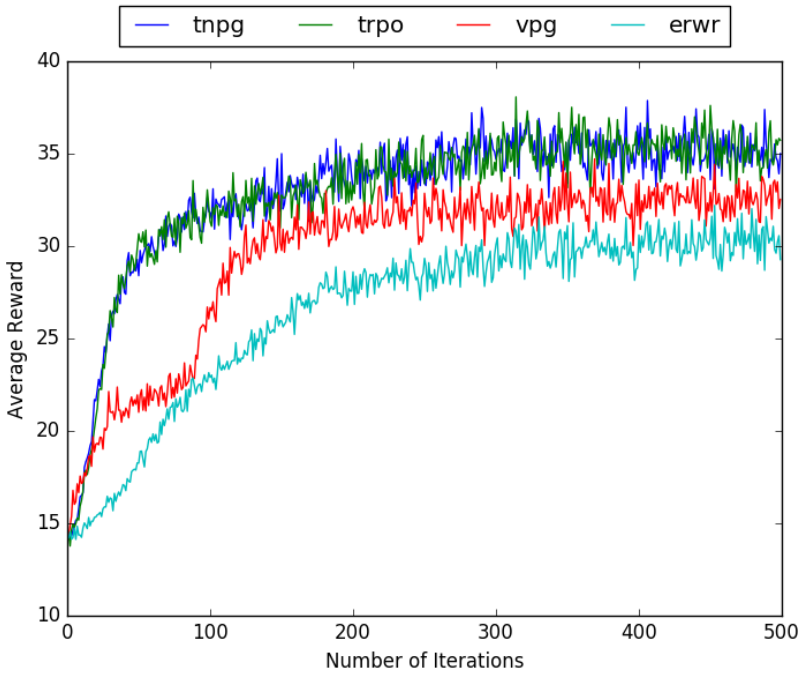

Figure 6.4: Performance of Reinforcement Learning Algorithms

Configurations

We feature examples integrating the environment with Rllab [14], which features implementations of these as well as other reinforcement learning algorithms. For this set of experiments, to reduce the state space of the simulator, we provide the state space as the positions and poses of all of the cars, in the set $\mathbb{R}^{18}$. Each of these algorithms is run with a batch size of 40000 time steps per update iteration, with a total of 500 iterations, and a step size of 0.01. Our neural net architecture features two fully connected hidden layers of size 64. Each layer is separated by tanh non-linearities. Including overhead from parallelization, we collect around 200-250 time steps per second per thread. We provide the results of these experiments, plotting average episode reward against the number of update iterations taken by the reinforcement learning algorithms.

Results

On this particular environment, we find that Truncated Natural Policy Gradient (TNPG) and Trust Region Policy Optimization (TRPO) perform the best with a score of approximately 35. Vanilla Policy Gradient (VPG) achieves a score around 32. Reward Weighted Regression (ERWR) achieves a score of around 30. All of these algorithms plateau and achieve their best performance around 250 iterations.

Chapter 7

Discussion and Future Work

7.1 Discussion

We present a lightweight simulator designed for autonomous driving experiments, with a focus on customizability and speed of integration with existing learning pipelines.The state space is a 2D birds-eye view of the simulator centered around the car, with simple graphics for performance. The car features the choice between kinematic and dynamic bicycle models with slip and friction. Its input controls include steering, acceleration, and braking, and can be input from an external program or manually through a keyboard or Xbox controller.

To facilitate ease of integration with external modules, the simulator implements the OpenAI Gym interface [8], a popular environment interface for learning problems. We also feature parallelization with Ray [36], a distributed execution framework aimed at making it easy to parallelize existing codebases, which allows for significant speed increases in data collection. We benchmark our performance against other simulators of varying degrees of complexity, and show that our simulator matches or outperforms their speeds of data collection.

7.2 Future Work

While the core functionality of the project has been implemented, there are multiple additional features we wish to implement in the future to improve usability and performance.

GUI

One important feature is to create a graphical user interface for the configuration module, as opposed to the current command-line interface. This will greatly improve the speed and usability of configuring the environment, especially for features that are more intuitively represented graphically such as track placement.

Box2D

We also wish to investigate porting the core of the system to Box2D [11], a two-dimensional physics simulator for games that offers a Python interface [21]. Box2D is a more complete game engine than PyGame due to better supported rendering features as well as direct native support for handling physics simulations. We may offer a choice between the different backends depending on comparisons of performance.

Symbolic Dynamics Models

Another possible feature is to allow specification of dynamics models symbolically in Tensorflow [1] or Theano [5]. Currently, the dynamics models for most simulators including this one are opaque and require inspection of the source code to extract. By specifying the models symbolically, the dynamics models become more readily usable for traditional motion planning approaches that require knowledge of the dynamics model. This allows for easier comparisons between analytically planning models and data-driven machine learning approaches. Additionally, this will allow for easier imports of custom dynamics models, rather than directly implementing a sub-class of the car model and overwriting the corresponding functions.

Use Cases

From a usability perspective, we also wish to improve the documentation of use cases of the simulator. We hope to provide more examples of configurations for different simulator setups. We also hope to include more examples of scripts for running different imitation learning or reinforcement learning algorithms, as well as integration with classical control algorithms that require symbolic specification of dynamics models.

Bibliography

[1] Martín Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] R Wade Allen et al. *A low cost PC based driving simulator for prototyping and hardware-in-the-loop applications*. Tech. rep. SAE Technical Paper, 1998.

[3] J Andrew Bagnell and Jeff Schneider. "Covariant policy search". In: IJCAI. 2003.

[4] Sven A Beiker. "Legal aspects of autonomous driving". In: *Santa Clara L. Rev.* 52 (2012), p. 1145.

[5] James Bergstra et al. "Theano: Deep learning on gpus with python". In: *NIPS 2011, BigLearning Workshop, Granada, Spain*. Vol. 3. Citeseer. 2011.

[6] Mariusz Bojarski et al. "End to end learning for self-driving cars". In: *arXiv preprint arXiv:1604.07316* (2016).

[7] Mariusz Bojarski et al. "Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car". In: *arXiv preprint arXiv:1704.07911* (2017).

[8] Greg Brockman et al. "OpenAI gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[9] Mark Campbell et al. "Autonomous driving in urban environments: approaches, lessons and challenges". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 368.1928 (2010), pp. 4649–4672.

[10] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. "Learning drivers for TORCS through imitation using supervised methods". In: *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE. 2009, pp. 148–155.

[11] Erin Catto. *Box2D*. `https://github.com/erincatto/Box2D`. 2007.

[12] Chenyi Chen et al. "Deepdriving: Learning affordance for direct perception in autonomous driving". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 2722–2730.

[13] Chenyi Chen et al. "Learning Affordance for Direct Perception in Autonomous Driving". In: ().

[14] Yan Duan et al. "Benchmarking deep reinforcement learning for continuous control". In: *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. 2016.

[15] Gijs Dubbelman and Frans CA Groen. "Bias reduction for stereo based motion estimation with applications to large scale visual odometry". In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE. 2009, pp. 2222–2229.

[16] Azim Eskandarian. *Handbook of intelligent vehicles*. Springer London, 2012.

[17] Artur Filipowicz, Jeremiah Liu, and Alain Kornhauser. *Learning to Recognize Distance to Stop Signs Using the Virtual World of Grand Theft Auto 5*. Tech. rep. 2017.

[18] Jannik Fritsch, Tobias Kuhnl, and Andreas Geiger. "A new performance measure and evaluation benchmark for road detection algorithms". In: *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on*. IEEE. 2013, pp. 1693–1700.

[19] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for autonomous driving? the kitti vision benchmark suite". In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE. 2012, pp. 3354–3361.

[20] Raymond Ghandour et al. "Tire/road friction coefficient estimation applied to road safety". In: *Control & Automation (MED), 2010 18th Mediterranean Conference on*. IEEE. 2010, pp. 1485–1490.

[21] Gustavo Goretkin. *pybox2d*. `https://github.com/pybox2d/pybox2d`. 2011.

[22] Petr Hejtmánek et al. "Measuring the Yaw Moment of Inertia of a Vehicle". In: *Journal of Middle European Construction and Design of Cars* 11.1 (2013), pp. 16–22.

[23] Itseez. *Open Source Computer Vision Library*. `https://github.com/itseez/opencv`. 2015.

[24] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 2017]. 2001–. URL: `http://www.scipy.org/`.

[25] Sham M Kakade. "A natural policy gradient". In: *Advances in neural information processing systems*. 2002, pp. 1531–1538.

[26] Jens Kober and Jan R Peters. "Policy search for motor primitives in robotics". In: *Advances in neural information processing systems*. 2009, pp. 849–856.

[27] Jason Kong et al. "Kinematic and dynamic vehicle models for autonomous driving control design". In: *Intelligent Vehicles Symposium (IV), 2015 IEEE*. IEEE. 2015, pp. 1094–1099.

[28] Michael Laskey et al. "Iterative Noise Injection for Scalable Imitation Learning". In: *arXiv preprint arXiv:1703.09327* (2017).

[29] Michael Laskey et al. "Shiv: Reducing supervisor burden in dagger using support vectors for efficient learning from demonstrations in high dimensional state spaces". In: *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE. 2016, pp. 462–469.

[30] Jesse Levinson et al. "Towards fully autonomous driving: Systems and algorithms". In: *Intelligent Vehicles Symposium (IV), 2011 IEEE*. IEEE. 2011, pp. 163–168.

[31] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[32] Guan-Horng Liu, Sai Prabhakar, and Avinash Siravuru. “Deep-RL using Overcomplete State Representation”. In: ().

[33] Daniele Loiacono et al. “Learning to overtake in torcs using simple reinforcement learning”. In: *Evolutionary Computation (CEC), 2010 IEEE Congress on*. IEEE. 2010, pp. 1–8.

[34] Volodymyr Mnih et al. “Asynchronous methods for deep reinforcement learning”. In: *International Conference on Machine Learning*. 2016, pp. 1928–1937.

[35] Russell Lee Mueller. “Full vehicle dynamics model of a formula SAE racecar using ADAMS/-Car”. PhD thesis. Texas A&M University, 2005.

[36] Robert Nishihara et al. *Ray*. `https://rise.cs.berkeley.edu/projects/ray`. 2017.

[37] Diego Perez, Gustavo Recio, and Yago Saez. “Evolving a fuzzy controller for a car racing competition”. In: *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE. 2009, pp. 263–270.

[38] Jan Peters and Stefan Schaal. “Reinforcement learning by reward-weighted regression for operational space control”. In: *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 745–750.

[39] Dean A Pomerleau. *ALVINN, an autonomous land vehicle in a neural network*. Tech. rep. Carnegie Mellon University, Computer Science Department, 1989.

[40] Rajesh Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[41] Stephan R Richter et al. “Playing for data: Ground truth from computer games”. In: *European Conference on Computer Vision*. Springer. 2016, pp. 102–118.

[42] Stéphane Ross, Geoffrey J Gordon, and Drew Bagnell. “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning.” In: *AISTATS*. Vol. 1. 2. 2011, p. 6.

[43] Aitor Ruano. *DeepGTAV*. `https://github.com/ai-tor/DeepGTAV`. 2016.

[44] Dorsa Sadigh et al. “Planning for autonomous cars that leverages effects on human actions”. In: *Proceedings of the Robotics: Science and Systems Conference (RSS)*. 2016.

[45] John Schulman et al. “Trust region policy optimization”. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015, pp. 1889–1897.

[46] Pete Shinners. *PyGame*. `http://pygame.org/`. 2011.

[47] Wolfgang Sienel. “Estimation of the tire cornering stiffness and its application to active car steering”. In: *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*. Vol. 5. IEEE. 1997, pp. 4744–4749.

[48] Chris Urmson et al. “Autonomous driving in traffic: Boss and the urban challenge”. In: *AI magazine* 30.2 (2009), p. 17.

[49] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[50] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[51] Bernhard Wymann et al. *TORCS: The open racing car simulator*. 2015.

[52] Huazhe Xu et al. “End-to-end Learning of Driving Models from Large-scale Video Datasets”. In: *arXiv preprint arXiv:1612.01079* (2016).

[53] Naoto Yoshida. *Gym TORCS*. 2016.

[54] Jiakai Zhang and Kyunghyun Cho. “Query-efficient imitation learning for end-to-end autonomous driving”. In: *arXiv preprint arXiv:1605.06450* (2016).