arXiv:2005.05178v1 [cs.RO] 6 May 2020

DeepRacing: Parameterized Trajectories for Autonomous Racing

Trent Weiss
Department of Computer Science
University of Virginia
`ttw2xk@virginia.edu`

Madhur Behl
Department of Computer Science
University of Virginia
`madhur.behl@virginia.edu`

May 12, 2020

Abstract

We consider the challenging problem of high speed autonomous racing in a realistic Formula One environment. DeepRacing is a novel end-to-end framework, and a virtual testbed for training and evaluating algorithms for autonomous racing. The virtual testbed is implemented using the realistic F1 series of video games, developed by Codemasters©, which many Formula One drivers use for training. This virtual testbed is released under an open-source license both as a standalone C++ API and as a binding to the popular Robot Operating System 2 (ROS2) framework. This open-source API allows anyone to use the high fidelity physics and photo-realistic capabilities of the F1 game as a simulator, and without hacking any game engine code. We use this framework to evaluate several neural network methodologies for autonomous racing. Specifically, we consider several fully end-to-end models that directly predict steering and acceleration commands for an autonomous race car as well as a model that predicts a list of waypoints to follow in the car's local coordinate system, with the task of selecting a steering/throttle angle left to a classical control algorithm. We also present a novel method of autonomous racing by training a deep neural network to predict a parameterized representation of a trajectory rather than a list of waypoints. We evaluate these models performance in our open-source simulator and show that trajectory prediction far outperforms end-to-end driving. Additionally, we show that open-loop performance for an end-to-end model, i.e. root-mean-square error for a model's predicted control values, does *not* necessarily correlate with increased driving performance in the closed-loop sense, i.e. actual ability to race around a track. Finally, we show that our proposed model of parameterized trajectory prediction outperforms both end-to-end control and waypoint prediction.

1 Introduction

Vision-based solutions are believed to be a promising direction for autonomous driving due to their low sensor cost, and recent developments in deep learning. End-to-end models for autonomous driving have attracted much research interest [Santana and Hotz, 2016, Janai et al., 2017, Xu et al., 2016a], because they eliminate the tedious process of feature engineering. Algorithms for end-to-end driving are being trained and evaluated in both simulation [Perot et al., 2017, Chen et al., 2015]. These methods show promise and in some cases are implemented on real vehicles [Bojarski et al., 2016]. However, there is room for significant progress as these studies primarily use simulators with simplified graphics and physics [Wymann et al., 2000, Brown et al., 2018] and so the obtained driving results lack realism. Additionally, there is little work showing how these methods will perform under "edge" or "corner" cases. That is, highly unusual or unexpected scenarios where an autonomous agent must make a decision in an environment it has either not seen or wasn't specifically programmed to handle. These cases are the cause of many failures in autonomous driving agents [tes, 2018, Wakabayashi]. There is an unfilled need to directly address these edge cases with autonomy that can be *agile* and is capable of handling these unexpected edge cases safely. As a first step toward agile autonomy, we consider the project of high-speed autonomous racing, specifically Formula One, as a proxy for agile behavior. This approach is not unprecedented. In the early 20th century, a time of great consternation about giving up horses in favor of motorcars, Ford Motor Company used motorsport racing as a means of demonstrating to the public that it's products were safe for everyday users. Additionally, many of the now-standard features of commercially available automobiles were conceived as innovations in motorsport racing.

Demonstrating high-speed autonomous racing can be considered as a grand challenge for vision based end-to-end models. Autonomous racing can be considered an extreme version of the self-driving car problem, making progress here has the potential to enable breakthroughs in agile and safe autonomy. To succeed at racing, an autonomous vehicle is required to perform both precise steering and throttle maneuvers in a physically-complex, uncertain environment, and by executing a series of high-frequency decisions. Autonomous racing is also highly likely to become a futuristic motorsport featuring a head-to-head complex battle of algorithms Scacchi [2018]. For instance, Roborace [rob] is the Formula E's sister series, which will feature fully autonomous race cars in the near future. Autonomous racing competitions, such as F1/10 racing and Autonomous Formula SAE [Koppula, 2017] are, both figuratively and literally, getting a lot of traction and becoming proving grounds for testing perception, planing, and control algorithms at high speeds.

We present DeepRacing, a novel end-to-end framework for training and evaluating algorithms specifically for autonomous racing. DeepRacing uses the Formula One (F1) Codemasters game as a virtual testbed. This game is highly realistic - both in physics and graphics - and is used by many real world F1 drivers for training. Our DeepRacing C++ API enables easy generation of training data under a variety of racing environments, without the cost and risk of a physical racecar, and racetrack. This allows anyone to use the high fidelity physics and photo-realistic capabilities of the F1 game as a simulator, and without hacking any game engine code. The DeepRacing framework is open-source *https://github.com/linklab-uva/deepracing.*

In addition we present AdmiralNet - a novel neural network architecture for predicting a parameterized trajectory (specifically, Bézier Curves) for the autonomous agent to follow. We conduct a case study of this model against several other autonomous racing approaches. Our evaluation demonstrates the ability to train and test end-to-end autonomous racing algorithms using this highly photo-realistic video game environment.

1.1 Contributions of this paper

The paper has the following contributions:

1. This is the first paper to demonstrate and enable the use of a highly photo-realistic Formula 1 Codemasters© game, with a high fidelity physics engine, as a test-bed for developing autonomous racing algorithms, and testing them in a closed-loop manner.
2. We implement and evaluate a deep neural network (DNN) called AdmiralNet. AdmiralNet uses a novel combination of Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Bézier Curves to autonomously race.
3. We compare AdmiralNet with several other approaches: both fully end-to-end (direct steering/acceleration prediction) and waypoint prediction in our closed-loop F1 testbed

Section 2 provides a description of the problem and it's mathematical formulation. Section 3 describes some related work in this domain. Section 4 describes our open-source testbed and evaluation framework. Section 5 describes our proposed network architecture and it's underlying mathematical basis in Bézier Curves. Section 6 presents our empirical evaluation of both the open-source testbed and a case study of several autonomous racing methods within that testbed. Section 7 concludes and proposes some future work.

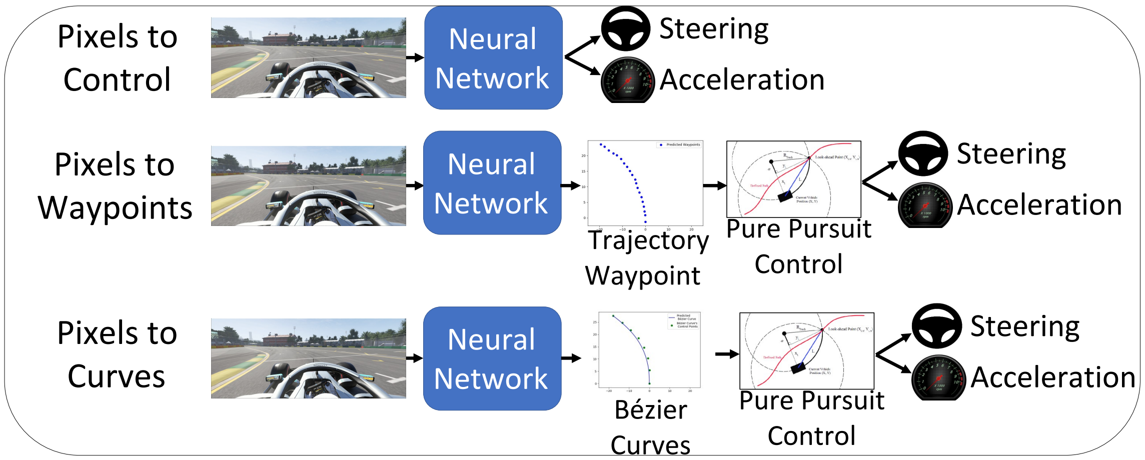


Figure 1: Several approaches to autonomous racing. Ultimately, any control strategy must provide steering and throttle values for the vehicle. We present a case study of these approaches in section 6.

2 Problem Statement:

The problem of autonomous driving distills to the task of scene understanding through sensor measurements, e.g. cameras, LIDAR point clouds, ultrasonic sensors, etc., and producing control inputs for the car, typically steering angle and throttle. Expressed mathematically, if the domain of the vehicle's entire sensor suite is $\mathbb{X}$ and the space of the vehicle's control outputs is $\mathbb{U}$, then the general problem of autonomous driving is a mapping from $\mathbb{X} \to \mathbb{U}$. Autonomous racing requires a great many control inputs: steering, acceleration, clutch, fuel mix selector, clutch bite point, and regenerative braking; just to name a few. For simplicity in this preliminary work, we assume that the control domain of the car is steering and acceleration. It is also assumed that these control values are linearly independent. This means the car's control domain is a 2-dimensional euclidean space, $\mathbb{R}^2$. An autonomous racecar would likely have multiple sensing modalities such as 2D vision, 3D LIDAR, and sound measurement. However, for this work, we focus on a vision based approach and assume that the car's input sensor domain is fixed-width images: $\mathbb{R}^{3xHxW}$, i.e. images with 3 channels (3-channel color images are assumed for this work) of height H and width W.

We consider three high-level approaches to this problem:

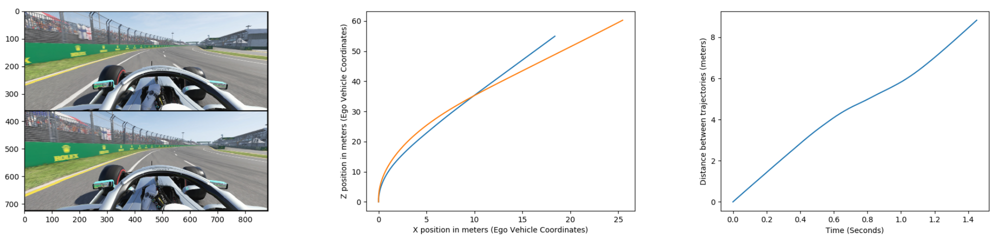
1. Fully End-To-End: training a network to map images directly to control inputs for the autonomous car (steering, throttle, etc.)
2. Trajectory Waypoint Prediction: training a network to map sensor data to a series of waypoints for a car to follow, with a traditional controller such as Pure Pursuit deciding on control values for the car to best follow those waypoints
3. Trajectory Prediction with Bèzier Curves: similar to the second approach, but training a network to map sensor data to a parameterized representation of the trajectory itself rather than a sample of points on that trajectory.

Figure 1 provides a graphical description of these three approaches.

2.1 End-To-End Driving (Pixels to Control)

There is a great body of work in this domain centered around the special case where $\mathbb{U}$ consists of only steering angles. NVIDIA's PilotNet architecture [Bojarski et al., 2017] is considered a seminal work on this approach. . It focuses on mapping what the car's sensor suite is seeing at the present time to a single control command, at the present time. This is done in an end-to-end manner i.e. the DNN is trained to directly map pixels to control outputs: $\mathbb{R}^{3xHxW} \rightarrow \mathbb{R}^2$.

However, this model of autonomous driving does a poor job of capturing how expert drivers behave. For instance, a Formula One racing driver does not simply analyze the pixels directly in front of him and map those pixels directly to a single steering angle and throttle pressure. An expert driver considers a history of previous observations to build up some temporal *context* about the scene. Additionally, using only single static images can create an ill-posed problem in which the map from pixels to control is not a function. Consider the images and corresponding ground-truth trajectories in Figure 2. Even though the images are similar, their ground-truth trajectories differ noticeably. Within only 1.4 seconds, the paths diverge by almost 9 meters. In a high-speed scenario like racing, this could easily be the difference between a successful race and a devastating crash.



(a) Two similar images (b) Corresponding trajectories (c) Distance between the two trajectories

Figure 2: Purely Markovian methods like CNNs can present an ill-posed problem. For two very similar static images, the trajectories differ significantly.

For example, if a driver an ego vehicle sees a car directly in front of him but knows (from previous observation) that this car is moving away from him, he would behave differently than he would if he saw the exact same car in the exact same position, but with a relative velocity *towards* the ego vehicle. This problem of the exact same scene mapping to multiple control decisions, depending on context, does not lend itself to machine learning techniques that assume a functional mapping between input measurements and output predictions. An alternative approach [Weiss and Behl, 2020] is to consider a sensor reading as a sequence of images: $\mathbf{I}_{i-N}, \mathbf{I}_{i-N+1}, \mathbf{I}_{i-N+2}, ..., \mathbf{I}_i$, where the subscript i represents the current time and c represents some number of time-steps into the past, as a single measurement that is then used to predict a sequence of control outputs $\mathbf{r}_{i+1}, \mathbf{r}_{i+2}, \mathbf{r}_{i+3}, ...\mathbf{r}_{i+P} \in \mathbb{R}^3$, where $\mathbf{r}_k$ represents a control output at time k. This view of the problem is a mapping from a *context window* of sensor readings to an *intent window* of control outputs for the autonomous vehicle:

$$\mathbb{R}^{NxCxHxW} \rightarrow \mathbb{R}^{Px3} \tag{1}$$

However, this fully end-to-end approach has it's weaknesses. For very high-speed systems such as a race car, small errors in a control input map to very large errors in the actual path the car follows and can create catastrophic results. This problem is somewhat intuitive to human drivers. For example, when driving at highway speeds, human drivers prefer not to make aggressive steering maneuvers unless needed to avoid a crash. As opposed to low-speed situations, where quick changes in steering are quite common to make sharp turns, maneuver around an obstacle, or park a vehicle.

2.2 Trajectory Prediction - Pixels to Waypoints

To remedy this problem, one could view the problem of autonomous driving not as a static function from pixels to control, but as a temporally varying task that maps 1-dimensional manifolds, *curves*, in a space of sensor inputs to curves in the ambient task space of the ego vehicle. In this approach, rather than mapping a sequence of context images directly to a sequence of control outputs, the autonomous car needs to predict a sequence of *waypoints* in the ego vehicle's task space. I.e. the problem becomes mapping as sequence of images: $\mathbf{I}_{i-N}, \mathbf{I}_{i-N+1}, \mathbf{I}_{i-N+2}, ..., \mathbf{I}_i$ to a sequence of waypoints: $\vec{r}_{i+1}, \vec{r}_{i+2}, \vec{r}_{i+3}, ...\vec{r}_{i+P} \in \mathbb{R}^D$ where D is the dimensionality of the task space. Expressed mathematically, a function of the form:

$$\mathbb{R}^{Nx3xHxW} \rightarrow \mathbb{R}^{P\ x\ D} \tag{2}$$

For this work, we consider the sensor space as only images and the ambient task space is assumed to be a 2-dimensional manifold embedded in $\mathbb{R}^3$. This approach falls along a similar lines as end-to-end control by mapping a context window to a long-term intent, but the intent window is a set of points for the car to follow rather than a control schedule. The task of mapping a predicted trajectory for the car to follow down to a specific steering and throttle value for the car's low-level control is left to classical control techniques based on a bicycle model of the car's kinematics. For this work, a Pure Pursuit controller is used, but this model is generalizable to other control strategies.

This approach also has it's weaknesses. It suffers from the so-called "curse of dimensionality" and can produce very non-smooth paths. With so many parameters required to represent a single output ($P\ x\ D$ of them), models that are already prone to over-fitting like deep neural networks can predict very noisy output trajectories that can vary significantly with only minor (possibly imperceptible) changes in the input images. Additionally, because these parameters represent *waypoints*, they are not just a set of real numbers, their *ordering* has specific and actionable meaning. If a model predicts even a single waypoint incorrectly, this can cause a serious error in the car's chosen control action. This is especially true for our chosen control algorithm of Pure Pursuit, if the incorrectly predicted point happens to be chosen as the lookahead point. Section 5.1 describes what this means in more detail.

2.3 Trajectory Prediction - Pixels to Bezier Curves

Finally, we consider a novel approach to autonomous racing. Rather than training a neural network fully end-to-end, we view the problem of trajectory prediction not as a mapping from images to waypoints, but from images to a parameterized description of a smooth 1-manifold embedded in the car's task space. B-Splines are a very intuitive choice, as they are C^∞ curves and are commonly used for motion planning tasks [Foo et al., 2009, Lian and Murray, 2002]. However, their recursive representation does not fit well with gradient back-propagation, making it difficult to apply them in the context of machine learning. To remedy this, we consider using *Bézier Curves* as a canonical form of curves in an autonomous racecar's task space. Bézier Curves are a linear combination of Bernstein Polynomials and are described in more detail in section 5.2.

Regardless of the chosen approach. Any supervised machine learning method technique requires significant amounts of training data for fitting a model. We describe our fully open-source DeepRacing testbed and data collection infrastructure for gather training data in a highly photorealistic racing environment in Section [4].

3 Related Work

There is existing literature for end-to-end autonomous driving. We divide the related work into simulation testbeds and autonomous driving methods and provide a brief reprise on both.

3.1 Autonomous driving simulators

There exists a great deal of related work in simulation for autonomous driving (AD). Traditionally, simulation capabilities have been primarily used in the planning and control phase of AD [Likhachev and Ferguson, 2009, Buehler et al., 2009, Katrakazas et al., 2015, Anderson et al., 2010, Best et al., 2017]. More recently, simulation has been used in the entire AD pipeline, from perception and planning to control Pendleton et al. [2017]. Waymo has claimed that its autonomous vehicle has been tested for billions of miles in their proprietary simulation system, CarCraft Madrigal [2017], little technical detail has been released to the public in terms of its fidelity for training machine learning methods. Researchers have tried to use images from video games to train deep-learning-based perception systems Johnson-Roberson et al. [2016], Richter et al. [2016b].

There are several examples of video games being used as a simulator to aid development of autonomous driving. Several are based on the popular Grand Theft Auto (GTA) game [Richter et al., 2016a], utilizing the high fidelity graphics of Rockstar Games'© state-of-the-art rendering engine. However, while being photo-realistic, this game is known for flaws in the underlying physics engine, since it was not intended to be used as a simulator. This technique also requires a modification to the underlying game engine code in order to extract data (steering, acceleration, etc.) attached to each game screenshot. This modification ran afoul of Rockstar's copyright protections, and both of these projects infamously received cease and desist letters from Rockstar Games to pull the code from public domain. While creating DeepRacing, we ensured that our F1 2019 simulator requires no such "hacking" of the unaderlying game engine. It uses public APIs for screen capturing as well as a stream of UDP packets that the F1 game broadcasts. End-to-end driving was showcased in the car racing game TORCS [Wymann et al., 2000] using Reinforcement Learning but its physics and graphics lack realism. The data collected from the racing game TORCS, for example, have a biased distribution in visual background and road traffic and thus severely diverge from real driving scenarios. Microsoft AirSim [Shah et al., 2017] and CARLA [Dosovitskiy et al., 2017] are examples of open-source autonomous driving simulators but they are are largely restricted to urban driving scenarios and are not suited for development and testing of end-to-end autonomous racing.

3.2 Autonomous Driving Architectures: Fully End To End

In one of the earliest work on end-to-end autonomous driving NVIDIA [Bojarski et al., 2017] presented the PilotNet CNN architecture. PilotNet is a feed-forward style network that directly regresses to a single steering value for each input image obtained from a front facing dashboard camera. However, PilotNet is limited by it's inability to capture temporal information. Each input image is run through the CNN separately with no time-varying context around that image. This is not just a limitation of PilotNet, but of CNNs in general. Maqueda et al. [2018] remedy this problem with an event camera to batch images from an arbitrary number of time-steps as an input to a CNN and achieve a performance increase. Fernando et al. [2017] present a different approach that uses Long Short-term Memory Cells [Gers et al., 1999] as a means of capturing a history of the steering trajectory and encode temporal structure of the problem. Xu et al. [2016b] present a similar technique that uses a Fully Convolutional Network (FCN) to extract a feature representation of the input space, but limit their model to classification among a discrete set of actions: go straight, stop, left turn, and right turn.

Eraqi et al. [2017] use a novel combination of CNN and a traditional auto-encoder approach. This network uses a CNN for feature extraction and applies an encoding function to translate the regression problem into a more manageable classification problem. Eraqi also presents the novel concept of a "sliding window", a variable length temporal sequence that is input into the LSTM, allowing the model to encode temporal information at arbitrary lengths. While their results indicate that auto-encoding the LSTM outputs improves performance over directly regressing to steering angle, their model also only predicts steering one time-step into the future and does not capture the expert driver's long-term intent.

Chen et al. [2015] also present a novel approach that blends expert domain knowledge of highway driving by defining a notion of image affordance that is then mapped to a steering command. However, like PilotNet, their approach only considers current image data and is limited to a fixed set of affordance templates that are purpose-built for only highway driving.

3.3 Autonomous Driving Architectures: Pixels to Trajectories

However, end-to-end control has it's limits. Wrapping the entirety of a perception-planning-control pipeline into one black-box network can exacerbate the the problem of overfitting, as a single deep network needs to learn all three high-level tasks as part of a single model. We show in our experiments in section 6 that just because an end-to-end (pixels $\longrightarrow$ control) method performs well on unseen validation data (as measured Root-Mean-Squared error), that does not mean that same method contains a useful driving strategy that will perform well in an actual live-driving test.

To help remedy this problem of over-fitting and produce more scalable methods, there is existing work in autonomous driving by trajectory prediction (pixels $\longrightarrow$ trajectories). For example, Altch and de La Fortelle [2017] use an LSTM cell to predict a series of waypoints for images of highway driving at varying traffic densities. Mohammed et al. [2019] present an approach for generating waypoints with a CNN and then using a Model Predictive Controller (MPC) to control the vehicle. Chen et al. [2019] use a similar approach to predict waypoints based on expert demonstration. Bansal et al. [2018] is a blend of several techniques, but their neural network is penalized (increased loss) for veering too far off of a prescribed list of waypoints.

3.4 Autonomous Driving Architectures: Other Approaches

Kim et al. [2017] use a similar approach but predict an occupancy grid of the ego vehicle's immediate surroundings. This occupancy grid is then used as input for a classical motion planning approach to determine control actions. Pan et al. [2017] use a Reinforcement Learning (RL) approach to autonomously drive in a TORCS simulator. However, their experimental evaluation is limited to an open-loop accuracy evaluation on unseen test data and does not offer closed loop testing to evaluate real driving performance. Michels et al. [2005] use a similar approach and also intentionally focus on high-speed scenarios, similar to racing.

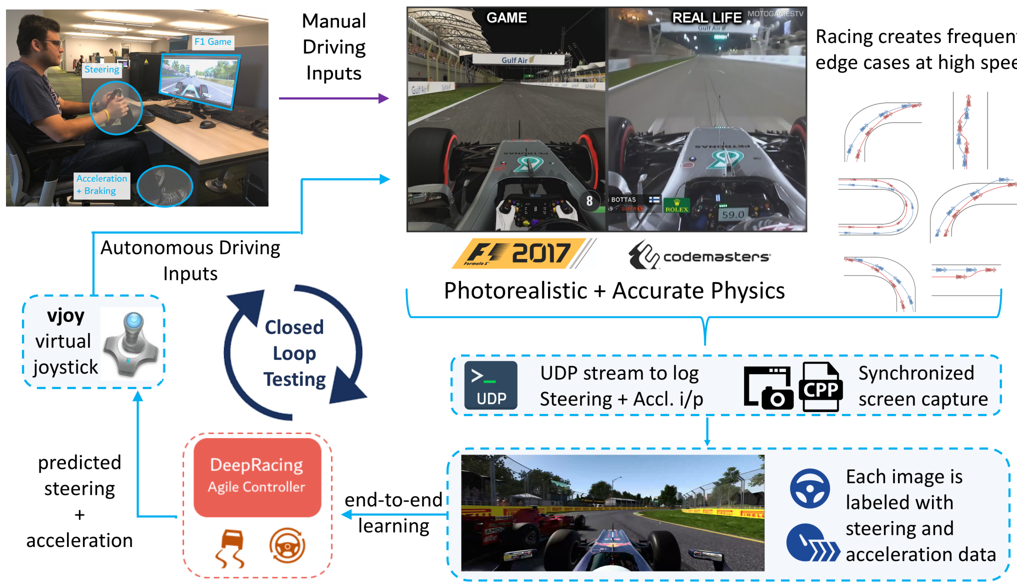
4 DeepRacing: F1 Racing Simulation

In order to train an end-to-end neural network to race autonomously using the context and the intent described in the previous section; we need a reliable way to generate annotated training data. Obtaining such annotated data for real motor-sport racing drivers is difficult since these data are often trade-secrets and not available in the public domain.

Furthermore, it is not enough to only obtain training data, but also important to close the loop and autonomously race in the same environment to enable evaluation of end-to-end models, and reinforcement learning approaches. Consequentially, we use the F1 2019 racing game released by Codemasters© as a realistic testbed for autonomous racing that is built on top of decades of work in 3D graphics and physics simulation. We now describe the novel features this first-of-it's kind testbed provides.

4.1 DeepRacing realism

Photo-realism and physics modeling: The game is extremely photo-realistic, as shown in Figure 3, and is based on high-fidelity simulated Newtonian physics, with detailed simulation of the car's drive train, high-speed aerodynamic drag, and even a simulated traction/slip interface between the car and the track surface. Due to its realism, the F1 series was the first game to be used in the Formula One eSports Series, which debuted in 2017 [esp, 2017]. There is also evidence to suggest that real-life F1 drivers use this game for practicing [max, 2017]. The photo-realism in the driver's point-of-view combined with the physics realism of the game's engine provide a strong opportunity to gather training data as close to real-world racing scenarios as one can get without the cost and risk of a real race-car.
Real weather having real effects: The racecar reacts to weather conditions, in terms of tire degrading, braking, and handling. For example, under wet track conditions braking at the last



F1 Game
Steering
Acceleration + Braking
Manual Driving Inputs
GAME
REAL LIFE
Racing creates frequent edge cases at high speeds
F1 2017
codemasters
Photorealistic + Accurate Physics
Autonomous Driving Inputs
vjoy virtual joystick
Closed Loop Testing
UDP
UDP stream to log Steering + Accl. i/p
CPP
Synchronized screen capture
predicted steering + acceleration
DeepRacing Agile Controller
end-to-end learning
Each image is labeled with steering and acceleration data

Figure 3: We use the F1 Codemasters© game as a virtual testbed for training and closed-loop testing for our autonomous racing deep neural network. This is the first time the highly photo-realistic and high fidelity physics engine enabled game has been used as a CPS testbed in an entirely legal way without cracking and tweaking game engine code.

minute before a corner, the car will have a different response as opposed to braking under dry conditions. Rainstorms are intense, reducing visibility and slowing races down, and again, thats replicated in this game.
Deep customisation: The game facilitates a high degree of customization including adjustable dead zones, linearity, and saturation for vehicle control. Settings for aerodynamics, traction, tyre choices, etc. are also highly customizable.
Closed-Loop Testing: Our infrastructure supports the ability to push steering and throttle commands back into the game. This closed-loop support will enable AI researchers to develop and test autonomous racing policies with this highly realistic game environment.
Figure 3 gives a graphical description of this testbed.

4.2 UDP data stream

The game advertises a "fire-and-forget" data stream of telemetry data containing a variety of information about the game's current state over a User Datagram Protocol (UDP) network socket. Each packet in the stream is a snapshot of the game's state tagged with a timestamp for when that state was generated. A full description of the F1 telemetry stream and all of the information it provides is available on a Codemaster's™ forum Codemasters. The state variables broadcast by the game include, but are not limited to:

1. Steering angle, throttle and brake of all vehicles (including the ego vehicle)
2. Position and velocity of all vehicles (including the ego vehicle)
3. Various state information about the ego vehicle such as wheel speed, amount of fuel remaining, and tire pressure.

4.3 DeepRacing framework

Unfortunately, the game does not tag each packet of state information with an image of the driver's point-of-view at that packet's timestamp. Because supervised machine-learning techniques require *labeled* training data, we present the DeepRacing API, a fully open-source C++17 API for both grabbing screenshots of the driver's perspective (the "ego" vehicle) in the F1 game and automatically tagging them with ground-truth values of the game's state at the time that image was captured. In this C++ framework, a single dedicated process spawns two threads:

1. For capturing screenshots of the drivers point of view, which we call the "screen-capture thread"
2. For listening for telemetry data from the game on a UDP socket, which we call the "telemetry thread"

Each thread has a copy of a shared CPU timer. The screen-capture thread uses our C++17 API, built on top of Microsoft's DirectX, for capturing images of the driver's point of view and tags each image with a system timestamp from the operating system. The telemetry thread listens for UDP data from the game and tags each packet with a system timestamp *from the same system clock.*

4.3.1 Data Synchronization

Because each UDP packet is also tagged with a session timestamp from the F1 game's internal (and black box) game clock, we can then convert OS timestamps to game session timestamps by fitting a least-squares regression line to the session timestamps versus the OS timestamps of the UDP packets. Note that, if our infrastructure is functioning properly, this relationship must be a line with slope close to 1.0, as 1 second on the game clock should correspond to exactly 1 second on the OS clock. We use the r^2 value of this regression line as well as how close its slope is to 1.0 to evaluate how closely the timescale in our logging infrastructure matches the internal clock of the F1 game. Results from this evaluation are included in section 6.

The UDP broadcast operates at 60 Hz and our screencapture framework can achieve frame-rates of up to 35 Hz, enabling high-fidelity data-logging.

4.4 Open-loop testing

Once each image is given a game session timestamp, state information from the UDP stream can then be assigned to each image by interpolation over the relevant state variable with respect to session time. For example, in our experiments, we use B-Spline fitting to determine the ego vehicles position and spherical linear interpolation over unit quaternions to determine the ego vehicle's orientation as a function of session time. We also evaluate our ability to push steering and acceleration control inputs back into the game via a virtual joystick API built on top of vJoy [Eizikovich] by measuring the latency between when a command was applied to the virtual joystick and when that command is reflected in the game's UDP stream. This experiment was conducted by setting the steering angle on our virtual joystick to 0.0 at a known system timestamp, then linearly ramping it up to 1.0 (all the way to the left) at a fixed update rate. We then measure the latency of our system by fitting a regression line to the measured steering angles coming off the UDP stream versus their respective system timestamps. If this system has 0 latency, the x-intercept (time) of this regression line would be the exact system time the ramp-up began. If there is some latency, then the x-intercept will be offset from the ramp start. An example of such a regression experiment is plotted in figure 4. Our experiment shows an x-intercept of 4926.16 milliseconds for a ramp-up that was begun at a system time of 4952.95 milliseconds. This indicates a lagtime of $4952.95 - 4926.16 = 26.79$ milliseconds between when control commands are applied and when they are reflected in the game. However, the UDP stream only publishes at $60Hz$, which means as much as $\frac{1}{60Hz} = 16.\bar{6}$ milliseconds of that delay could stem from the UDP stream itself. In other other words, it's possible that as much as $16.\bar{6}$ milliseconds of the measured lagtime stems from the fact that changes in the game's current state might not get published on the UDP stream for as much as $16.\bar{6}$ milliseconds.

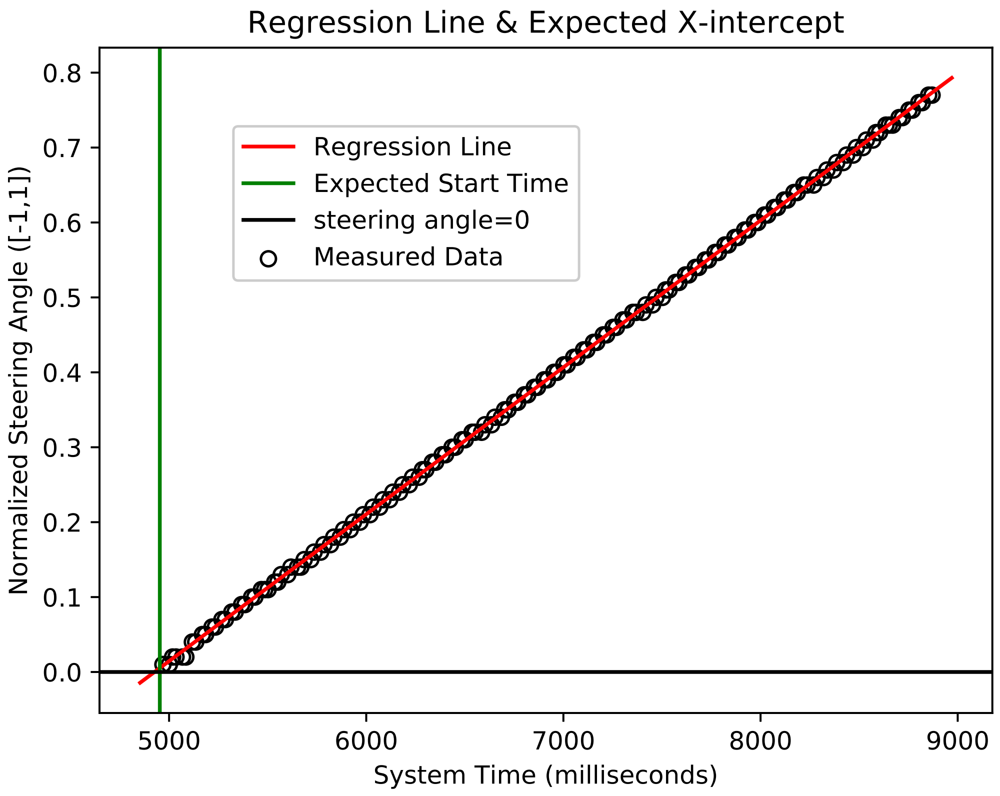


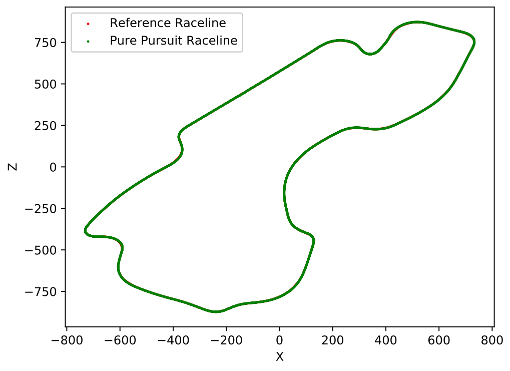
Figure 4: An example experiment evaluating our virtual joystick infrastructure. For this experiment, the ramp-up in steering angle began at a system time of 4.952950 seconds. The regression line shown has an x-intercept of 4.926211 seconds, indicating a latency of 26.79 milliseconds

4.5 Closed-loop testing

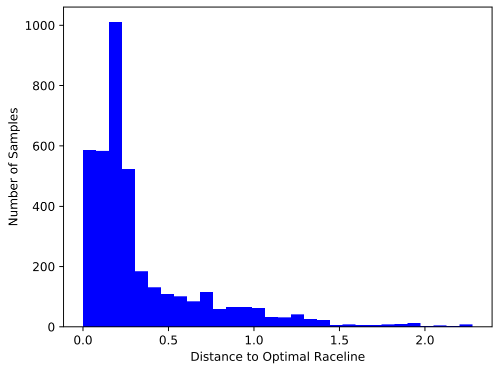
Finally, our test-bed setup also supports the ability to close the loop and autonomously drive the F1 car in the game using control inputs predicted by autonomous driving policies. This is accomplished with the same virtual joystick setup described in subsection 4.4. We evaluate this closed-loop capability by utilizing "oracle" data in the form of a pre-recorded list of vehicle positions (waypoints) that represent the optimal raceline for the Australia circuit in the F1 game and a simple pure-pursuit controller to steer the car with ground-truth knowledge of the track's optimal raceline. We then measure the effectiveness of this controller by the distance from the path followed by the pure-pursuit controller to this optimal raceline. Note that this evaluation is **not** intended to evaluate any particular autonomous driving model, but is intended to show the accuracy of our closed-loop framework for testing such models. This closed-loop test shows a mean distance to the optimal raceline of **0.340936 meters**. Figure 5 a) shows an optimal raceline overlaid with the path followed by our pure pursuit control, the two paths are almost indistinguishable. Figure 5 b) is a histogram of the distance between the pure pursuit control's path and the optimal raceline.

4.6 DeepRacing API

This data collection and testing infrastructure, implemented in C++, is called the DeepRacing API, the first closed-loop environment of it's kind for collecting training data, and testing learned models on simulated F1 race-cars in the photo-realistic F1 2019 game. The software itself is architected as



(a) Racelines



(b) Distance Histogram

Figure 5: A comparison between the path followed by our closed-loop infrastructure an the optimal raceline for the Australia Circuit. The two paths are almost indistinguishable.

an object-oriented library that exposes a simple interface for allowing user-written code to handle data captured by the underlying infrastructure.

A simple interface, called IF1FrameGrabHandler in the C++ API, is exposed that defines what the library does with new timestamped images as they comes off of the screencapture, with the "hard work" of actually obtaining that data hidden in a separate library. User's need only overwrite the appropriate methods in this interface with an implementation of what to do with captured images. A similar interface, called IF1DatagrabHandler in the C++ API, is exposed for handling captured UDP packets. Again, users need only overwrite a few methods to receive timestamped UDP packets as a C++ object. This clean abstraction allows users to define whatever data flow model makes sense for their application. To generate datasets for our experiments in 6, we implement this interface to save image data and UDP data to the file system with a buffered approach to ensure no packets or images are lost during the delay inherent to filesystem access. Other data flow models are possible, e.g. sending data to a cloud-based server, displaying to screen, etc. Creating such models within our framework is only a matter of implementing the simple C++ interfaces.

4.7 DeepRacing ROS2 Interface:

Additionally, we present bindings to the very popular Robot Operating System 2.0 (ROS2). A ROS2 Publisher class is provided that broadcasts both telemetry data and images from the F1 game on several ROS2topics (fire-and-forget broadcasts). Any ROS2 nodes can then listen to these data streams, enabling researchers to integrate this testbed with existing codebase in a ROS2 ecosystem. All of this software has been released under an Open-Source license.

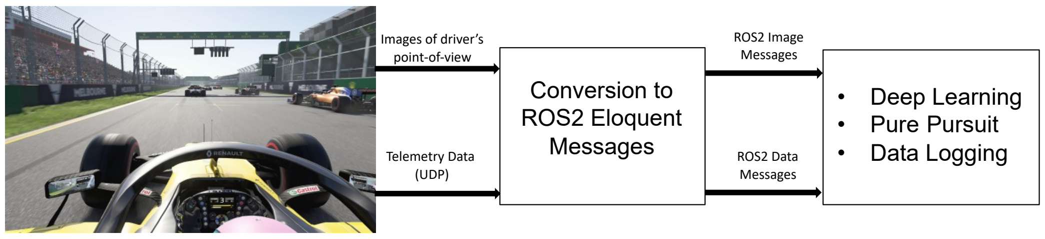


Figure 6: ROS2 integration for the DeepRacing framework. In this figure, "Image Transport" refers to an open-source ROS2 library that handles compressing/decompressing images and routing them through the ROS2 network via a set of rostopics. F1 2019$^{\text{TM}}$is a registered trademark of Codemasters©. The Formula One logo is a registered trademark of the Formula One Group.

5 AdmiralNet: Mapping Context to Intent

The second research contribution of this work is AdmiralNet, an improvement over NVIDIA's PilotNet Bojarski et al. [2017], a deep neural network capable of learning to race autonomously. We train two forms of AdmiralNet, one designed to map images to waypoints for a car to follow, and another to map images to a parameterized trajectory for a car to follow, in our case: Bézier Curves. The key building blocks in AdmiralNet's architecture are:

1. A Pure Pursuit low-level controller.
2. 2D Convolution Neural Networks
3. 3D Convolution Neural Networks
4. Recurrent Neural Networks
5. Bézier Curves

We go over each of these components of our implementation next.

5.1 Pure Pursuit Control

Pure pursuit, first described in Wallace et al. [1985], is a widely used algorithm for path following in front-wheel steered, non-holonomic vehicles [Naeem et al., 2004, Samuel et al., 2016]. At it's core, given a sequence of waypoints $\{W_1, W_2, W_3, ...W_n \in \mathbb{R}^d\}$ expressed in the car's local coordinate system, Pure Pursuit sets a steering angle that puts the car on the circular arc connecting it's current position and a desired "lookahead-point"(see Figure 7). A common practice is to select the point that is closest to a *lookahead distance* away from the ego vehicle, with the lookahead distance set to a constant factor times the ego vehicle's current speed. That is:

$$d_{lookahead} = \gamma v$$
$$W_{lookahead} = \underset{W}{\arg\min} \; | \, \|W\| - d_{lookahead}|$$

Where v is the ego vehicle's current speed and γ is a tuneable parameter. This proportional control strategy keeps the car from making maneuvers that are too aggressive for the car's actuators to handle at the expense of potentially "cutting corners" on sudden sharp turns at high speed.

As will be described later, in our novel AdmiralNet framework, we produce a set of waypoints for the autonomous racecar to follow, and the low-level Pure-Pursuit controller will generate the steering and velocity commands to follow those waypoints.

We evaluate the closed-loop performance of our pure-pursuit implementation in Section 6.

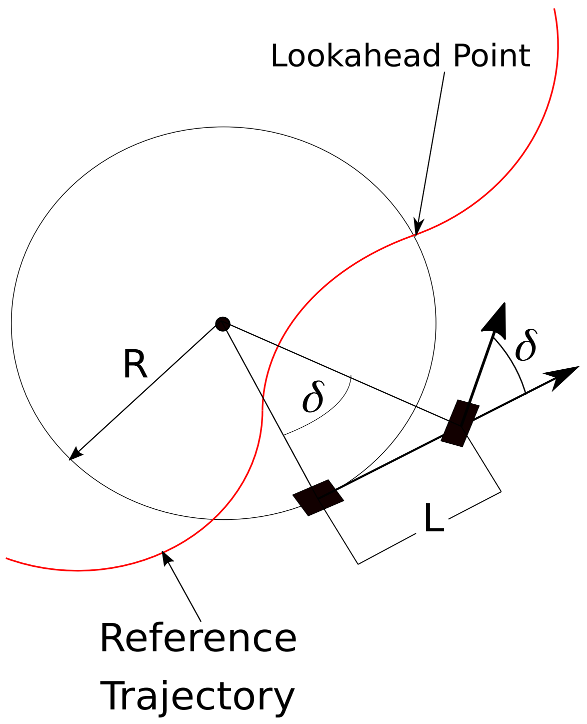


Figure 7: Illustration of Pure Pursuit Control. A steering angle δ is selected such that a car with wheelbase (distance between the axles) L follows a circular arc towards a lookahead point selected from a reference trajectory. The radius of this circle, R, can vary as it needs to and becomes infinite in the case of the car directly facing the lookahead point

5.2 Bézier Curves

A Bézier curve is a parametric curve heavily used in computer graphics and related fields. The curve, a linear combination of Bernstein Polynomials, is named after Pierre Bézier, who (somewhat poetically) developed them to model car bodies on Renault racecars.

A Bézier curve is formed from a combination of Bernstein polynomials that maps a scalar parameter $t \in [0, 1]$ to a point in a euclidean space of dimension d, $\mathbb{R}^d$. More specifically, a Bézier Curve is a weighted combination of a set of "control points", with the weights computed from the Bernstein polynomial basis. These control points define a Bézier curve as follows [Mortenson, 1999]:

For a set of control points:

$$\mathbb{P} = \{\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, ..\mathbf{P}_n \in \mathbb{R}^d\}$$

The corresponding Bézier curve, $\mathbf{B} : [0, 1] \to \mathbb{R}^d$, as a function of the parameter, t, is:

$$\mathbf{B}(t) = \sum_{k=0}^{n} \binom{n}{k} (1-t)^{n-k} t^k \mathbf{P}_k$$

. Figure 8 shows several such curves. For most practical cases, as well as our case, the parameter t will represent time (or step size) normalized to the interval $[0, 1]$

The dimensionality of the ambient euclidean space, d, is called the "dimension" of the Bézier curve and the integer n, 1 less than the number of control points, is called the "degree" or "order" of the curve and is also the degree of the underlying Bernstein polynomials. If the parameter, t, is normalized to the interval $[0, 1]$, then the Bézier curve always starts at $\mathbf{P}_0$ and ends at $\mathbf{P}_n$, but does not necessarily pass through any of the control points from $\mathbf{P}_1$ to $\mathbf{P}_{n-1}$.

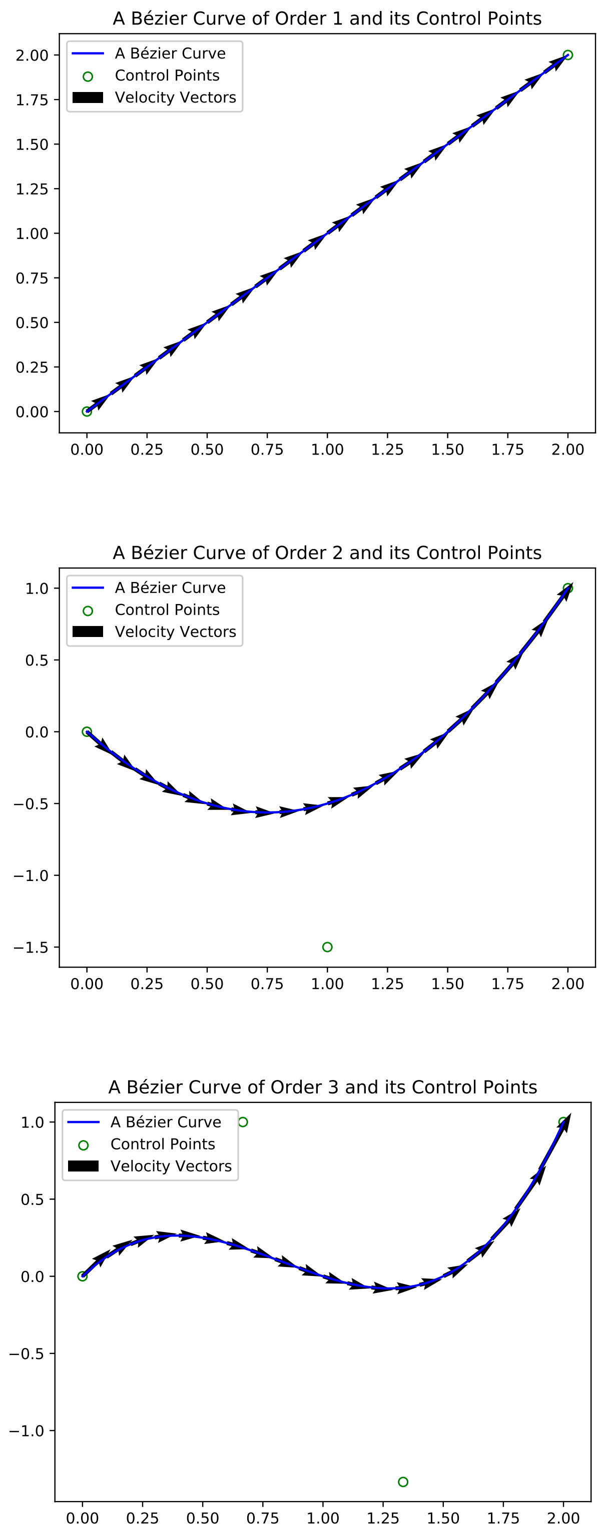


Figure 8: Several Bézier Curves of varying degree

We present a method of evaluating Bézier curves and their derivatives framed as a *matrix multiplcation*, making such a step very easy to integrate into a machine learning model trained by a gradient back-propagation algorithm. Consider a fixed time-vector, $\mathbf{t}$ with N elements linearly spaced on the interval $[0,1]$ i.e $\mathbf{t}_k = \frac{k}{N-1}$:

$$\mathbf{t} = [0, \frac{1}{N-1}, \frac{2}{N-1}, \frac{3}{N-1}, ..., \frac{N-3}{N-1}, \frac{N-2}{N-1}, 1]$$

Consider a time matrix: $\mathbf{A}$ such that

$$\mathbf{A_{ij}}(\mathbf{t}, n) = \binom{n}{j}(1-\mathbf{t}_i)^{n-j}\mathbf{t}_i{}^j$$

$\mathbf{A}$ would then be N rows by n columns with each row corresponding to a time on the interval $[0,1]$ and each column corresponding to a term of the Bézier curve's Bernstein basis polynomial.

Consider a control point matrix, $\mathbf{P}$, such that the k_{th} row of $\mathbf{P}$ is just the k_{th} control point of the curve.

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_0 \\ \mathbf{P}_1 \\ \mathbf{P}_2 \\ ... \\ \mathbf{P}_{n-2} \\ \mathbf{P}_{n-1} \\ \mathbf{P}_n \end{bmatrix}$$

$\mathbf{P}$ would then be $n+1$ rows by d columns, with each column corresponding to a dimension in the curves ambient euclidean space.

Evaluating a Bézier curve, $\mathbf{B}$, of order n with a control point matrix $\mathbf{P}$ on all of the times in $\mathbf{t}$ then just becomes a matrix multiplication:

$$\mathbf{B}(\mathbf{t}) = \mathbf{A}(\mathbf{t}, n)\mathbf{P}$$

Additionally, Bézier curve's derivatives can also be evaluated as follows:

$$\mathbf{B}^{'}(t) = n\sum_{k=0}^{n-1}\binom{n-1}{k}(1-t)^{n-k-1}t^k(\mathbf{P}_{k+1} - \mathbf{P}_k)$$

.

Note that this is equivalent to:

$$\mathbf{B}^{'}(t) = n\mathbf{A}(\mathbf{t}, n-1)\Delta\mathbf{P}$$

Where

$$\Delta\mathbf{P} = \begin{bmatrix} \mathbf{P}_1 - \mathbf{P}_0 \\ \mathbf{P}_2 - \mathbf{P}_1 \\ \mathbf{P}_3 - \mathbf{P}_2 \\ ... \\ \mathbf{P}_n - \mathbf{P}_{n-1} \end{bmatrix}$$

And has 1 row fewer than $\mathbf{P}$.

We also present a method for fitting a bezier curve to a set of points (in the least-squares sense). For the same time vector, $\mathbf{t}$, if given a matrix of points $\mathbf{L}$ sampled at the times in $\mathbf{t}$ such that:

$$\mathbf{L} = \begin{bmatrix} \mathbf{l}_0 \\ \mathbf{l}_1 \\ \mathbf{l}_2 \\ ... \\ \mathbf{l}_{N-2} \\ \mathbf{l}_{N-1} \\ \mathbf{l}_N \end{bmatrix}$$

I.e., each $\mathbf{l}_k \in \mathbb{R}^d$ is a point sampled at time $\mathbf{t}_k$. The task of least squares fitting becomes:

$$\mathbf{P}^* = \arg\min_{\mathbf{P}} \; \|\mathbf{A}(\mathbf{t})\mathbf{P} - \mathbf{L}\|$$

The solution to this least-squares problem can be framed in terms of the Singular Value Decomposition of $\mathbf{A}$.

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

$$\mathbf{P}^* = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T\mathbf{L}$$

We use Bézier curves as a parameterized representation of trajectories for a pure pursuit controller as well as a means of predicting trajectories for a racecar to follow in subsection 5.5.

5.3 Fully End-To-End Baseline

A large body of work in this domain focuses on fully end-to-end algorithms[Chi and Mu, 2017, Huston and Krapp, 2008, Katz et al.]. I.e. a network that is trained to map pixels in images directly to control outputs for the vehicle. In this paper, we consider two baseline algorithms in this domain:

1. NVIDIA's end-to-end PilotNet CNN [Bojarski et al., 2017]
2. A CNN-LSTM architecture that uses the output of a CNN as the input for a Long Short-Term Memory (LSTM) Cell.

Both of these architectures are trained fully end-to-end under a loss function of the mean-squared-error between the ground truth steering and throttle for a particular image and the predicted steering and throttle:

$$\mathbb{L}_{endtoend} = \frac{1}{2}[(\phi^* - \hat{\phi})^2 + (a^* - \hat{a})^2]$$

Where ϕ^* and $\hat{\phi}$ are the ground-truth and predicted steering angles; and a^* and $\hat{a}$ are the ground-truth and predicted throttle values for the car. Although it is technically possible to apply both throttle and brake at the same time, doing so can severely damage a racecar's drive train. For this work, we only consider throttle as a single scalar value in which positive indicates acceleration and negative indicates braking. We show in our experiments in section 6 that this approach to autonomous racing performs very poorly in a closed-loop sense.

5.4 AdmiralNet For Waypoint Prediction

We present a network architecture designed to map sequences of images (a context window) to sequence of wayppoints in the car's ambient task space (an intent window). The input context window of this model is a sequence of C color (RGB) images, each with height H and width W, such that each input tensor is $Cx3xHxW$ (the second dimension is 3 because each RGB image has 3 channels). For our experiments, we use $C = 5$, $H = 66$, and $W = 200$. It's output is a sequence of waypoints in the car's task space that are predicted to be optimal in some sense. For our experiments, we define "optimal" to be the trajectory followed by an expert example upon seeing that sequence of images. Additionally, we approximate the car's ambient task space as the euclidean plane, $\mathbb{R}^2$, of the car's axles. This architecture is named The AdmiralNet Waypoint Predictor.

Each image is passed through a Convolutional Neural Network (CNN) that maps each image to a "deep feature" vector. These C feature vectors are then used as the inputs to C recurrent calls to an LSTM with a hidden dimension h to build up the LSTM's hidden state with a learned encoding of the context window. We also extend Chi and Mu [2017]'s method of 3D "spatio-temporal convolution" by passing the same sequence of images through a 3D convolutional network. The output of this 3D

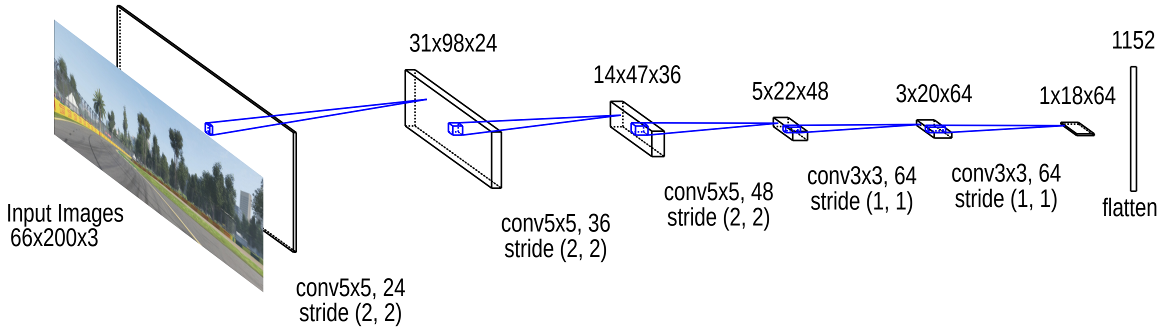


Figure 9: Our CNN that maps images to feature vectors. These feature vectors serve as a learned encoding of the context window. Although omitted from the graphic for clarity, batch normalization layers are also included after each convolutional layer. Image/feature map sizes are displayed as Height x Width x Channels

convolution is then used as the input for p additional recurrent calls to the LSTM. The resulting p outputs are then passed to a linear layer with input dimension h and output dimension 2. The outputs of this linear layer are taken as the sequence of predicted waypoints, $[\hat{\vec{y}}_1, \hat{\vec{y}}_2, \hat{\vec{y}}_3, ...\hat{\vec{y}}_p \in \mathbb{R}^2]$, each being a point in the car's local coordinate system, consisting of a lateral axis and a forward axis. To produce control outputs of steering and throttle, these points are then passed to a Pure Pursuit Controller described in Section 5.1. For our experiments, we train this network to minimize the average euclidean distance between the predicted waypoints and the ground-truth waypoints, $[\overset{*}{\vec{y}}_1, \overset{*}{\vec{y}}_2, \overset{*}{\vec{y}}_3, ..., \overset{*}{\vec{y}}_p \in \mathbb{R}^2]$, followed by a human example for that same sequence of images:

$$\mathbb{L}_{waypoint} = \sum_{i=0}^{p} \sqrt{\left\| \hat{\vec{y}}_i - \overset{*}{\vec{y}}_i \right\|}$$

To enable an ablative analysis of this 3D convolutional subnetwork, we also test this same model with fixed constants used as the input for the final p calls of the LSTM. These constants are learned parameters of the model and are optimized during the network training process. The results of this ablative analysis are contained in section 6.

5.5 AdmiralNet For Bézier Curve Prediction

Waypoint prediction also has it's limits. Derivative information cannot be implicitly encoded in a list of waypoints, they must be inferred numerically and are therefore not well suited to gradient back-propogation. Additionally, this approach suffers from the "curse of dimensionality". Each trajectory required Nxd parameters to represent, forcing the machine learned model to learn significantly more parameters to predict more dense trajectories.

To address these limitations, we present a novel approach for predicting future trajectories by using Bézier Curves as a dimensionality reduction technique. This approach is very similar to the approach in Section 5.4, but we train our model to predict the control points of a Bézier Curve instead of directly predicting waypoints. I.e., *we train a model to predict a parameterized representation of a curve rather than to learn samples from that curve.* This method addresses both limitations of the waypoint method. Derivatives of a Bézier Curve can be readily computed in closed-form. Additionally, the curse of dimensionality is mitigated by the fact that any number of points can be sampled from a Bézier Curve without the need to learn any additional parameters in a machine learned model. We employ the same methodology as in Section 5.4 to build up a temporal context for the LSTM and use some recurrent calls to project that context into an intent window. However, we now employ two key additional techniques:

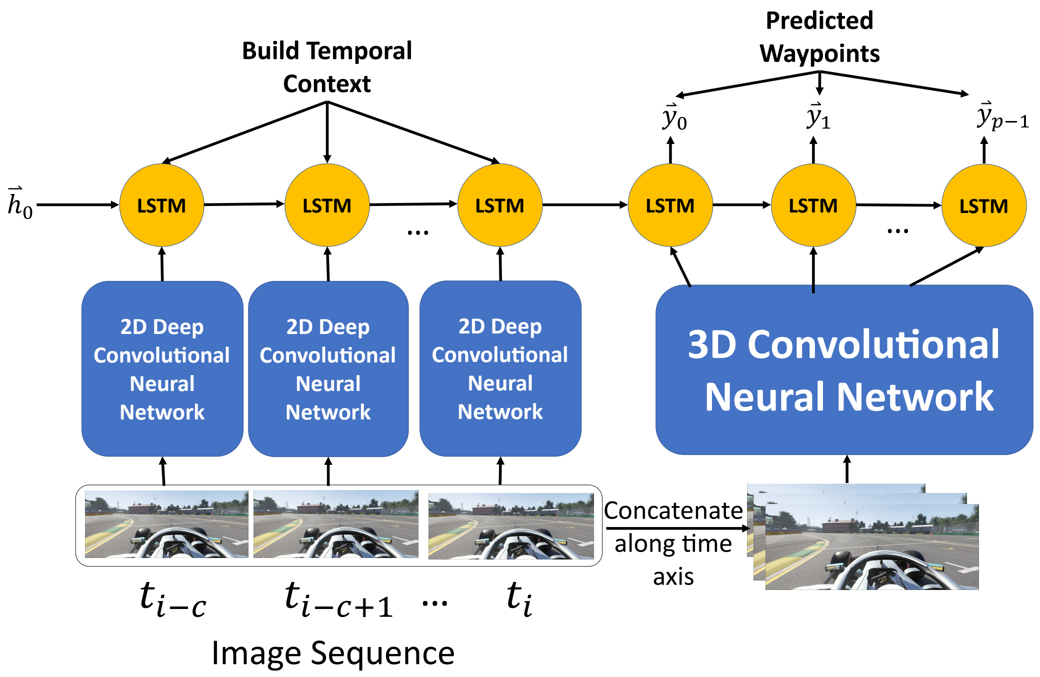


Figure 10: Architecture For The AdmiralNet Waypoint Predictor. The Deep CNN layers are the ones shown in Figure 9

1. The outputs of the p additional calls to the LSTM are concatenated along the time axis to form a 2D “feature grid”. This feature grid is then interpreted as an image and passed through another CNN

2. The output of this feature grid CNN is then linear mapped to predicted control points for a Bézier Curve rather than directly predicting waypoints

Just as for waypoint prediction, we use a Pure Pursuit controller to map a predicted trajectory to steering and throttle. However, we evaluate the predicted Bézier Curve on a fixed sample of the interval $[0, 1]$ to produce predicted waypoints rather than predicting the waypoints directly. For each pair of image sequences and ground truth waypoints, the network is trained to minimize a loss function that is a weighted sum of three terms:

1. The average squared norm between the predicted Bézier Curve control points and that of a least squared fit (described in section 5.2) to the ground truth trajectory points: What we call “control point loss”

2. The average euclidean distance between the predicted Bézier Curve evaluated on the interval $[0, 1]$ and the ground truth waypoints: What we call “Position Loss”

3. The average euclidean distance between the predicted Bézier Curve’s derivative (velocity) evaluated on the interval $[0, 1]$, but rescaled to the same time range as the ground truth labels, and the ground truth velocities: What we call “Velocity Loss”

One could also use sums instead of averages, but this only scales each loss term by a constant factor. The quantities described above are computed as follows for Bézier Curves of degree n, i.e. with $n + 1$ control points. N represents the number of ground truth position/velocity waypoints sampled for each image sequence. $\hat{\mathbf{P}}$ is the matrix of control points predicted by AdmiralNet, $\overset{*}{\mathbf{P}}$

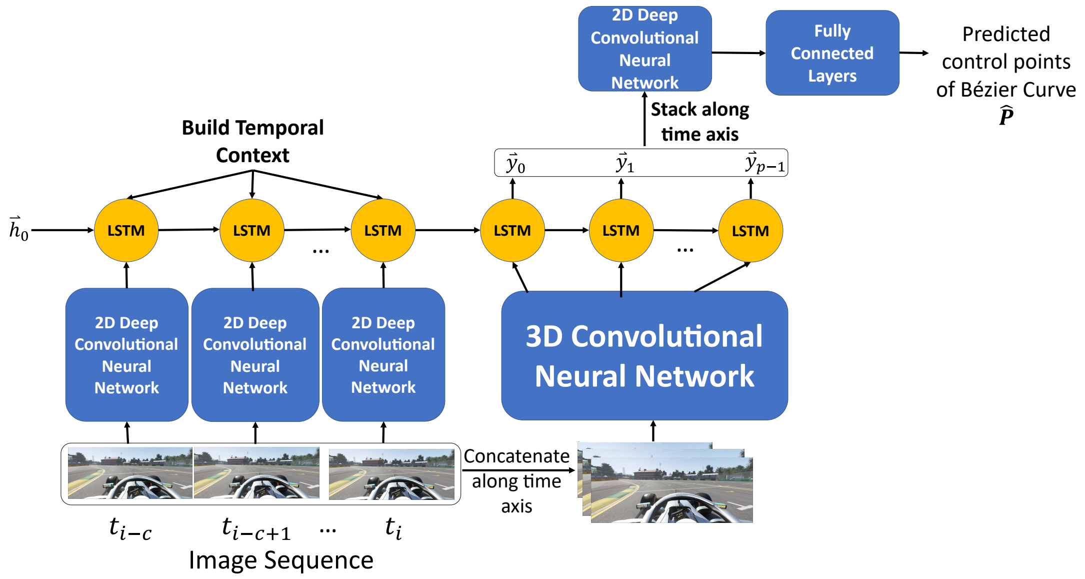

2D Deep Convolutional Neural Network
Fully Connected Layers
Predicted control points of Bézier Curve $\hat{\boldsymbol{P}}$
Stack along time axis
Build Temporal Context
$\vec{y}_0$
$\vec{y}_1$
$\vec{y}_{p-1}$
$\vec{h}_0$
LSTM
3D Convolutional Neural Network
Concatenate along time axis
t_{i-c}
t_{i-c+1}
t_i
Image Sequence

Figure 11: Architecture For The AdmiralNet Bézier Curve Predictor. The Deep CNN layers are the ones shown in Figure 9

is the control point matrix derived from a least squares fit (from section 5.2) to the ground truth waypoints, $\hat{\mathbf{Y}}$ is the sequence of predicted waypoints obtained by evaluating $\hat{\mathbf{P}}$ on $[0, 1]$.

$$\mathbf{t} = [t_0, t_1, t_2, ...t_{N-1}] \tag{3}$$

$$\Delta t = t_{N-1} - t_0 \tag{4}$$

$$\mathbf{s} = \frac{\mathbf{t} - t_0}{\Delta t} \subset [0, 1] \tag{5}$$

$$\hat{\mathbf{Y}} = [\hat{\vec{y}}_0, \hat{\vec{y}}_1, \hat{\vec{y}}_2, ..., \hat{\vec{y}}_{N-1}] = \mathbf{A}(\mathbf{s}, n)\hat{\mathbf{P}} \tag{6}$$

$$\frac{d\hat{\mathbf{Y}}}{d\mathbf{s}} = n\mathbf{A}(\mathbf{s}, n-1)\Delta\hat{\mathbf{P}} \tag{7}$$

$$\frac{d\hat{\mathbf{Y}}}{d\mathbf{t}} = [\frac{d\hat{\vec{y}}_0}{dt}, \frac{d\hat{\vec{y}}_1}{dt}, \frac{d\hat{\vec{y}}_2}{dt}, ..., \frac{d\hat{\vec{y}}_{N-1}}{dt}] = \frac{d\hat{\mathbf{Y}}}{d\mathbf{s}}\frac{d\mathbf{s}}{d\mathbf{t}} \tag{8}$$

$$\frac{d\mathbf{s}}{d\mathbf{t}} = \frac{1}{\Delta t} \tag{9}$$

$$\frac{d\hat{\mathbf{Y}}}{d\mathbf{t}} = \frac{d\hat{\mathbf{Y}}}{d\mathbf{s}}\frac{1}{\Delta t} \tag{10}$$

$$\mathbb{L}_{position} = \frac{1}{N}\sum_{i=0}^{N-1}\left\|\hat{\vec{y}}_i - \overset{*}{\vec{y}_i}\right\| \tag{11}$$

$$\mathbb{L}_{velocity} = \frac{1}{N}\sum_{i=0}^{N-1}\left\|\frac{d\hat{\vec{y}}_i}{dt} - \frac{d\overset{*}{\hat{\vec{y}}_i}}{dt}\right\| \tag{12}$$

$$\mathbb{L}_{control\_point} = \frac{1}{n-1}\sum_{i=0}^{n}\left\|\hat{\vec{p}_i} - \overset{*}{\vec{p}_i}\right\| \tag{13}$$

The overall loss function for the AdmiralNet Bézier Curve Predictor is a weighted sum of these three loss functions:

$$\mathbb{L} = w_{position}\mathbb{L}_{position} + w_{velocity}\mathbb{L}_{velocity} + w_{control\_point}\mathbb{L}_{control\_point} \tag{14}$$

Following the same procedure as with the Waypoint Predictor, we perform an abalative analysis of the additional 3D convolutional subnetwork by replacing it's output with learnable constants. The results of this ablative analysis are contained in section 6.

5.6 Comments on Implementation Details

Autonomous racing is a very high-speed task. As such, a very fast running time for any autonomous racing algorithm is desirable. Because evaluating a neural network can be a computationally expensive task, our approach uses a multi-threading approach to limit the impact of the computation required to evaluate our model. At runtime, a separate thread is spawned to maintain a circular buffer containing the C images that make up the input to both the waypoint predictor and the Bèzier Curve predictor. This thread listens for new images and adds them to the circular buffer, completely independent of the main control loop. The main control loop then grabs a snapshot of this buffer and runs the model to produce a control value. This approach guarantees that the sequence of images in the circular buffer are *always* contiguous in time, there is no risk of dropping an image because the main control loop is busy evaluating a neural network. Empirical measurements of the runtimes of each of these loops are explored in subsection 6.3.

Additionally, to generate a lookahead point for the Pure Pursuit controller, the Bèzier Curve predictor takes uniformly spaced samples from the predicted Bèzier Curve and selects a lookahead point from these samples.

To generate throttle commands, a bang-bang control approach is used. If the car's current velocity is slower than the reference velocity of the predicted Bèzier Curve at the lookahead point, the throttle is set to it's maximum value and brake is set to 0. Otherwise, the brake is set to it's maximum value and throttle is set to 0. The Waypoint Predictor uses a similar approach, but fits a spline to it's predicted waypoints and evaluates that spline's derivative to generate a reference velocity.

These details are specific to our experiments, the models presented are scalable to other strategies for transforming a predicted trajectory into steering and throttle commands.

We implement these models in the PyTorch [Paszke et al., 2019] framework. Additionally, an NVIDIA GeForce GTX 1080Ti was used to test each model. Each model was trained on a distributed cluster containing many NVIDIA GPUs.

6 Experimental Results

We now present several empirical results of this work.

1. An evaluation of how well our data-logging infrastructure is synchronized with the F1 game
2. Open-loop results of several neural network architectures trained on data from the game
3. A closed-loop driving evaluation of those same models
4. An evaluation of the running time of our approach

We show that our infrastructure infrastructure produces negligible error between logged values and the true internal state of the F1 game. Additionally we provide case studies of several deep learning models both in the open-loop (root-mean-square error) sense and in the closed-loop (true driving performance) sense.

6.1 Data Logging Infrastructure

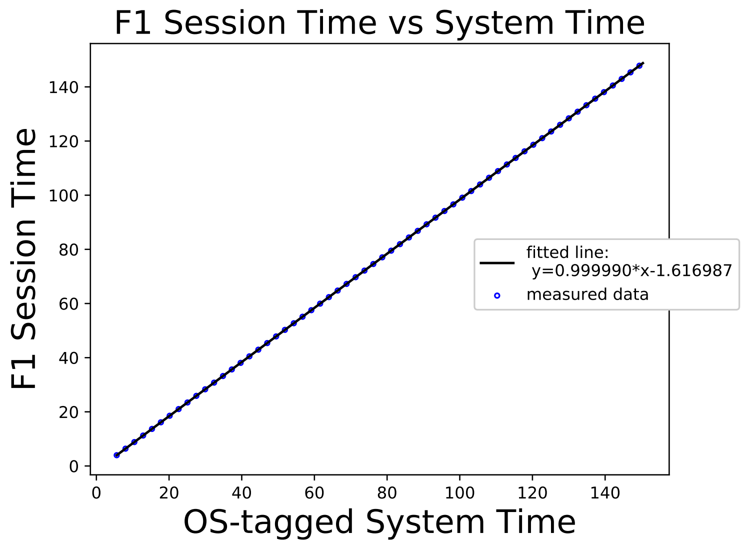


Figure 12: A plot of F1 session times versus measured system times. For readability, the data has been downsampled by a factor of 100. The slope of the regression line very close to 1.0 implies that the timing between the F1 game clock and the measurable system clock is consistent. The intercept of −1.616876 on the regression line indicates that the F1 session began broadcasting 1.616876 seconds before the datalogger was started.

We present an evaluation of our datalogging infrastructure by measuring the relative slope between the timescale on the measured OS clock and the received timestamps from the F1 game. In the ideal case, a slope of 1.0 would indicate that 1 second on our system clock corresponds to exactly 1 second on the games internal clock, indicating our data logging infrastructure is synchronized perfectly with the game. We evaluate our datalogging infrastructure by fitting a regression line to measurements of the system timestamp assigned to the UDP packets versus their stated F1 session time. Figure 12 shows a plot of such a regression line. The slope of this line is 0.99999, indicating that each each second of system time adds 1 microsecond of error in the corresponding session time. To put this in perspective, a data logging session that lasts 4 hours would only introduce 14 milliseconds of error. NumPy reported an r^2 value of exactly 1.0000, indicating that the error in this fit is so small, NumPy is numerically incapable of computing it. The intercept of −1.616876 on the fit line indicates that the measurements on the system clock began 1.616876 seconds after the F1 session began.

6.2 Neural Network Architectures

Each neural network architecture was trained for 100 epochs under it's corresponding loss function with a mini-batch size of 128. Stochastic Gradient Descent with a step size of 10^{-4} was used as the underlying optimization routine for network weight training. For the Bézier Curve predictor, we use weighting factors of $w_{position} = 1.0, w_{velocity} = 0.1, w_{control\_point} = 0.05$. Each model was then tested on the Australia circuit by having each model control a standard F1 car on 5 test laps around the track, the car was reset to the same starting position on each lap. To measure how effectively each model races a lap, we define a “boundary failure” (BF) to be when an autonomous agent veers outside the bounds of the track. We define a “Boundary Failure Score” (BFS) to be the average distance outside the track the autonomous agent went during a boundary failure. A boundary failure with a lower BFS is considered a better (“less bad”) failure.

6.2.1 Open-Loop RMSE Results

A traditional way to evaluate neural network models is with a classical “training set/validation set” approach where models are trained on a specified dataset and then tested on an unseen validation set under some open-loop metric. We show that such a metric does not necessarily correlate with real-time driving performance. We performed such an evaluation of the purely end-to-end models. The Root-Mean-Square-Error (RMSE) between predicted steering and ground-truth steering is used as an evaluation metric.

Model Configurations	**RMSE Steering**	**RMSE Throttle**
PilotNet	0.179	0.317
CNN-LSTM	0.184	0.302

Note that the CNN-LSTM setup actually performs *worse* than PilotNet in the open-loop sense in predicted steering angles and slightly better than PilotNet in predicting throttle values.

6.2.2 Closed Loop (Driving Performance) Results

To more appropriately evaluate these neural network architectures, we use a fully closed-loop approach. Rather than evaluating based only on an offline metric like Root-Mean-Square error. We utilize our DeepRacing framework’s ability to close the loop and send steering, throttle, and brake commands back into the Codemaster’s$^{\text{TM}}$ game to evaluate each network on it’s ability to actually race a simulated F1 car. Specifically, we trained each model on ~25000 images of training data from the Australia F1 circuit. For PilotNet and the CNN-LSTM architectures, each image is labeled with steering and acceleration. For both the Waypoint predictor and the Bézier Curve predictor, each image is labeled with 60 future waypoints from an expert driver.

For these experiments, a context length of $C = 5$ is used for both the waypoint predictor and the Bèzier Curve predictor. The waypoint predictor was configured to predict 20 timesteps into the future, corresponding to 1.4 seconds. The same timescale was used for the Bèzier Curve predictor.

We ran each model 5 times and recorded the following metrics:

1. Whether the model successfully completed a lap
2. Mean Lap time (if a lap was successfully completed)
3. Mean time between boundary failures (TBF)
4. Mean distance along the track between boundary failures (DBF)
5. Mean Number of Boundary Failures (NBF)
6. Mean Boundary Failure Score (BFS)

Additionally, we conduct an ablative analysis of the effect of using the 3D Convolutional layers to encode to compute the inputs for the final recurrent calls to the LSTM layer. For comparison, we evaluate both the waypoint predictor and the Bézier Curve predictor with constant values as the input to the final LSTM calls. These constant values are defined as model parameters and their values are learned during the training process. The results from these experiments are tabulated below. All figures are arithmetic means across all 5 runs. DNF indicates “did not finish” a successful lap.

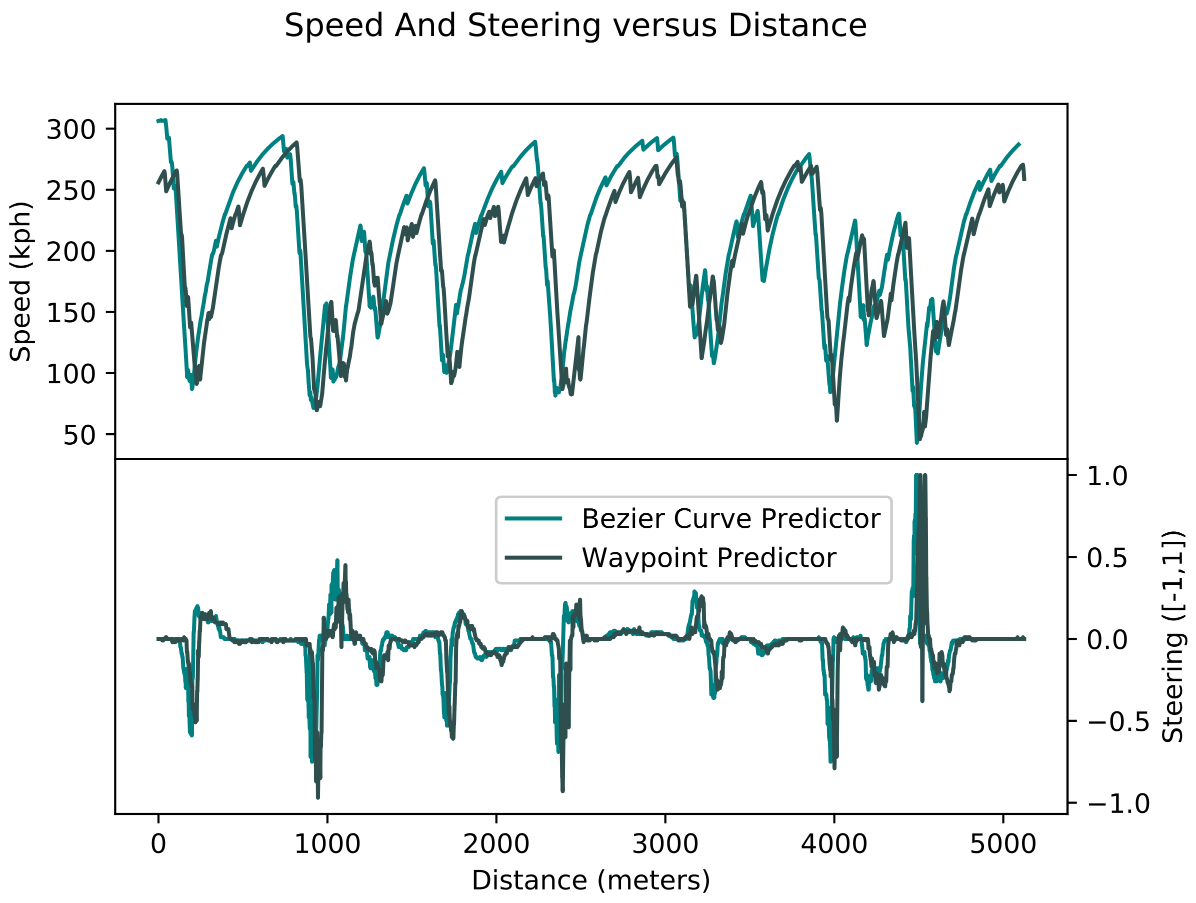


Figure 13: A Formula 1 style control plot for a test run. The Bézier Curve predictor produces much smoother velocity profiles

Note that the Bézier Curve predictor with the 3D convolution outperforms all 5 other models on all metrics, except for lap time. However, the modest improvement in lap time by removing the 3D convolutional subnetwork comes at the expense of significantly more boundary failures with a significantly larger boundary failure score, indicating that this improvement in lap time is really just a result of illegally "cutting corners". PilotNet was unable to complete a successful lap, as it crashed into a wall almost immediately (within 5 seconds on all 5 runs). The CNNLSTM architecture was also unable to complete a successful lap, but managed to make it around the first turn and down the initial straightaway. The Bézier Curve predictor also results in a smoother velocity curve than direct waypoint regression (see Figure 14).

Interestingly, the CNNLSTM architecture outperforms PilotNet in the closed loop sense (it was at least able to make the initial turn and most of the way down the initial straightaway) despite performing slightly worse than PilotNet under a purely open-loop evaluation.

Finally, we performed a longevity test of each of the trajectory-based methods. In this test, each model was deployed to a test run and allowed to continue until the vehicle either crashed or became inoperable. The results of this longevity test are in table 2.

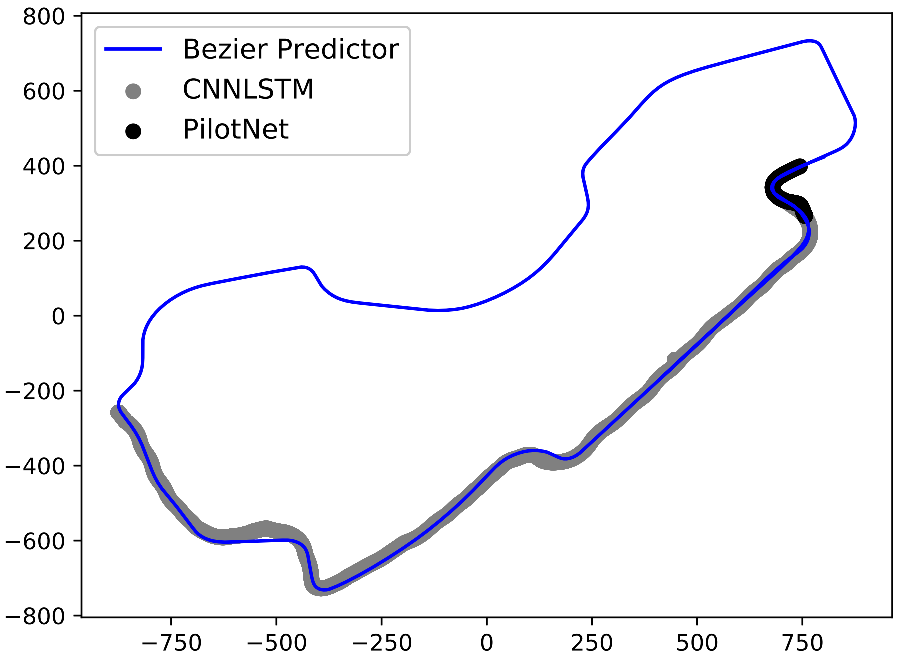


Figure 14: A plot of the path followed by our approach versus the two open-loop baselines. PilotNet fails almost immediately and the CNNLSTM only makes it about half-way around the track.

6.3 Computation Time Analysis

The main control loop for each approach consists of three high level steps:

1. Get a snapshot of the circular buffer containing the C most recently measured images
2. Evaluate the model on this snapshot
3. Generate steering/throttle commands from the model's output

We measured the running time of this control loop for each approach. Additionally, we measure the running time required to save a newly received image to the circular buffer. The results are tabulated in table 3.

The Bèzier Curve predictor is the slowest model, this is not surprising as it involves the most neural network layers. However, at $16.393Hz$, it is still sufficiently fast for autonomous racing.

Model Configuration	Lap Time (seconds)	TBF (seconds)	DBF (meters)	Number of Boundary Failures	BFS (meters)	Successful Laps
PilotNet	DNF	4.07	181.444	1.800	4.267	0
CNN-LSTM	DNF	6.367	304.490	3.600	2.539	0
Waypoint Predictor without 3D Convolution	113.38	11.92	626.23	6.6	7.02	1
Waypoint Predictor	106.683	16.739	855.817	5.6	0.239	4
Bézier Curve Predictor without 3D Convolution	**99.95**	19.01	1008.46	7.4	2.89	5
Bézier Curve Predictor	101.72	**33.62**	**1786.36**	**1.8**	**0.041**	**5**

Table 1: Results of our closed-loop testing. Note that the Bézier Curve Predictor outperforms all of the other models on (almost) all metrics. Also note that removing the 3d convolutional layers and replacing their outputs with learnable constants significantly degrades performance on both trajectory prediction models. "DNF" indicates the model did not finish a lap.

Model Configuration	Laps To Failure
Bzier Curve Predictor	106
Bzier Curve Predictor (No 3d Convolution)	86
Waypoint Predictor	11
Waypoint Predictor (No 3d Convolution)	1

Table 2: The results of out longevity testing. Both fully end-to-end models did not complete a single lap and are excluded from this table.

Model Configuration	Mean Runtime (seconds)	Mean Frequency (Hz)
Image Buffer	0.000707	1,414.42
PilotNet	0.008	125
CNNLSTM	0.011	90.90
Waypoint Predictor	0.040	25
Waypoint Predictor (No 3d Convolution)	0.0370	27.034
Bzier Curve Predictor	0.061	16.396
Bzier Curve Predictor (No 3d Convolution)	0.056	17.816

Table 3: The runtime of each model as well as the thread for maintaining the image buffer. The Bezier Curve Predictor is the slowest model, but is still sufficiently fast for an autonomous driving task. Also note that the image buffer thread operates an order of magnitude faster than any of the tested models, indicating the risk of dropping an image due to computational overhead is very low.

Additionally, note that the separate thread maintaining the image buffer runs at over $1414Hz$, several orders of magnitude faster than the image capture rate of our infrastructure, indicating the risk that this approach will drop images and produce discontinuous image sequences is negligible.

7 Conclusion and Future Work

7.1 Discussion

In this paper, we address the problem of learning expert agile driving behavior from demonstration. We consider motor-sport racing as the proxy for agile driving since it involved high speed driving at the limits of control, and often in close proximity of other vehicles. This hypothesis leads us to the problem of learning how to autonomous race in a realistic motor-sport racing environment. One of our primary contribution is DeepRacing - a novel end-to-end framework, and a virtual testbed for training and evaluating algorithms for the hard challenge of autonomous racing. The virtual testbed is implemented using a highly realistic and professional Formula One gaming environment. Our open-source DeepRacing framework, which is integrated with ROS2, will enable researchers to explore the limits of vision based end-to-end autonomous racing.

In addition, we also develop and present a new parametrized trajectory-based end-to-end trainable network for autonomous racing. Our novel AdmiralNet architecture, which uses Bézier Curves to parameterize a lower-dimensional manifold within the task space of the ego vehicle, outperforms traditional end-to-end networks by a very large margin and outperforms waypoint-based planning by ~ 5 seconds in terms of lap time and by over 100% in terms of time between failures, all while being robust and computationally tractable.

7.2 Future Work

The results and the work so far has focused on autonomous racing in a time-trial manner, i.e. only one vehicle on the track at a time. Our ongoing and future work include a more focused study of head-to-head racing (multi-agent setting) with a stronger focus on real-world racing metrics like lap time and overall race position. We also intend to explore and compare our method with reinforcement learning based approaches for this problem. Additionally, this work does not address how the agile behavior learned in a racing environment can make day-to-day driving safer. This challenging question of transferring agile behavior between domains is the subject of our future research in this area.

References

Global championship of driverless cars. url=https://roborace.com/, journal=Roborace.

Formula one esports series. *F1 Esports*, Nov. 2017. `https://f1esports.com/`.

Max verstappen trains using sim racing and shows incredible speed in both. *Redline UK*, 2017. `http://www.teamredline.co.uk/work/max-verstappen/`.

Tesla car that crashed and killed driver was running on autopilot, firm says. *The Guardian*, Mar 2018. `https://www.theguardian.com/technology/2018/mar/31/tesla-car-crash-autopilot-mountain-view`.

F. Altch and A. de La Fortelle. An lstm network for highway trajectory prediction. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 353–359, Oct 2017. doi: 10.1109/ITSC.2017.8317913.

S. J. Anderson, S. C. Peters, T. E. Pilutti, and K. Iagnemma. An optimal-control-based framework for trajectory planning, threat assessment, and semi-autonomous control of passenger vehicles in hazard avoidance scenarios. *International Journal of Vehicle Autonomous Systems*, 8(2-4): 190–216, 2010.

M. Bansal, A. Krizhevsky, and A. S. Ogale. Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst. *CoRR*, abs/1812.03079, 2018. URL `http://arxiv.org/abs/1812.03079`.

A. Best, S. Narang, D. Barber, and D. Manocha. Autonovi: Autonomous vehicle planning with dynamic maneuvers and traffic constraints. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2629–2636. IEEE, 2017.

M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. D. Jackel, and U. Muller. Explaining how a deep neural network trained with end-to-end learning steers a car. *CoRR*, abs/1704.07911, 2017. URL `http://arxiv.org/abs/1704.07911`.

A. Brown et al. Udacity self-driving car simulator. In *GitHub Repository*. 2018.

M. Buehler, K. Iagnemma, and S. Singh. *The DARPA urban challenge: autonomous vehicles in city traffic*, volume 56. springer, 2009.

C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

D. Chen, B. Zhou, V. Koltun, and P. Krhenbhl. Learning by cheating, 2019.

L. Chi and Y. Mu. Learning end-to-end autonomous steering model from spatial and temporal visual cues. In *Proceedings of the Workshop on Visual Analysis in Smart and Connected Communities*, VSCC '17, pages 9–16, NY, USA, 2017. ACM. ISBN 978-1-4503-5506-3. doi: 10.1145/3132734.3132737. URL `http://doi.acm.org/10.1145/3132734.3132737`.

Codemasters. F1 2017 d-box and udp output specification. `http://forums.codemasters.com/discussion/53139/f1-2017-d-box-and-udp-output-specification`.

A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

S. Eizikovich. vjoy. `http://vjoystick.sourceforge.net/site/`.

H. M. Eraqi, M. N. Moustafa, and J. Honer. End-to-end deep learning for steering autonomous vehicles considering temporal dependencies. *CoRR*, abs/1710.03804, 2017. URL `http://arxiv.org/abs/1710.03804`.

T. Fernando, S. Denman, S. Sridharan, and C. Fookes. Going deeper: Autonomous steering with neural memory networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 214–221, Hawaii Convention Center HI, 2017. URL `https://eprints.qut.edu.au/114117/`.

J. L. Foo, J. Knutzon, V. Kalivarapu, J. Oliver, and E. Winer. Path planning of unmanned aerial vehicles using b-splines and particle swarm optimization. *Journal of aerospace computing, Information, and communication*, 6(4):271–290, 2009.

F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. 1999.

S. J. Huston and H. G. Krapp. Visuomotor transformation in the fly gaze stabilization system. *PLoS biology*, 6(7):e173, 2008.

J. Janai, F. Güney, A. Behl, and A. Geiger. Computer vision for autonomous vehicles: Problems, datasets and state-of-the-art. *CoRR*, abs/1704.05519, 2017. URL `http://arxiv.org/abs/1704.05519`.

M. Johnson-Roberson, C. Barto, R. Mehta, S. N. Sridhar, K. Rosaen, and R. Vasudevan. Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks? *arXiv preprint arXiv:1610.01983*, 2016.

C. Katrakazas, M. Quddus, W.-H. Chen, and L. Deka. Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions. *Transportation Research Part C: Emerging Technologies*, 60:416–442, 2015.

G. Katz, A. Roushan, and A. Shenoi. Supervised learning for autonomous driving.

B. Kim, C. M. Kang, J. Kim, S. H. Lee, C. C. Chung, and J. W. Choi. Probabilistic vehicle trajectory prediction over occupancy grid map via recurrent neural network. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 399–404, Oct 2017. doi: 10.1109/ITSC.2017.8317943.

S. Koppula. Learning a cnn-based end-to-end controller for a formula sae racecar. *arXiv preprint arXiv:1708.02215*, 2017.

F.-L. Lian and R. Murray. Real-time trajectory generation for the cooperative path planning of multi-vehicle systems. In *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, volume 4, pages 3766–3769. IEEE, 2002.

M. Likhachev and D. Ferguson. Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research*, 28(8):933–945, 2009.

A. C. Madrigal. Inside waymos secret world for training self-driving cars. *The Atlantic*, 23, 2017.

A. I. Maqueda, A. Loquercio, G. Gallego, N. N. García, and D. Scaramuzza. Event-based vision meets deep learning on steering prediction for self-driving cars. *CoRR*, abs/1804.01310, 2018. URL `http://arxiv.org/abs/1804.01310`.

J. Michels, A. Saxena, and A. Y. Ng. High speed obstacle avoidance using monocular vision and reinforcement learning. In *Proceedings of the 22nd International Conference on Machine Learning*, ICML 05, page 593600, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595931805. doi: 10.1145/1102351.1102426. URL `https://doi.org/10.1145/1102351.1102426`.

E. Mohammed, M. Abdou, and O. Ahmed Nasr. End-to-end deep path planning and automatic emergency braking camera cocoon-based solution. In *Machine Learning for Autonomous Driving, NeurIPS 2019 Workshop*, December 2019.

M. E. Mortenson. *Mathematics for Computer Graphics Applications: An Introduction to the Mathematics and Geometry of CAD/Cam, Geometric Modeling, Scientific Visualizati*. Industrial Press, Inc., USA, 2nd edition, 1999. ISBN 083113111X.

W. Naeem, R. Sutton, and S. Ahmad. Pure pursuit guidance and model predictive control of an autonomous underwater vehicle for cable/pipeline tracking. In *Proceedings-Institute of Marine Engineering Science and Technology Part C Journal of Marine Science and Environment*, pages 25–35. THE INSTITUTE OF MARINE ENGINEERING, SCIENCE AND TECHNOLOGY, 2004.

X. Pan, Y. You, Z. Wang, and C. Lu. Virtual to real reinforcement learning for autonomous driving, 2017.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

S. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. Eng, D. Rus, and M. Ang. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5(1):6, 2017.

E. Perot, M. Jaritz, M. Toromanoff, and R. De Charette. End-to-end driving in a realistic racing game with deep reinforcement learning. In *International conference on Computer Vision and Pattern Recognition-Workshop*, 2017.

S. R. Richter, V. Vineet, S. Roth, and V. Koltun. Playing for data: Ground truth from computer games. *CoRR*, abs/1608.02192, 2016a. URL `http://arxiv.org/abs/1608.02192`.

S. R. Richter, V. Vineet, S. Roth, and V. Koltun. Playing for data: Ground truth from computer games. In *European Conference on Computer Vision*, pages 102–118. Springer, 2016b.

M. Samuel, M. Hussein, and M. B. Mohamad. A review of some pure-pursuit based path tracking techniques for control of autonomous vehicle. *International Journal of Computer Applications*, 135(1):35–38, 2016.

E. Santana and G. Hotz. Learning a driving simulator. *CoRR*, abs/1608.01230, 2016. URL `http://arxiv.org/abs/1608.01230`.

W. Scacchi. Autonomous emotorsports racing games: Emerging practices as speculative fictions. *Journal of Gaming & Virtual Worlds*, 10(3):261–285, 2018.

S. Shah, D. Dey, C. Lovett, and A. Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017. URL `https://arxiv.org/abs/1705.05065`.

D. Wakabayashi. Self-driving uber car kills pedestrian in arizona, where robots roam. *New York Times.* `https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html`.

R. S. Wallace, A. Stentz, C. E. Thorpe, H. P. Moravec, W. Whittaker, and T. Kanade. First results in robot road-following. In *IJCAI*, pages 1089–1095. Citeseer, 1985.

T. Weiss and M. Behl. Deepracing: A framework for agile autonomy. *Design, Automation and Test in Europe Conference*, 2020. URL `https://past.date-conference.com/proceedings-archive/2020/html/1027.html`.

B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4:6, 2000.

H. Xu, Y. Gao, F. Yu, and T. Darrell. End-to-end learning of driving models from large-scale video datasets. *CoRR*, abs/1612.01079, 2016a. URL `http://arxiv.org/abs/1612.01079`.

H. Xu, Y. Gao, F. Yu, and T. Darrell. End-to-end learning of driving models from large-scale video datasets. *CoRR*, abs/1612.01079, 2016b. URL `http://arxiv.org/abs/1612.01079`.