Dynamic-Weighted Simplex Strategy for Learning Enabled Cyber Physical Systems

Shreyas Ramakrishna, Charles Harstell, Matthew P Burruss, Gabor Karsai, Abhishek Dubey

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA

Abstract

Cyber Physical Systems (CPS) have increasingly started using Learning Enabled Components (LECs) for performing perception-based control tasks. The simple design approach, and their capability to continuously learn has led to their widespread use in different autonomous applications. Despite their simplicity and impressive capabilities, these models are difficult to assure, which makes their use challenging. The problem of assuring CPS with untrusted controllers has been achieved using the Simplex Architecture. This architecture integrates the system to be assured with a safe controller and provides a decision logic to switch between the decisions of these controllers. However, the key challenges in using the Simplex Architecture are: (1) designing an effective decision logic, and (2) sudden transitions between controller decisions lead to inconsistent system performance. To address these research challenges, we make three key contributions: (1) *dynamic-weighted simplex strategy* – we introduce "weighted simplex strategy" as the weighted ensemble extension of the classical Simplex Architecture. We then provide a reinforcement learning based mechanism to find dynamic ensemble weights, (2) *middleware framework* – we design a framework that allows the use of the dynamic-weighted simplex strategy, and provides a resource manager to monitor the computational resources, and (3) *hardware testbed* – we design a remote-controlled car testbed called DeepNNCar to test and demonstrate the aforementioned key concepts. Using the hardware, we show that the dynamic-weighted simplex strategy has 60% fewer out-of-track occurrences (soft constraint violations), while demonstrating higher optimized speed (performance) of 0.4 m/s during indoor driving than the original LEC driven system.

Keywords: Convolutional Neural Networks, Learning Enabled Components, Reinforcement Learning, Simplex Architecture.

Abbreviations

CPS Cyber Physical Systems

CNN Convolutional Neural Network

DNN Deep Neural Network

e2e end-to-end

LD Lane Detection

LEC Learning Enabled Component

RL Reinforcement Learning

1. Introduction

Cyber Physical Systems (CPS) have increasingly started relying on the use of Learning Enabled Components (LECs) as part of the control loop, performing varied perception-based autonomy tasks. These data-driven components are trained using machine learning approaches like deep learning [1], and reinforcement learning (RL) [2]. These approaches have given these components the capability to continuously learn and work in unfamiliar environments. Recently, these components are seeing widespread acceptance and use in various autonomous applications like NVIDIA's DAVE-II [3] convolutional neural network (CNN) that performs end-to-end (e2e) learning-based self-driving, and Tesla's autonomous driving that has recently completed 2 billion driving miles [4] using several LECs to perform object detection and tracking, and image segmentation [5].

Despite their impressive capabilities, it is still a challenge to assure the correctness of these systems under all circumstances. Applying conventional assurance based development [6] or designing safety and assurance cases [7] is complicated for these systems because of the following limitations: First, the black-box nature of these models limits testing coverage. Pei, Kexin, et al. [8] discuss the limitations of performing exhaustive testing of these models, and introduce a whitebox framework called DeepXplore that can perform limited testing. Secondly, the existing verification tools are limited by the complexity of the neural network and non-linearity of the activation functions. The authors in [9] discuss the limitations of performing verification techniques on CNNs. Third, these components learn from data, and they perform well only when the test observations resemble the training data. The authors in [8], [10] have shown how subtle changes (or adversaries) in the test image confuse the model and results in erroneous predictions. These limitations make it challenging to design safety or assurance cases for these systems. Recently, the DARPA Assured Autonomy project [11] has been focusing on designing tools to overcome these limitations.

The problem of assuring safety guarantees in CPS has been tackled using the Simplex Architecture [12]. This architec-

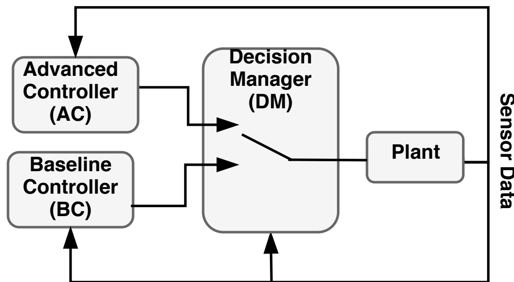

Figure 1: The Simplex Architecture adopted from [15] combines an unverified advanced controller (AC) with a safe baseline controller (BC) and a decision manager (DM). The DM is responsible for selecting the controller that can maintain the system safe.

ture (see Fig. 1) improves the system's safety by combining an unverified high-performance controller (Advanced Controller (AC)) with a safety controller (Baseline Controller (BC)) and a decision manager (DM). The DM is the core component in the Simplex Architecture, which arbitrates the control between the two controllers based on a safety criteria. This architecture requires verifying only the BC and the DM, thus alleviating the requirement of verifying the AC (e.g. LEC), which is sometimes difficult or not possible. Simplex Architectures have been used in several applications: the authors in [13] have demonstrated the utility of the Simplex Architecture to avoid a collision in a fleet of remote-controlled cars. [14] discusses a case study of using the Simplex Architecture for the automatic landing of a F-16 aircraft. These applications demonstrate the use of Simplex Architecture for complex and safety-critical CPS applications.

However, the key challenges in using the Simplex Architectures for CPS applications are: (1) designing effective decision logic: as discussed in [16], the critical challenge is the design of appropriate decision logic. It is trivial to design safe logic by always using the BC to make the decisions. However, that would make the system too defensive without utilizing the high performance AC. Therefore, the the architecture should seek to utilize the AC as much as possible to avoid conservatively using the BC. That is, it is important to design decision logic that balances the safety and performance of the system. Key challenge (2) sudden transitions: the instantaneous transitions (see Fig. 3) from the AC to the BC drastically degrades the system performance. For example, Bak, Stanley, et al.[15] have discussed the applicability of the Simplex Architecture in heart pacemakers. The authors discuss how sudden transitions between simplex controllers are dangerous in safety critical applications. For a pacemaker case study, they illustrate how sudden jumps between two discrete heart rates (65 to 120 beats per minute) from two simplex controllers can make the patients dizzy and uncomfortable. Similarly, in automotive applications the sudden changes in the control (e.g. speed) will be perceived as jerks (high rate of change of acceleration) and can be uncomfortable for passengers (e.g. Toyota sudden acceleration problem [17]).

Contributions: To address these challenges we perform a weighted blending of the simplex control actions. We refer to this weighted extension of the conventional Simplex Architecture as the "weighted simplex strategy". Instead of selecting a single controller action, this strategy computes a weighted ensemble of all the controllers actions. Such blending mechanisms have shown to improve performance or accuracy in model ensembles [18], and multiple model adaptive predictive control [19]. For the Simplex Architecture, we hypothesize blending the controllers actions could optimally balance the performance and soft constraint violations of the system while avoiding abrupt transitions. Specifically, this strategy aims at optimizing the performance while reducing the constraint violations. So, it can only be used for systems which can tolerate soft constraint violations (ie. constraints that can be violated, but will incur a penalty). We summarize our contributions below.

Table 1: Notation Lookup Map

Symbol	Description
θ_L	Steering PWM value of DeepNNCar using LEC controller
θ_C	Steering PWM value of DeepNNCar using OpenCV controller
θ_D	Steering PWM value using dynamic-weighted simplex strategy
θ_F	Steering PWM value using fixed-weighted simplex strategy
W_L	Ensemble weight given to LEC controller
W_C	Ensemble weight given to OpenCV controller
W_{SET}	Ensemble weights $\{W_L, W_C\}$ computed by dynamic-weighted simplex strategy
T_R	Inference pipeline time of DeepNNCar
v_t	Current speed PWM value of DeepNNCar

- Dynamic-Weighted Simplex Strategy – We discuss a mechanism to find dynamic ensemble weights of the weighted simplex strategy using reinforcement learning (RL). We show the design of the reward function that is responsible for reducing the soft constraint violations while improving the performance of the system.
- Middleware Framework – We design a middleware framework to deploy the weighted simplex strategy on a physical CPS platform called DeepNNCar. In addition, the framework also has a resource manager that is used to monitor the computational resource of the system while mitigating any overload introduced by the complex computations of the proposed strategy.
- Hardware Testbed – We discuss the design of a resource constrained remote controlled car, called the DeepNNCar, that will be used as a case study to test the dynamic weighted blending mechanism and the middleware framework.

Outline: The outline of the this paper is as follows: in Section 2, we discuss the background topics. In Section 3, we present related research. Section 4 describes our DeepNNCar testbed and its control algorithms. Section 5 introduces the dynamic-weighted simplex strategy and the setup to compute dynamic weights. In Section 6, we design the resource manager for managing the systems resource utilization. Section 7 describes the system integration. Section 8 evaluates the dynamic-weighted simplex strategy and the resource manager. Section 9 presents our conclusions. The notations used in the paper are described in Table 1.

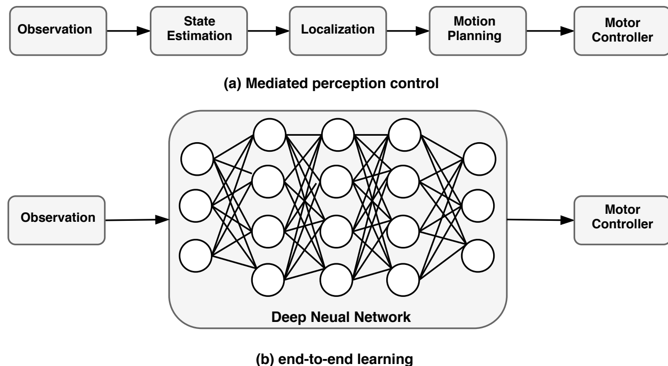

Figure 2: Perception based control approaches in robotics. Adopted from [20].

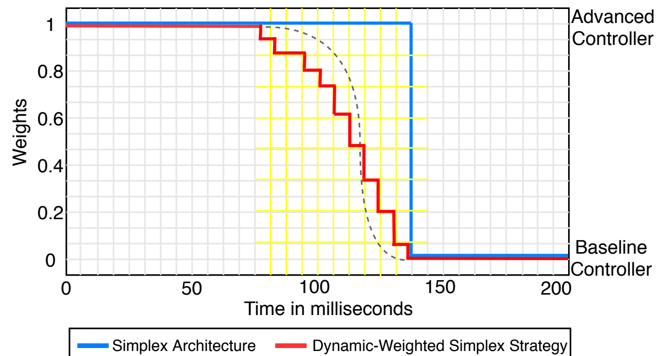

Figure 3: The weighted simplex strategy works alike the conventional simplex architecture in the high performance and safe regions. However, the targeted spectrum of this strategy is the transition region represented by yellow grid, where the unverified controller actions start drifting the system towards the unsafe state. Here, instead of instantaneous transition, the dynamic weights of the weighted simplex strategy provide a mechanism for a smoother transition between the controller outputs. The discreet weight adjustment in the dynamic-weighted simplex strategy (red line) takes a staircase function approximating a smoother curve (dotted gray line).

2. Background

In this section we discuss a few key concepts that are required to understand our methodology discussed in the later sections of this paper. Readers who are familiar can skip this section.

2.1. LEC Based Control Approaches in Autonomous Systems

Perception based control paradigms in autonomous vehicles can be classified into the mediated perception approach, and end-to-end (e2e) learning approach as shown in Fig. 2.

The mediated perception approach [21], [22] decomposes the problem into multiple sub-goals which then together form a processing pipeline for performing autonomous driving. The pipeline (see Fig. 2) has multiple stages of operation like sensing, state estimation, localization, motion planning and motor control, which use sensor data to learn high level representations of lanes, objects, cars, and traffic lights. This high-level information is then used by the controller to compute the low level actuation of the car.

The advantages of using this approach (as discussed in [23]) are: (1) transparent internal modes of operation for testing and debugging, (2) robust decision-making capabilities due to multiple dedicated algorithms, and (3) a high degree of freedom for the designer to select and fine tune the internal stage algorithms. However, multiple stages of operation along with the requirement to create and maintain a large code base even for simple navigation tasks makes its application on small-scale systems challenging.

In contrast, e2e learning [24] is a perception-based control approach that uses supervised learning [25] to directly compute the control action. It has been applied to different indoor and outdoor navigation tasks, such as obstacle avoidance [26], off-road autonomous navigation systems [27], and autonomous driving [3],[28]. In this work, we use an e2e learning approach for navigating a DeepNNCar (explained in Section 4) around an indoor track. In this approach, a data-driven model (e.g. DNN or CNN) takes in observations from different sensors such as camera, IMU, LIDAR, etc. and predicts an output control action that could either be the steering or speed of the system. Unlike traditional software where the execution logic is derived from analytical models, DNNs learn the underlying relationship between the observed input and the output from the data. The functionality of DNNs can be tuned with training hyperparameters like learning rates, epochs, optimization algorithms, etc. This approach has recently gained considerable attention because of its conceptual simplicity and computational efficiency.

Problems: Some limitations of the e2e learning approach are: (1) the direct sensor-actuation mapping is only limited to perform point based predictions, which can limit the use of the approach in complex navigation based tasks, (2) LECs used in mapping the sensor to actuation values learn from training data. So, even slight distribution shift in the test observations will result in inconsistent behavior in these models, and (3) training the LEC is hard as there are several hyperparameters to be tuned. Improper tuning of these parameters can lead to underfitting or overfitting problems.

2.2. Simplex Architecture

As discussed by Seto, et al. [12], the Simplex Architecture is built on the concept of analytic redundancy, i.e. redundant components with different design and implementations, but similar interfaces. Fig. 1 illustrates the structure of the Simplex Architecture, where the analytic redundancy is achieved using two controllers with same interfaces but with different implementation and specifications. An advanced controller (AC) with high-performance specification is combined with a reliable and safe baseline controller (BC), and a decision manager (DM). The decision logic of the DM is encoded to guarantee the safety of the system, i.e. the logic will transfer the control from the AC to the BC, when the AC decisions start moving the system toward unsafe states. The simplex combination of controller outputs can be represented as the weighted combination of the two controller output and can be written as in Eq. (1):

$$C_{SA} = W_1 \cdot C_{AC} + W_2 \cdot C_{BC} \tag{1}$$

Where, C_{SA} is the system's control output, C_{AC}, C_{BC} represents the controller outputs of the AC and the BC respectively. W_1, W_2 represents the weights and they sum up to 1. For the

conventional Simplex Architecture, W_1=1 and W_2=0 in most scenarios, but when the AC is on the verge of jeopardizing the safety of the system, the control transfers to the BC and the weights become W_1=0 and W_2=1. The blue line in the Fig. 3 shows the control transition between the two controllers of the classical Simplex Architecture. As seen, the control remains with the AC up to a certain point, and when the DM decides to switch there is a sudden (or instantaneous) transition of the control to the BC.

Problems: As mentioned in Section 1 the key challenge of using the Simplex Architectures are: (1) designing an effective decision logic that can optimally balance the safety and performance of the system, and (2) avoiding sudden transitions in the control between the AC to BC. Such sudden control transitions may result in inconsistent system performance in some domain specific applications like pacemakers [15] which are sensitive to minute changes in control actions. These challenges make it difficult to directly apply Simplex Architectures to certain sensitive domains in CPS applications.

We solve these challenges by introducing the dynamic-weighted simplex strategy, which is the weighted extension of the classical Simplex Architecture (Fig. 3). We also introduce a mechanism to find the dynamic weights using a RL approach.

3. Related Work

This section provides an overview of the existing literature work in the dimensions of decision logic for Simplex Architecture, middleware framework for small scale CPS platforms, and autonomous robot testbeds. A complete survey of papers in these areas is beyond the scope of this paper.

3.1. Decision Logic for Simplex Architectures

The existing literature on designing decision logic for the Simplex Architecture can be classified into two categories:

- Lyapunov function-based techniques (linear matrix inequality (LMI) [29]) have been used widely as the switching criteria for continuous systems. LMI is based on designing a verifiable decision logic by solving linear matrix inequalities. The authors in [30] have used LMI tools to design the decision logic for inverted pendulum control.
- Reachability analysis [31] is another technique used popularly in hybrid systems. In this technique the systems are modeled as hybrid automata, and a set of reachable states by the system is computed. A comprehensive discussion about reachable state computations is made in [31], [32] .

In addition, there are a few other techniques used to design decision logic for the Simplex Architecture. The authors in [16] have designed a switching criterion called Real-Time reachability algorithm. The algorithm proposes a unified approach that uses the offline LMI results along with online reachability analysis to design an effective decision logic. In [33], the authors have designed a simple application specific decision logic by designing a set of conditions under which the AC fails and when the controller transition must be made. This decision logic has been used in the Simplex Architecture to control an unmanned aerial vehicle (UAV). Phan, Dung et al. [34] have proposed the use of Assume-Guarantee (A-G) contracts to construct the decision logic for Simplex Architectures. A-G is a powerful reasoning, proof-based technique that show the guarantees that the system can provide when a set of assumptions are satisfied. They demonstrate their approach on a quick bot rover.

All these discussed techniques are based on either traditional control theory, reachability analysis or pre-defined rules. The rule-based technique does not scale well across applications and would require us to write new rules for different applications. The reachability-based techniques would be too slow to implement at runtime if the state space is large. Also, the reachability analysis for LECs using DNNs is still limited to feed forward ReLU based networks [35] and gets complex for CNNs.

In this work, we use an RL algorithm as the decision logic of the simplex combination. For this we design a reward function that involves information about the performance and the constraint (safety) of the system. The motivation for using RL are as follows: (1) the inability of the traditional verification techniques to verify complex CNNs, (2) the reward based system of RL that can continuously learn to choose safe actions in dynamic environments, and (3) an RL algorithm takes very less time (typically fractions of millisecond, depending on the complexity of state space) to compute a decision, as compared to the traditional verification techniques. Such short time decision computation is critical at runtime for CPS testbeds.

3.2. Middleware Framework for Small Scale Robots

There are several middleware frameworks designed for managing the functionalities of small-scale robots. Seiger, Ronny, et al [36] have designed a Robot Operating System (ROS) [37] based high-level programming framework to control the functionalities of domestic robots. ROS commands are used through the framework to control the robot in manual, semi-autonomous and fully-autonomous modes. The authors in [38] have introduced a middleware platform for distributed applications involving robots, sensors and the cloud. The framework uses AIOLOS [39], which is a distributed software framework that allows the developer to design software on multiple components (sensors, robots) without having them to design the inter-component communication. This framework seeks to make the software design process easier and distributed to developers. The F1/10 platform [40] is a small-scale remote-controlled race car that uses a ROS based communication framework to control the steer and speed actuations of the car.

In [41], the authors have discussed an architecture for intelligent manufacturing units in shop floors. The key responsibility of their middleware layer is to manage devices, define interfaces and data management. The key focus of the middleware layer is to relay the data collected from the robotic sensors to the server for analysis. [42] discusses a multi-agent cloud-based framework for management of collaborative robots. The sensor data collected by a swarm of robots is sent to a cloud-based server over the internet. The server then computes the actions that need to be taken by the robots.

All these frameworks aim at providing a middleware layer to control the actuations of the system. However, the key contribution of our middleware framework is the resource management strategy. We hypothesize an optimal resource management strategy is required to improve the performance of CPS applications.

3.3. Autonomous Robot Testbeds

There have been several ongoing research projects related to physical testbeds for autonomous systems. MIT's RaceCar [43], and University of Pennsylvania's F1/10 [40] have become popular autonomous racing platforms built on Traxxas 1/10 scale remote-controlled car with an NVIDIA's Jetson TX1 onboard computational unit. These cars use cameras, IMUs, and expensive LIDAR ($1,775) systems for performing simultaneous localization and mapping (SLAM), whereas DeepNNCar performs e2e learning using data from a limited array of sensors (camera, IR-Optocoupler, LIDAR ($100)). The cost comparison for F1/10 and DeepNNCar platforms is $3000 vs. $518.

DeepPicar [44] is another remote-controlled car platform that uses a smaller 1/24 scale chassis. Like DeepNNCar, this platform also uses a Raspberry Pi3 (RPi3) as the computational unit and performs e2e learning based autonomous driving using NVIDIA's DAVE-II CNN. This platform is relatively inexpensive ($70) but has a considerably smaller chassis and uses discrete steering actuation unlike DeepNNCar which performs continuous steering. Donkey car [45] is an open source autonomous car platform for small-scale remote-controlled cars. These cars are built on a 1/16 or 1/10 scale chassis, and use RPi3 as the computational unit along with a wide angle RPi camera that is the primary sensor. Like the other e2e learning platforms these cars use a 7-layer CNN, which take in image inputs and predict categorical throttle values (i.e. discreet steer and speed actuation controls). In comparison to our platform, this car uses a different CNN model, and it performs discreet control actuations.

Compared to the existing platforms, DeepNNCar has an ideal tradeoff of cost vs. autonomous driving functionalities.

4. DeepNNCar: Testbed for Autonomous Driving

To understand the methodology discussed in later sections, we first introduce DeepNNCar[1] (in Fig. 4) that is built using the chassis of the Traxxas Slash 2WD 1/10 scale remote-controlled car. It has two on-board motors - a servomotor for steering control, and a Titan 12T 550 motor for motive force - both of which are powered by an 8.4 V NiMH battery. A Raspberry Pi 3 (RPi3) is the onboard computational unit which performs all the required computations and interfaces with the sensors. RPi3 reserves two GPIO pins to generate Pulse Width Modulation (PWM) signals that are used to control the motors of the car. The PWM signal is defined by a duty cycle component and a frequency component. The duty cycle of a PWM signal is the proportion of time the signal remains in the high state (or logical 1) over the total time it takes to complete one cycle.

[1] Build instructions, source code, and videos of DeepNNCar can be found at: `https://github.com/scope-lab-vu/deep-nn-car`

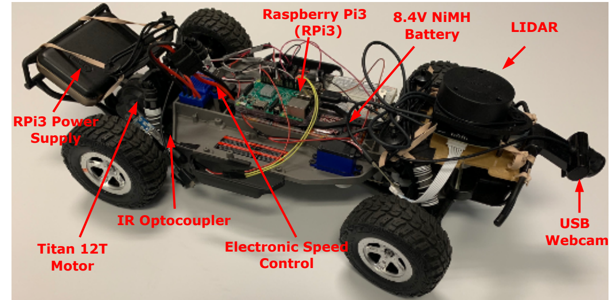

Figure 4: DeepNNCar is a resource constrained remote-controlled car that is designed to perform end-to-end learning based autonomous driving. The hardware components and operating modes of the car is discussed in Section 4.

For the DeepNNCar the duty cycle is the percentage of a digital square pulse with a period 10 ms (frequency = 100 Hz). Varying the PWM duty cycle percentage allows us to control the two onboard motors of the car. For the servomotor, varying the duty cycle $\in$ [10%, 20%] results in a continuous steering angle $\in$ [-30°, 30°]. Similarly, for the titan motor, varying the duty cycle $\in$ [15.58%, 15.70%] results in a speed $\in$ [0, 1] m/s. Throughout this work we use the notations (v) for speed and (θ) for steering. The speed is always refereed in-terms of m/s, and steering in degrees. However, to control the testbed, the steering and speed are varied as PWM duty cycle values.

The goals of DeepNNCar are: (1) minimize the soft constraint violations, i.e. reduce the out-of-track occurrences, and (2) optimize the speed based on the different track segments. i.e. each track segment has different maximum attainable speed (straight segment – 0.65 m/s, curved segment – 0.25 m/s), and the car must learn to switch between the speed modes.

4.1. Sensors and Operating Modes

Sensors: A USB webcam is attached to the RPi3 to capture images at 30 Frames Per Second (FPS) with a resolution of 320 × 240 RGB pixels. A slot-type IR Opto-coupler is attached to the chassis near the rear wheel that counts the revolutions of the wheel. The speed of the car is calculated based on the frequency of revolutions. The captured RGB images and the speed values are recorded on the computational unit and later utilized during the data collection, and training modes.

Operating Modes: DeepNNCar (server) is used along with a desktop (client) to provide a server-client setup for data collection, monitoring and runtime diagnostics. An Xbox controller is connected to the desktop via Bluetooth and it communicates with the car using TCP messages. Using this setup there are three different modes in which the car can function: (1) data collection – manually driving the car to collect training data (images, steering PWM duty cycle, and speed PWM duty cycle), (2) autonomous driving – for autonomous driving using LEC or other simplex strategies discussed in this work, and (3) livestream tracking – for streaming runtime images captured by the camera to the client for runtime tracking.

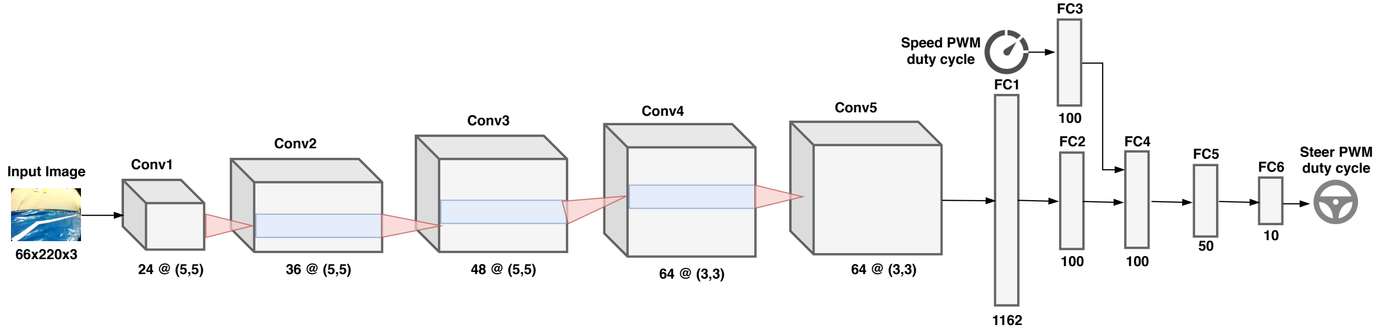

Figure 5: Modified NVIDIA's DAVE-II CNN, which takes in images and speed PWM duty cycle as input and predicts the steering PWM duty cycle. This modified model also takes in the speed as the inputs, which is not considered by the original NVIDIA DAVE-II model [3]. The model has five convolutional layers and six fully connected layers. Conv– represents the convolutional layers and sizes of the filters are mentioned below. FC – represents the fully connected layers and the number of neurons is mentioned below.

Figure 6: Track 1 and Track 2: on which the training data for the LEC was collected. The RL exploration for the dynamic-weighted simplex strategy was performed on Track 1; Track 3: used to test the performance and accuracy of the trained LEC, and the dynamic-weighted simplex strategy. The tracks were built indoor in our laboratory using 10' x 12' blue tarps. The videos of DeepNNCar preforming on these three tracks can be found in `https://github.com/scope-lab-vu/deep-nn-car`

4.2. Vehicle Control

Two independent controllers were developed for the DeepNNCar: one using a Learning Enabled Component (LEC) and the other using traditional image processing algorithms provided by OpenCV. Both controllers took the available camera images as input and were required to output PWM duty cycle values for both steering and speed control.

The speed control while driving the DeepNNCar using the LEC and OpenCV controller is initially set by a human supervisor. It is then controlled either using a constant throttle or a PID controller. Each of these controllers is discussed in more detail in the following sections.

4.2.1. LEC Controller

In the current implementation, the hardware performs autonomous driving based on e2e learning. For this we use a modified version of NVIDIA's DAVE-II CNN that takes in image and PWM duty cycle of speed v as inputs to predict the PWM duty cycle of steering (θ_L). This is an extension to the original DAVE-II CNN [3] that took in only image as input to predict steering θ_L.

Network Architecture: Our CNN model (in Fig. 5) has five convolutional layers and six fully connected layers. We modified the CNN architecture for two reasons. First, the steering and speed actions cannot be treated independently. Any change in the speed will impact the steering performance. Through experiments we observed that the modified CNN takes wider trajectories at turns compared to the original one. Second, since the quality of the captured image deteriorates as speed increases, additional information is required for the CNN to predict correct steering values.

Model Training and Validation: We train the CNN with 6000 labeled images collected from Track 1 and Track 2 (in Fig. 6). We also performed hyperparameter tuning [46] using random search [47] to select the appropriate learning rate, and number of epochs.

We validated the trained model using a test dataset of 1000 images. We then used Mean Square Error (MSE) [48] as a metric to quantify the distance between the actual and predicted steering values. Once the MSE was within our acceptable limits (0.1 – found through varied experiments), we deploy the model on the car for autonomous driving.

4.2.2. OpenCV Controller

The OpenCV controller is designed using classical image processing algorithms and it performs two tasks: (1) lane detection – finds lanes in images using classical lane detection algorithm, and (2) steer computation – associates a discreet steering (θ_C) based on the lanes detected. The output of these tasks are the detected lanes, the lane segments ($\hat{M}$) identified from number of lanes, the discreet steering θ_C (in degrees) and its corresponding PWM duty cycle, and a stop signal alarm when the car runs out of the track.

Lane Detection (LD) algorithm: We convert the 200×66 RGB image to gray scale (to reduce computation time) and then apply the LD algorithm. The different image processing involved in the LD process are:

- Gaussian blur and white masking: A 3x3 Gaussian kernel is convolved across the image to reduce noise. Next, all pixels except those within a specified range (e.g., [215, 255]) are masked, thus differentiating the track lanes from the foreground.
- Canny edge detection [49]: The algorithm first computes a gradient of pixel intensities. An upper and lower threshold of these gradients is defined at compile time. A comparison of

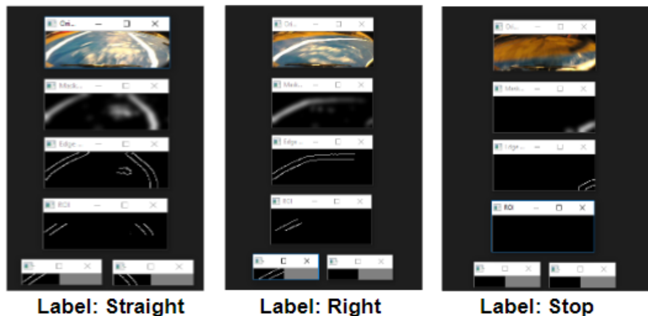

Figure 7: Image transformations as it passes through the different steps of the LD algorithm. Based on the lanes detected the OpenCV controller assigns a track segment and then associates a discreet steering value. During the livestream mode, real-time images captured from the car are relayed to the client, which runs LD algorithm continuously to get this visualization.

the pixel gradients to these thresholds in addition to hysteresis (suppress all weak and unconnected edges) can determine if a pixel is an edge or not. The edges reveal the boundary of the lanes.

- Region of interest (ROI) selection: The image is divided into two similar 30x66 regions of interest to capture the left and right lane respectively.
- Hough line transform [50]: A Hough line transform is applied to each ROI to detect the existence of a lane based on the results of the canny edge detection algorithm. Using this information, we determine a label for the track segment.

Fig. 7 shows the image transformation as it passes through the different steps of the LD algorithm. As seen from the figure, when the OpenCV controller detects two lanes it classifies the segment to be straight. Similarly, if it finds only the left lane, it classifies the segment to be right. Finally, if it does not find both the lanes, it classifies the track segment to be out and issues a stop signal.

Once the lanes are detected, the OpenCV controller associates a discrete steering angle θ_C, and its corresponding PWM duty cycle (similar to [51]): if two lanes are detected, $\theta_C = 0°$ (corresponds to a PWM duty cycle of 15%); if the right lane is detected, $\theta_C = -30°$ (corresponds to a PWM duty cycle of 10%); and if the left lane is detected, $\theta_C = 30°$ (corresponds to a PWM duty cycle of 20%).

The LD algorithm was tested with a dataset of 3000 images and correctly labeled the track regions with an accuracy of 89.6%. Table 2 summarizes the accuracy of the LD algorithm in correctly classifying the lanes in to straight or curved (left and right) segments. To generate this plot, we manually iterated through the 3000 images and made a note of the actual lane segment visualized by a human supervisor, and the lane segment as classified by the LD algorithm. Using this we gathered the number of correct lane predictions and mis-classifications to generate the data for the precision-recall graph. Using the precision and recall values we also computed the F1 score [52] - a metric which indicates the accuracy of the LD algorithm. The F1 scores for the straight and curved segments were 92.4% and 91.8% respectively. Higher F1 score indicates the LD algorithm's accuracy in classifying the lane segments correctly.

Track Segment	Precision (%)	Recall (%)
Straight	97.73	87.78
Curved	90.78	93.05

Table 2: The precision and recall values to evaluate the performance of the LD algorithm in different segments of the track. We manually iterated through 3000 images and compared the actual lane segments to those predicted by the LD algorithm.

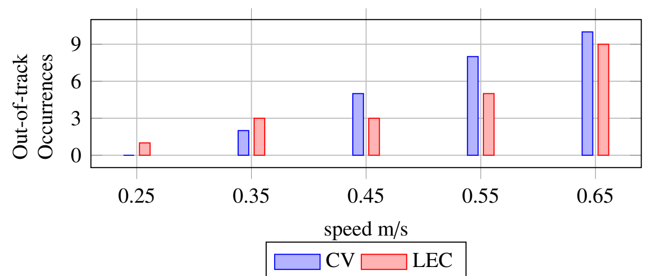

Figure 8: The out-of-track occurrences for different speeds in the curved segments of the track. From figure, CV: driving only with the OpenCV controller, LEC: driving only with the modified Dave-II model. The horizontal axis shows the different speeds of the car during the experiment. The data is collected by running DeepNNCar with each of these controllers independently around the track for 10 laps. The out-of-track occurrences was manually noted down by a human supervisor.

4.3. Performance of the Controllers

We evaluate the performance of these controllers based on their highest achievable speeds and the number of out-of-track occurrences. During the testing regiment, the DeepNNCar drives 10 laps each around the track for both the trained LEC controller and the OpenCV controller. The track was divided into straight and curved (left and right) segments, and performance results were compiled for each segment. The out-of-track occurrences was manually noted by a human supervisor. In the straight segment the LEC controller performed well with very few (typically 2 or 3) out-of-track occurrences up to a speed of 0.55 m/s. The OpenCV controller had typically 1 or no out-of-track occurrences up to 0.35 m/s. Above 0.35 m/s the OpenCV controller had higher chances of leading the car out-of-track as compared to the LEC controller.

The out-of-track occurrences of the different controllers in the curved segment of the track is shown in Fig. 8. The LEC has very few (typically 3) out-of-track occurrences up to a speed of 0.45 m/s and gradually as the speed increases the out-of-track occurrences increases. In comparison, the OpenCV controller performed well up to a speed of 0.35 m/s with only 1 out-of-track occurrences. Again, above 0.35 m/s the OpenCV controller started making higher wrong predictions leading the car out-of-track. At 0.65 m/s, the OpenCV controller had a higher out-of-track occurrence of 10 as compared to 8 of the LEC controller.

These results show that each controller outperforms the other controller in certain segments of operation. This implies that the total number of out-of-track occurrences can be reduced by intelligently blending the two controller actions. In the next section, we introduce the weighted simplex strategy for this purpose.

5. Dynamic-Weighted Simplex Strategy

Earlier, we discussed the two key challenges of using the Simplex Architecture, they are: (1) designing an effective decision logic, and (2) instantaneous switching between the controllers which may cause undesirable transient effects on system performance. In addition, the OpenCV controller did not perform reliably at high speeds (> 0.35 m/s) in the curved segments of the track (discussed in Section 4.3). So, the OpenCV controller cannot always be relied on as a safe controller when the vehicle is operating at high-speed in curved track segments.

To overcome these challenges, we compute the systems output as a weighted blending of the two controller outputs. This means the weights in Eq. (1) are no longer restricted to taking only values of 0 or 1, but can take any value in the continuous range [0,1]. However, the combination of the ensemble weights must sum to 1. Such blending approaches have been popularly used in model ensembles [18], and multiple model adaptive predictive control [19]. We call this approach the "weighted simplex strategy".

As shown in Fig. 3, the weighted simplex strategy works like the conventional Simplex Architecture in the high performance and the safe regions. The targeted spectrum of this strategy is the arbitration region, which is represented by the yellow grid in Fig. 3. In this region, the conventional Simplex Architecture performs an instantaneous transition from the AC to the BC. In contrast, the continuous ensemble weights of the weighted simplex strategy can be used to provide a smoother transition between the two controllers. However, one key challenge in using this strategy is the calculation of appropriate ensemble weights.

For this, we present a strategy called the "dynamic-weighted simplex strategy" which computes dynamic ensemble weights. This strategy uses an RL technique to find the optimal dynamic ensemble weights for the different segments of the track. In this work, we demonstrate the RL based ensemble weight selection process for DeepNNCar. The car's steering is computed as the weighted ensemble of the steering values computed by the LEC controller and the OpenCV controller. The weighted simplex combination for DeepNNCar steering is computed using Eq. (2).

$$\theta_D = W_L \cdot \theta_L + W_C \cdot \theta_C \tag{2}$$

Where, θ_D represents the PWM duty cycle corresponding to steering computed while using dynamic weights, W_L is the ensemble weight to the LEC, W_C is the ensemble weight to the OpenCV controller, θ_L is the PWM duty cycle corresponding to steering computed by the LEC controller and θ_C is the PWM duty cycle corresponding to steering computed by the OpenCV controller.

Other than selecting the W_L and W_C, the reward function (explained in detail in Section 5.1) in the RL setup is also designed to control the speed of the car. The PWM duty cycle of the new speed (v_{t+1}) can be incremented or decremented by (δv) based on the PWM duty cycle of the system's current speed (v_t).

$$v(t+1) = v_{(t)} \pm \delta v \tag{3}$$

5.1. The Learning Approach

The key contribution of our work is to find dynamic weights that optimally balances the performance and safety across all the segments of the track. As discussed earlier, we use RL to compute the dynamic ensemble weights. The continuous learning based setup of RL encouraged us to use it for our application, which has a dynamically changing environment (changing track segments). Any RL algorithm can be used in the decision logic, however in this work, we use the simplest of RL algorithms called the Q-learning [53] algorithm. Through this section we discuss the important elements and steps required to setup the Q-learning problem. The important elements required to setup the Q-learning problem are:

Environment: estimate the state (s) of a system based on an internal estimate of the Markov Decision Process (MDP) and computes a reward value (r) for each action (a) produced by an RL component. In the DeepNNCar, the environment is implemented as a standalone component within our middleware framework (described in the next section).

State (s): represents the current state of the system. For the DeepNNCar, the system state is continuously changing as the car interacts with the environment (track). Since our problem is to find optimal ensemble weights and optimal speed, we encode weights (W_L, W_C), the PWM duty cycle corresponding to speed (v), and the PWM duty cycle corresponding to steering θ_L and θ_C as the state information. As we do not have an explicit sensor for locating the position of the car, we use the steering values θ_L and θ_C to identify the position, and thus it is included as internal state information. The state s_t of the car at time t is: ($W_{L(t)}$, $W_{C(t)}$, $v_{(t)}$, θ_L, θ_C). The transition between the states happen when the element W_L changes by $\pm\ \delta W_L$, W_C changes by $\pm\ \delta W_C$, and the corresponding PWM duty cycle of v changes by $\pm\ \delta v$. These transitions are illustrated in Eq. (4).

$$\begin{bmatrix} W_{L(t+1)} \\ W_{C(t+1)} \\ v_{(t+1)} \end{bmatrix} = \begin{bmatrix} W_{L(t)} \\ W_{C(t)} \\ v_{(t)} \end{bmatrix} \pm \begin{bmatrix} \delta W_L \\ \delta W_C \\ \delta v \end{bmatrix} \tag{4}$$

Further, to limit the dimension of our state space we reduce the number of states in our MDP, by discretizing the elements of the state. Each weight $W_L, W_C \in [0, 1]$ is divided into 21 elements (W_C = 1-W_L, as the two weights sum up to 1) in increments of 0.05. Also, our tracks were small and we could not maintain the car within the track at speed > 0.65 m/s. Thus, we restricted the speed ∈ [0, 0.65] m/s, and the corresponding PWM duty cycle to $v \in$ [15.58%,15.62%]. We then divided this PWM range into 41 elements in increments of 0.001.

Reward: Designing an appropriate reward function is the key step in the Q-learning process. Our goal is to reduce the soft constraint violations while improving the performance of the system. Thus, we need to incorporate them in designing the reward function. Our reward function is a combination of the performance factor and the safety factor. In the case of the DeepNNCar, we chose speed to be the performance factor and a deviation of the car from the center of the track ($\hat{t}$) to be the safety factor. The calculated reward is expressed in Eq. (5):

$$r(s_t, a_t) = v_t \cdot (1 - \hat{t}) \tag{5}$$

	θ_L= 10% (Left)	θ_L= 11%	θ_L= 12%	θ_L= 13%	θ_L= 14%	θ_L= 15% (Straight)	θ_L= 16%	θ_L= 17%	θ_L= 18%	θ_L= 19%	θ_L=20% (Right)
θ_C = 20% (Right)	-	-	-	-	W_L=0.95 W_C=0.05	W_L=0.95 W_C=0.05	W_L=0.95 W_C=0.05	W_L=0.85 W_C=0.15	W_L=0.80 W_C=0.20	W_L=0.80 W_C=0.20	W_L=0.80 W_C=0.20
θ_C= 15% (Straight)	-	-	-	-	W_L=0.95 W_C=0.05	W_L=0.95 W_C=0.05	W_L=0.90 W_C=0.10	-	-	-	-
θ_C= 10% (Left)	-	-	-	W_L=0.80 W_C=0.20	-	-	-	W_L=0.90 W_C=0.10	-	-	-

Table 3: The variations in the ensemble weights are captured with respect to the change in the steering PWM duty cycle of LEC and OpenCV controller. The rows indicate the discretized steering PWM duty cycle of the OpenCV controller $\theta_C \in [10\%,15\%,30\%]$. The columns indicate the steering PWM duty cycle of the LEC controller $\theta_L \in [10\%,20\%]$. These steering values are discretized in steps of 1%. The blocks with "-" indicate those steering value combinations were not encountered in our experiments. The prime reason for this was our track was small and did not have very steep left turns. A large track with lots of turns could have better explored all the combinations. Also, it is evident from top right corner of the table that, W_C starts to increase as car starts turning in the right segment.

where, v_t is the current speed of the car and $\hat{t}$ is a scalar quantity calculated based on the deviation of the car from the center of the track. The measure $\hat{t}$ is calculated based on the lane segment information given by the LD algorithm. If the algorithm detects both lanes in the captured image, we infer that the car is at the center of the track and we assign $\hat{t} = 0$. If the algorithm detects only one lane, we assume the car has deviated from the center and we assign $\hat{t} = 1/2$. Finally, if no lanes are detected, we assume the car has moved out of the track and we assign $\hat{t} = 10$, a large penalty.

The positive reward given for remaining in the center of the track encourages the car to always select an action that keeps it within the track boundaries. Also, the reward value increases proportionately to vehicle speed which encourages the RL component to optimize for the highest achievable speed without exiting the track boundaries.

Action Space	$\uparrow v_t$ by 0.001	$\downarrow v_t$ by 0.001	NOP
$\uparrow W_L$ by 0.05	(0.95,0.05,15.591, 16,15)	(0.95,0.05,15.589, 16,15)	(0.95,0.05,15.590, 16,15)
$\downarrow W_L$ by 0.05	(0.85,0.15,15.591, 16,15)	(0.85,0.15,15.589, 16,15)	(0.85,0.15,15.590, 16,15)
NOP	(0.90,0.10,15.591, 16,15)	(0.90,0.10,15.589, 16,15)	(0.90,0.10,15.590, 16,15)

Table 4: Action space for a given state ($W_L = 0.90$, $W_C = 0.10$, $v_t = 15.590\%$, $\theta_L = 16\%$, θ_C=15%). W_C is computed as (1-W_L). Similar action combinations are generated for other states. NOP: means no operation. This is a sample action space when the car is in the straight segment of the track. So, the steering values remain almost the same in the next achievable states too.

Agent: In our setup the DeepNNCar is the agent. For each state $s \in S$, the agent performs an action $a \in A$, which results in a reward, $r : S \times A \rightarrow \mathbb{R}$, as the agent transitions from the initial state to a new state $s \rightarrow s' \in S$. An action space is created for all the different combinations of the state. As an example, the possible action space for the DeepNNCar when starting from the state $s = (W_L = 0.90,\ W_C = 0.10,\ v_t = 15.590\%,\ \theta_L = 16\%,\ \theta_C=15\%)$ is shown in Table 4. Since, there are three possible actions δW_L, δW_C and δv, there are 9 possible actions that can be performed from any state.

5.2. Exploration

Exploration is the training phase of RL, where the RL agent learns to select an appropriate action by continuously interacting with the environment. During this phase, the car stores results as state-action pairs $Q(s_t, a_t)$, which is a measurement of the quality of the immediate reward $r(s_t, a_t)$ from being in state s_t and taking action a_t discounted by the maximum expected future award in the new state denoted $Q'(s_{t+1}, a_{t+1})$. The new Q-state, $Q'(s_{t+1}, a_{t+1})$, is obtained by selecting an action which results in the maximum Q-value as shown in the Eq. (6) below.

$$Q'(s_{t+1}, a_{t+1}) = \max_{a_k \in A} Q(s_{t+1}, a_k) \tag{6}$$

The Q-state is updated using the Bellman equation which takes the current state and action as inputs along with the parameters $\alpha \in [0, 1]$ and $\gamma \in [0, 1]$. α controls the learning rate of the algorithm, while γ represents the discount factor which balances the importance of future benefits over immediate benefits (i.e. decreasing γ increases priority on obtaining immediate rewards).

$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + \alpha[r(s_t, a_t) + \gamma \cdot \max Q'(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{7}$$

The new Q-states and the Q-value calculated from the Q-learning algorithm are stored in a Q-Table, which is a look up table that holds the state-action pairs $Q(s_t, a_t)$ and the associated reward $r(s_t, a_t)$. For our experiments we used a learning rate $\alpha = 0.1$, discount factor $\gamma = 0.4$, and 1000 exploration steps. These hyper-parameters were obtained by tuning through various training runs. Also, for all the exploration runs we fix W_L=0.5, and W_C=0.5 as the initial state, and we start from different parts of the tracks. We hypothesize doing this could help us obtain a robust Q-Table with larger percentage of the state-action pairs explored.

Table 3 shows the ensemble weights learned during exploration of the different steering values. In the table, the columns list the PWM duty cycle corresponding to the LEC controller steering, $\theta_L \in [10\%,20\%]$, that is discretized in steps of 1. Similarly, the row lists the PWM duty cycle corresponding of the OpenCV controller steering, $\theta_C \in [10\%,20\%]$, that has been discretized in steps of 5. The resulting rectangular regions is shown in Table 3. The learned ensemble weights for each region of operation are listed within the corresponding block. A block with a "-" indicates a valid system state which was not represented in the training dataset, and thus no ensemble weights were learned. In addition, most of the blocks for θ_C=10

are not explored. The reason for this is our tracks were small and did not have steep left turns. A larger track with different turns would result in a larger exploration. Also, when the car is driving in a straight segment (row with θ_C=15), a higher weight is given to the output of the LEC. However, the OpenCV controller starts to get higher weights (see right corner of the row with θ_C=20) as the car moves towards the right curved segment.

5.3. Exploitation

During exploitation the car uses the learned action sequence stored in the Q-Table. The state-action pair with the highest Q-value is chosen as the action of the current state. During exploitation the learnt Q-Table is loaded onto the car and based on the steering PWM values, the agent starts moving towards the appropriate weights. In straight segments the weights quickly move to W_L=0.95, and W_C=0.05, and in curved segments the weights hover around W_L=0.8, and W_C=0.2.

If the state-space is not completely explored during the exploration phase, then there is a possibility that the agent may encounter unexplored states during exploitation and perform an incorrect action. This problem was encountered during our exploitation, causing occasional out-of-track occurrences. Since our tracks are small, 5 trial runs of 1000 steps exploration were enough to overcome this problem. Additionally, we had a decaying learning rate as the exploration run progressed. For systems with larger state spaces we would recommend a higher number of episodes and exploration steps.

To compare the performance of the dynamic weighted approach, we also introduce a fixed-weighted simplex strategy. The fixed weights version uses static weights for the different segments of the tracks.

5.4. Fixed-Weighted Simplex Strategy

The fixed-weighted simplex strategy is an empirical weight selection mechanism. It uses fixed weights across the different segments of track. The weights are found through a trial-and-error approach by a human supervisor. For the DeepNNCar running on Tack 1 (see Fig. 6), we found the optimal weights to be W_L=0.8, W_C=0.2. The weighted simplex steering equation using these weights is shown below:

$$\theta_F = 0.8 \cdot \theta_L + 0.2 \cdot \theta_C \tag{8}$$

where, θ_F represents the PWM duty cycle corresponding to steering computed while using fixed weights, θ_L is the PWM duty cycle corresponding to steering computed by the LEC controller and θ_C is the PWM duty cycle corresponding to steering computed by the OpenCV controller.

To implement the fixed weighted strategy on the DeepNNCar we use the concept introduced by Fridman et. al. [54] called "Arguing Machines" with a slight modification. Their work focused on combining an LEC based system with a human supervisor to aid its decision making during uncertain situations. Their approach had two LEC controllers predicting steering values, and when the difference (or argument) between their predicted steering values is higher than a predefined threshold (τ_{SW}), then the steering action is performed by a human driver. We replicate the same idea for DeepNNCar, but the human driver is replaced by Eq. (8). In our case, if the computed steering difference between the LEC controller and the OpenCV controller is greater than τ_{SW}, then Eq. (8) is used to compute the steering of the car. However, if the difference is lower than τ_{SW}, then the predicted steering of the LEC is chosen to drive the car.

Further, the speed of the car is also calculated based on the argument between the controllers. If the difference among the controller predictions is greater than τ_{SW}, then v_t is decremented using Eq. (3). However, if the difference among the controller predictions is lesser than τ_{SW}, then v_t is incremented as it suggests consensus among the controllers predictions.

5.5. Discussion

It is evident from the above sections that computing weights dynamically based on the operating contexts (like changing track segments) is better than the fixed weights. The main reason for this is the computation of the dynamic weights involves domain information (like the track segment), whereas the fixed weight is an empirical value calculated based on trail runs. Including contextual information has been found effective in different machine learning applications of data-driven anomaly detection [55], face recognition [56], speech recognition, and query classification [57]. In our setup, the use of contextual information in the reward function helps us compute a superior dynamic weights compared to fixed weights. We hypothesize these weights are optimal and are required to improve the safety and the performance of the system. Currently these weights are learnt offline, but an online learning technique that adjusts weights on the fly could be useful to tackle new unseen contexts like track, lighting levels, etc.

6. Resource Management

The dual controller operations of the weighted simplex strategy requires significant computational resources, which are not often available in small scale autonomous vehicles such as those used in hospitals [58], warehouse [59], and laboratory research testbeds [40], [43]. For the DeepNNCar, the weighted simplex strategy workload increased the power consumption, CPU utilization, and temperature of the RPi3 beyond 70°C (configured soft limit). Beyond the soft limit, the clock speed and the operating voltage of RPi3 are reduced [60], which could affect the performance of the car. We can address this problem in two ways: (1) *multiple computational units* – deploy multiple computational devices on-board the DeepNNCar, and distribute the tasks among them. However, this approach requires additional external power sources, which increases development cost of the platform, and (2) *computation offloading* – as the RPi3 supports WiFi connectivity, we setup wireless communication with other edge devices [2] or fog devices [3] and offload some tasks. In this work, we use the computation offloading

[2]Devices that have similar computational capacity as the onboard RPi3.

[3]Devices that have higher computational capacity than the onboard RPi3.

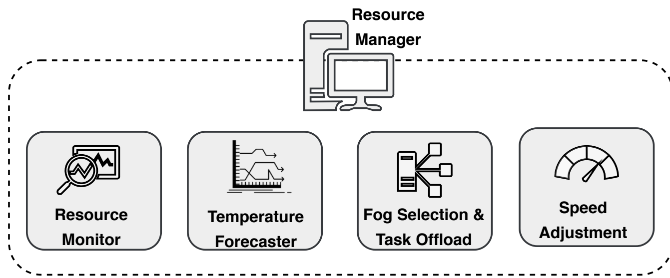

Figure 9: Resource Manager is responsible for resource monitoring, temperature forecasting, fog selection and task offload, and speed adjustment.

Fog1 Latency (ms)	Fog2 Latency (ms)	Fog1 avg Latency (ms)	Fog2 avg Latency (ms)	Selected Fog Device
13.0 11.4 11.2	10.8 11.7 14.0	11.87	12.17	Fog1
11.7 11.9 10.1	11.4 10.6 10.3	11.23	10.77	Fog2

Table 5: Fog device selection based on the average of three latency pings. The fog device with the lowest latency is selected for offloading. The experimental testbed for the fog device selection experiments had Fog1 device (laptop), Fog2 device (desktop), and each of these were connected to different WiFi networks.

approach to keep the development costs low, and utilize the wireless communication capability that enables computation on other available devices.

For this, we designed a Resource Manager (RM) that performs the following tasks: (1) resource monitoring – continuous monitoring of resource state (temperature and CPU Utilization), (2) temperature forecasting– forecast the temperature of the RPi3 based on current temperature and CPU utilization, (3) fog device selection and task offloading – selecting an optimal fog device and offloading non-critical tasks, and (4) vehicle speed adjustment– adjusting the vehicle speed (v) according to variations in the inference pipeline times. The workflow of the RM is illustrated in Fig. 9.

6.1. Resource Monitoring

For resource constrained computation platforms (e.g. RPi and NVIDIA Jetson), it is important to continuously monitor system utilization including CPU, memory, network, and disk. Variations in these utilization levels could result in degraded computational performance of the system. In our case, temperature and CPU utilization of RPi3 are identified to be the important system parameters that must be monitored, as they directly affect both the computational and hardware performance of the DeepNNCar.

To perform resource monitoring, the RM in the background continuously monitors the temperature and CPU utilization of the RPi3. This monitoring capability gives the ability to enact preventive measures that could avert the RPi3 from overheating. We use the psutil [61] library for retrieving utilization information and it runs in the background once every 30 seconds.

6.2. Temperature Forecasting

Forecasting the future system utilization based on the current workload is important for resource constrained computation platforms. The forecasting model when used in conjunction with the resource monitor can aid resource management by allowing time to enact mitigation strategies. For example, in our experiments, forecasting the RPi3 temperature based on current workload helped in preparing for task offloading (see Fig. 15).

The RM is responsible for forecasting the temperature of the RPi3 based on the current temperature and the CPU utilization (which represents the workload). To perform this forecasting, we designed a simple 2-layer DNN with 20 and 40 neurons in the first and second layer respectively. To collect data, we set up an experiment in which we continuously monitored temperature, CPU utilization, and offload status (indicated if the task was performed on the RPi3 or offloaded to a fog device).

The dataset consisted off three hours of resource monitoring data and was a representation of the entire offload process. We trained the model for 200 epochs with the data collected from the system utilization experiment. The performance and the accuracy of the model is discussed in Section 8 and illustrated in Fig. 14. The trained model was then deployed at runtime along with the resource monitor to forecast the temperature of the RPi3 in the next 30 seconds.

6.3. Fog Device Selection and Task Offloading

The RM continuously performs a latency ping every 10 seconds using the iPerf [62] networking tool to the available fog devices. After 30 seconds, an average time of the latest three ping tests is calculated and the fog device with the lowest average latency is selected for task offloading. Table 5 shows the latency test results for two 30 second periods, and how we select the appropriate fog device using average latency values. For these experiments we do not perform online discovery, but instead assume that we have prior knowledge of the available fog devices.

Once, the temperature forecasting model predicts the temperature to exceed 70°C and an optimal fog device has been selected, the RM prepares to offload the tasks. In our context it is not feasible to transfer the entire task code-base to the fog device at runtime, so we assume these devices have a copy of the code-base pre-installed. During runtime offloading the RM deactivates the code on the RPi3 and activates the same code on the fog device. The message transferring between the RPi3 and the fog device is managed using ZeroMQ (ZMQ). A similar task activation exchange is performed when the predicted temperature falls below 70°C. The forecasting model and the offloader synchronously work at runtime to plan about the offloading of the computations.

6.4. Vehicle Speed Adjustment

During the task offload process, the inference time T_R (discussed in Section 7) increases because of the additional network overhead in the wireless communication channel. As the inference time increases, the maximum allowable speed of the

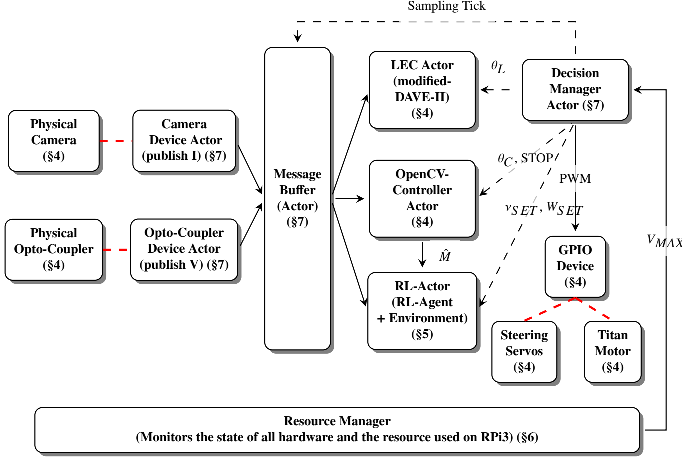

Figure 10: A block diagram of DeepNNCar along with actors. There are asynchronous interactions among various actors and thus different messaging patterns were used. The request-reply communications are shown with dotted lines, the publish-subscribe communications are shown in solid lines, and the red dotted lines indicate the hardware connections. Also, refer to the listed section numbers for a detailed description about the components.

vehicle must decrease. For this, the RM instructs the DM to saturate (limit) the top speed of the car. The saturated maximum speed v_{MAX} is calculated using the safe distance (d_S) which is the closest distance to the track during turning where the car can still safely perform an action to avoid going off the track. The d_S is a track specific quantity which was found to be 0.09m for our track (see Fig. 6). Therefore, any decision taken before reaching this distance will give the car a good turning radius. However, any decision taken after this distance will leave the car with insufficient space to turn and will result in a safety violation. v_{MAX} is computed as: $v_{MAX} = \frac{d_S}{T_R}$, where T_R is the inference pipeline time. This speed is converted to the corresponding PWM duty cycle value and applied to the RPi3 to control the speed of the car. During task offloading, the Decision Manager Actor (DMA) must wait longer for a reply from the offloaded component due to the latency overhead, which increases the T_R.

7. System Integration

We use an actor based design [63] to integrate the components discussed in the aforementioned sections. As discussed in [64], an actor is a self-constrained and restartable process that has its own execution thread and communicates synchronously or asynchronously with other actors. The actors of DeepNNCar and the data flow between them is shown in Fig. 10. Communication between actors is done with various ZMQ messaging patterns. The camera provides new images at 30 Hz and the IR opto-coupler continuously collects RPM data to compute the speed of the car. Then, the camera device actor[4] and the opto-coupler device actor periodically publish the images and current-speed to all subscriber actors. The LEC controller actor, the OpenCV controller actor, and the RL-Actor are aperiodic consumers (see [66]) which do not consume the sensor values until prompted by the DMA.

The interactions between the periodic publishers and aperiodic consumers are handled with the help of a Message Buffer Actor (MBA), which has a one buffer queue to store the published data (both images and current-speed) along with a sequence label. The data in the MBA gets updated according to the sampling period of the sensors. However, this data cannot be published until the MBA has received a sampling tick and a request from the DMA to publish the data of a certain label. Once the MBA receives this request, it publishes the image and current-speed messages to all subscribed actors. Using this data, the LEC actor predicts θ_L, the OpenCV controller computes θ_C, track position $\hat{M}$, and $STOP$ (a command issued if the car goes out of the track), and the RL-Actor computes W_L, W_C and v_{SET}.

Decision Manager Actor (DMA): This is the key component in the architecture that controls and initiates the entire message exchange process every inference cycle. The DMA issues requests for the sequence label and data θ_L, θ_C from the controllers, and for W_L, W_C, and v_{SET} from the RL-Actor. Once the controller and the RL-Actor have finished computation, they respond to the DMA with their label and values. The DMA then matches the labels and computes θ_D using Eq. (2) before feeding θ_D and v_{SET} to the GPIO device actor, which controls the two motors.

After applying the controls to the GPIO, the DMA starts a new cycle. This cycle continues indefinitely until it is terminated by the STOP signal from the OpenCV controller or manually by the user. The tasks performed between two sampling ticks of the DMA is one control cycle of the system and the time taken to perform one control cycle is referred to as the inference pipeline time T_R. T_R varies for every control cycle, but the average inference time for the dynamic-weighted simplex strategy is experimentally found to be 130 ms (in Fig. 13).

8. Evaluation

We evaluate the performance of the middleware framework and the introduced weighted simplex strategies by deploying it on the DeepNNCar. For this evaluation, we built three different indoor tracks shown in Fig. 6. These tracks were built in our laboratory using 10' x 12' tarps under controlled lighting conditions (higher lighting intensities create reflections on the tarp, causing the LD algorithm to fail). Each track had different geometric shapes and turns. These tracks also look wrinkled since they are folded for storage after experiments are complete. All the experiments were performed using the tracks in the same wrinkled conditions. The LEC was trained on the images collected from Track 1 and Track 2 and then tested on Track 3 to ensure the trained CNN had not overfit (generalized with the training data).

For evaluating the performance of the RM, we designed the DeepNNCar to communicate with a laptop (with an Intel 4-core processor) and a desktop (with AMD Ryzen Threadripper 16-core processor) using wireless communication over WiFi.

[4] A device actor converts hardware sensor information into topics that can be published and subscribed to (see [65]).

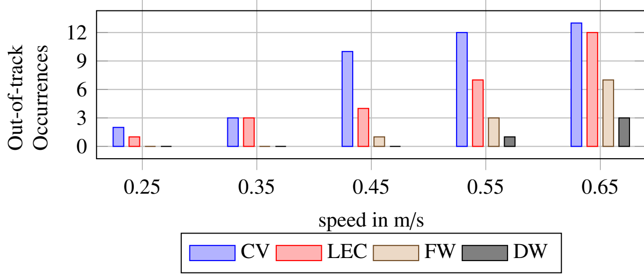

Figure 11: The out-of-track occurrences by different controllers for different speeds. From figure, CV: driving only with the OpenCV controller, LEC: driving only with the modified Dave-II model, FW: driving with fixed-weighted simplex strategy, and DW: driving with dynamic-weighted simplex strategy. The horizontal axis shows the different speeds of the car during the experiment. The DeepNNCar performs fewer out-of-track occurrences when combined with the dynamic-weighted simplex strategy. The data for this experiment was collected by running DeepNNCar with each of these controllers independently around Track 1 (see Fig. 6) for 10 laps.

8.1. Experiments on Weighted Simplex Performance

Out-of-track Occurrences: To evaluate the performance of the LEC controller, the OpenCV controller, and the weighted simplex strategies, we deployed them on DeepNNCar and performed various trial runs. Fig. 11 shows the number of out-of-track occurrences performed by the different controllers at different speeds. We varied the speed between [0.25 - 0.65] m/s and ran the different controllers separately for 10 laps around Track 1. From Fig. 11 it is evident that, at low speeds (< 0.35) m/s, both variants of weighted simplex strategy have lower out-of-track occurrences compared to the independent controllers. At higher speeds (0.65 m/s), the dynamic-weighted simplex strategy outperforms all other controllers reducing the number of out-of-track occurrences to 3.

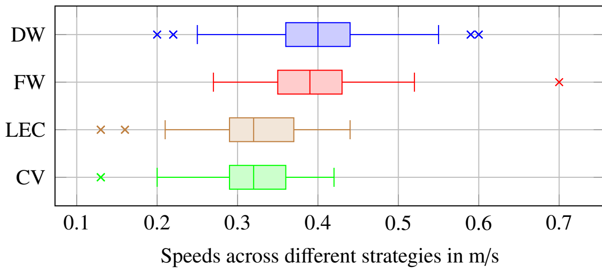

Figure 12: Speeds in meters per second of CV: driving only with the OpenCV controller, LEC: driving only with the modified Dave-II model, FW: driving with fixed-weighted simplex strategy, and (d) DW: driving with dynamic-weighted simplex strategy.

Speed Performance: Fig. 12 shows the maximum speed (represented as anomalies in the figure) and the average speed performance of the different controllers. To collect data for this experiment we ran the car with different controllers independently for 10 laps around Track 1. Both the OpenCV controller and the LEC controller operate with approximately the same average speeds of 0.29 - 0.39 m/s. This is because these controllers have a small number of out-of-track occurrences (see Fig. 11) in this speed range. However, the two weighted simplex strategies operate in a higher speed range because they perform significantly less out-of-track occurrences for speeds (< 0.4) m/s. Notably, the dynamic-weighted simplex strategy operates at a higher speed with lower out-of-track occurrences as compared to the other controllers.

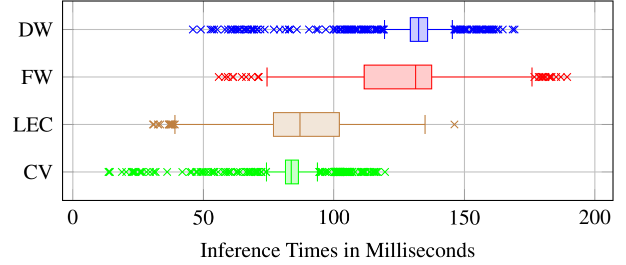

Figure 13: Inference times in milliseconds of CV: driving only with the OpenCV controller, LEC: driving only with the modified Dave-II model, FW: driving with fixed-weighted simplex strategy, and (d) DW: driving with dynamic-weighted simplex strategy. Inference time is affected by the state (computational load, temperature), so we have very high variance in the inference times.

Inference Times: We also evaluate the performance of the controllers based on inference times. Fig. 13 illustrates the inference times of the different controllers. It is evident from the figure that, the dual controller operation and the decision computation steps of the weighted simplex strategy increases the inference times of the car. The dynamic-weighted simplex strategy has an average inference time of 130 ms, and the fixed-weighted simplex strategy has an average of 120 ms whereas both independent controllers have smaller inference times of about 80 ms. The higher number of anomalies in this graph is due to the variation in inference times based on the computational load and the onboard temperature of the RPi3. As the RPi3 gets overheated and overloaded, the chances of varying inference times is higher.

Weighted Simplex Strategy	Straight Segment	Near Curved Segment	In Curved Segment
Dynamic Weights (WL, WC)	0.95, 0.05	0.85, 0.15	0.80, 0.20
Fixed Weights (WL, WC)	0.80, 0.20	0.80, 0.20	0.80, 0.20

Table 6: Comparing the ensemble weights of different simplex strategies. For the dynamic-weighted simplex strategy the weights were dynamically updated by the Q-learning algorithm. For the fixed-weighted simplex strategy we have a fixed weight for all the track segments, these weights were manually tuned by a human supervisor.

Fixed and Dynamic Weights: We performed an experiment to show how the weights of different weighted simplex strategies vary across different segments of the track. For this we clustered the Track 1 into three segments: Straight, Near Curved, and In Curved. Then as the car ran using the two weighted simplex strategies, we recorded the ensemble weights. We later classified the weights into the three segments. From Table 6 we see the weights used in the fixed-weighted strategy, remains constant across the three track segments. However, in the dynamic-weighted strategy the weights change dynamically and are different for the three different track segments.

The dynamic-weights in each segment are not always the same as shown in Table 6, but they vary ± 0.05. To simplify the table we have just listed a single value that occurred most often for a particular track segment.

Summary: Fig. 11 shows the introduced dynamic-weighted simplex strategy can operate at higher speeds with lower out-of-track occurrences. However, this comes with a penalty of increased inference times (see Fig. 13). From these results, it can be inferred that dynamic blending of the simplex weights helps in reducing soft constraint violations while achieving higher performance.

8.2. Experiments on Task Offloading

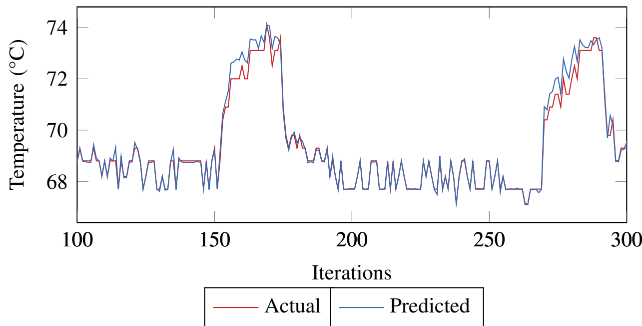

Figure 14: The result of the DNN temperature predictions vs. the actual temperature values. The DNN forecasts the temperature of the computing unit based on the current state (temperature, CPU utilization). We see the DNN predictions closely match the actual values. The graph shows a subset of iterations (total 10000 iterations).

Temperature variations: As discussed in Section 6, we design a temperature forecasting model using a DNN. Fig. 14 shows the predicted DNN temperatures vs. the actual temperature readings. We can see the DNN predictions closely match to the actual ones. To quantify the performance of the temperature forecaster, we use Mean Absolute Percentage Error (MAPE) as the metric. The results of MAPE = 0.16% across 1000 predictions indicates the model accurately predicts the temperature of the RPi3.

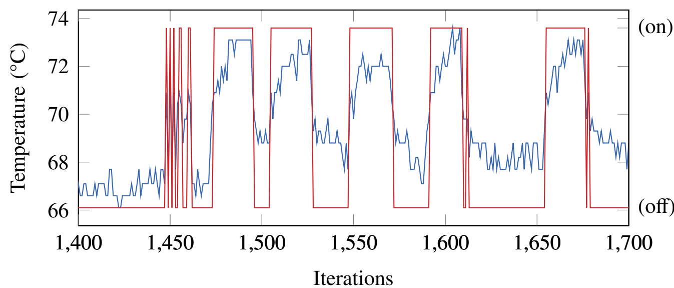

Figure 15: The effect of offloading the tasks in response to high temperature per iteration of the inference pipeline. The trigger to offload the task is 70°C. The blue line shows the temperature in Celsius. The red line shows when the tasks were offloaded to the fog (on=on fog, off=off fog). The graph shows a subset of iterations (total 10000 iterations).

Task Offloading: To evaluate the performance of the task offloader, we continuously offloaded the task onboard the DeepNNCar to any one of the available fog devices (laptop or a desktop). We then note the temperature variations when the tasks stay onboard and when they are offloaded. The fluctuating blue lines in Fig. 15 shows the temperature variations when the task was moved on and off the RPi3. The red lines indicate if the tasks were executed on or off the RPi3. As observed, the temperature drops below the threshold (70°C) when the tasks get offloaded.

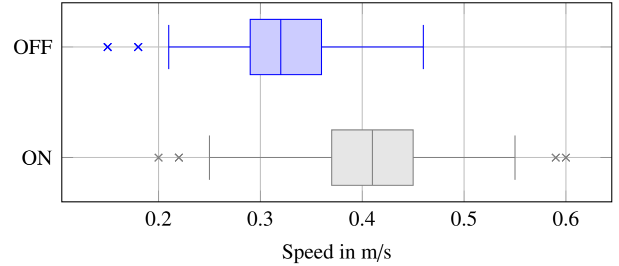

Figure 16: Speed readjustment during offload (m/s) ON: dynamic-weighted simplex strategy with all tasks executed onboard, and OFF: dynamic-weighted simplex strategy with RL task offloaded (Q-table was offloaded).

Speed Performance: Fig. 16 shows that the DeepNNCar can operate at higher speeds (≈ 0.42 m/s) when all tasks are preformed onboard. As tasks are offloaded to fog devices, the operating speed decreases to 0.34 m/s. This is because of the continuous speed adjustment performed to compensate for the increased inference times (due to increased latency overhead).

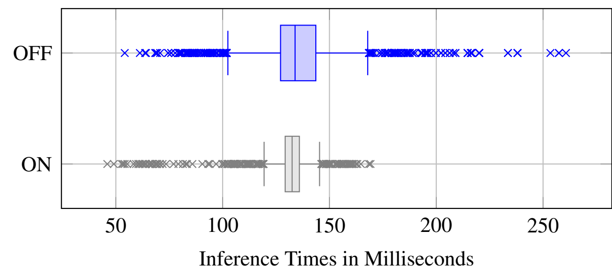

Figure 17: ON: dynamic-weighted simplex strategy with all tasks executed onboard, and OFF: dynamic-weighted simplex strategy with RL task offloaded (Q-table was offloaded). The higher overloaded, and overheated the RPi3 gets, the inference times gets higher. Inference time is affected by the state (computational load, temperature), so we have very high variance in the inference times.

Inference Times: Fig. 15 shows task offloading performed to maintain the RPi3's temperature below the threshold (70°C). During offloading, the inference times increase because of the network overhead involved in sending out the computations to the fog devices. The inference time comparison of the car when tasks get offloaded vs. not-offloaded is shown in Fig. 17. The dynamic-weighted simplex strategy with all tasks performed onboard has a lower inference time (≈ 130 ms) compared to the inference time (≈ 140 ms) with RL tasks offloaded (Q-Table was offloaded).

The higher number of anomalies in this graph is because inference times vary based on the computational load and the onboard temperature of the RPi3. As the RPi3 overheats and gets overloaded, the chance of varying inference times are higher. In

addition, during offloading the wireless exchange of messages is performed over Vanderbilt University WiFi. The network traffic variations in the WiFi adds on to the latency overhead.

Summary: It is evident from Fig. 15 that the resource manager tries to keep a tight check on the RPi3 temperature by offloading the RL task onto available fog devices. However, to balance the increase in inference times during offloading (see Fig. 17), the resource manager has to penalize or adjust the speed of the car. The average speed during offload is 0.34 m/s, compared to 0.42 m/s without the offload. Since, the primary concern of this paper is to reduce the soft constraint violations, the slight penalization in the performance (speed) is acceptable. For small-scale CPS systems with limited computational capacity, we would recommend the use of a resource manager along with an optimal dynamic-weighted blending strategy as discussed in this work.

9. Conclusion

In this work, we have discussed the problems associated in using LECs for perception in autonomous systems. We have further implemented an LEC based end-to-end learning controller on our physical testbed, the DeepNNCar. We have also discussed the two key challenges of using the classical Simplex Architecture. They are: (1) designing effective decision logic, and (2) mitigating sudden transitions. To address these research challenges, we discuss: (1) a *dynamic-weighted simplex strategy* which computes dynamic simplex weights according to the segments of the track, (2) a *middleware framework* which allows the integration of the introduced dynamic-weighted strategy and provides a resource manager for monitoring the onboard computational unit, and (3) a *hardware testbed* to deploy and test the proposed concepts. We have further evaluated the performance of the dynamic-weighted simplex strategy in terms of its capability to reduce soft constraint violations while improving the system's performance. Our results show the dynamic-weighted simplex strategy works well at higher speeds ($\sim$ 0.4 m/s) with low out-of-track occurrences as compared to only the LEC driven controller and the OpenCV controller. In addition, the evaluation results also show the resource manager to be effective in mitigating the computational overload generated by the dynamic-weighted simplex strategy.

This framework finds utility in factories, warehouses, hospitals where various levels of safe autonomy are required to perform tasks. Currently, the dynamic-weighted simplex strategy is suitable for systems which can tolerate soft constraint violations. However, in the future we would like to extend this strategy to compute dynamic ensemble weights that can be applicable to systems with hard constraints. Further, we would like to extend our middleware framework with an architecture such as CHARIOT [67], which provides a mechanism for achieving autonomous resilience. Network failure is a significant problem in our current setup. Using CHARIOT could help us detect this failure and mitigate it.

Acknowledgements: This work was supported by DARPA's Assured Autonomy project and Air Force Research Laboratory. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or AFRL.

References

[1] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.

[2] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[3] 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).

[4] Just how far ahead is tesla in self-driving, `https://www.forbes.com/sites/greatspeculations/2019/11/08/just-how-far-ahead-is-tesla-in-self-driving/#3b71f97c1b24`.

[5] V. Badrinarayanan, A. Kendall, R. Cipolla, Segnet: A deep convolutional encoder-decoder architecture for image segmentation, IEEE transactions on pattern analysis and machine intelligence 39 (12) (2017) 2481–2495.

[6] P. J. Graydon, J. C. Knight, E. A. Strunk, Assurance based development of critical systems, in: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), IEEE, 2007, pp. 347–357.

[7] R. Bloomfield, P. Bishop, Safety and assurance cases: Past, present and possible future–an adelard perspective, in: Making Systems Safer, Springer, 2010, pp. 51–67.

[8] K. Pei, Y. Cao, J. Yang, S. Jana, DeepXplore: Automated whitebox testing of deep learning systems, in: Proceedings of the 26th Symposium on Operating Systems Principles, ACM, 2017, pp. 1–18.

[9] W. Xiang, H.-D. Tran, T. T. Johnson, Reachable set computation and safety verification for neural networks with ReLU activations, arXiv preprint arXiv:1712.08163 (2017).

[10] A. Boloor, K. Garimella, X. He, C. Gill, Y. Vorobeychik, X. Zhang, Attacking vision-based perception in end-to-end autonomous driving models, arXiv preprint arXiv:1910.01907 (2019).

[11] S. Neema, Assured autonomy, DARPA Research Program. (2019).

[12] D. Seto, B. Krogh, L. Sha, A. Chutinan, The simplex architecture for safe on-line control system upgrades, in: Proceedings of the American Control Conference, Vol. 6, AMERICAN AUTOMATIC CONTROL COUNCIL, 1998, pp. 3504–3508.

[13] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, P. Kumar, The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures, in: 28th IEEE International Real-Time Systems Symposium (RTSS 2007), IEEE, 2007, pp. 400–412.

[14] D. Seto, E. Ferreira, T. F. Marz, Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis), Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST (2000).

[15] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, L. Sha, The system-level simplex architecture for improved real-time embedded system safety, in: Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE, IEEE, 2009, pp. 99–107.

[16] S. Bak, T. T. Johnson, M. Caccamo, L. Sha, Real-time reachability for verified simplex design, in: Real-Time Systems Symposium (RTSS), 2014 IEEE, IEEE, 2014, pp. 138–148.

[17] J. Finch, Toyota sudden acceleration: a case study of the national highway traffic safety administration-recalls for change, Loy. Consumer L. Rev. 22 (2009) 472.

[18] D. Jiménez, Dynamically weighted ensemble neural networks for classification, in: 1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98CH36227), Vol. 1, IEEE, 1998, pp. 753–756.

[19] W. Zhang, Stable weighted multiple model adaptive control with improved convergence rate, IFAC Proceedings Volumes 45 (13) (2012) 570–575.

[20] S. Levine, Deep reinforcement learning, `http://rll.berkeley.edu/deeprlcourse/f17docs/lecture_1_introduction.pdf`.

[21] S. Ullman, Against direct perception, Behavioral and Brain Sciences 3 (3) (1980) 373–381.

[22] C. Chen, A. Seff, A. Kornhauser, J. Xiao, Deepdriving: Learning affordance for direct perception in autonomous driving, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 2722–2730.

[23] D. Mantegazza, J. Guzzi, L. M. Gambardella, A. Giusti, Vision-based control of a quadrotor in user proximity: Mediated vs end-to-end learning approaches, arXiv preprint arXiv:1809.08881 (2018).

[24] S. Levine, C. Finn, T. Darrell, P. Abbeel, End-to-end training of deep visuomotor policies, The Journal of Machine Learning Research 17 (1) (2016) 1334–1373.

[25] X. Zhu, A. B. Goldberg, Introduction to semi-supervised learning, Synthesis lectures on artificial intelligence and machine learning 3 (1) (2009) 1–130.

[26] U. Muller, J. Ben, E. Cosatto, B. Flepp, Y. L. Cun, Off-road obstacle avoidance through end-to-end learning, in: Advances in neural information processing systems, 2006, pp. 739–746.

[27] M. Bajracharya, A. Howard, L. H. Matthies, B. Tang, M. Turmon, Autonomous off-road navigation with end-to-end learning for the LAGR program, Journal of Field Robotics 26 (1) (2009) 3–25.

[28] D. A. Pomerleau, ALVINN: An autonomous land vehicle in a neural network, in: Advances in neural information processing systems, 1989, pp. 305–313.

[29] D. Seto, L. Sha, A case study on analytical analysis of the inverted pendulum real-time control system, Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST (1999).

[30] K. Lee, L. Sha, A dependable online testing and upgrade architecture for real-time embedded systems, in: 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05), IEEE, 2005, pp. 160–165.

[31] S. Bak, K. Manamcheri, S. Mitra, M. Caccamo, Sandboxing controllers for cyber-physical systems, in: Cyber-Physical Systems (ICCPS), 2011 IEEE/ACM International Conference on, IEEE, 2011, pp. 3–12.

[32] E. Asarin, O. Bournez, T. Dang, O. Maler, Approximate reachability analysis of piecewise-linear dynamical systems, in: International Workshop on Hybrid Systems: Computation and Control, Springer, 2000, pp. 20–31.

[33] P. Vivekanandan, G. Garcia, H. Yun, S. Keshmiri, A simplex architecture for intelligent and safe unmanned aerial vehicles, in: 2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE, 2016, pp. 69–75.

[34] D. Phan, J. Yang, M. Clark, R. Grosu, J. D. Schierman, S. A. Smolka, S. D. Stoller, A component-based simplex architecture for high-assurance cyber-physical systems, in: 17th International Conference on Application of Concurrency to System Design, ACSD 2017, Zaragoza, Spain, June 25-30, 2017, 2017, pp. 49–58. doi:10.1109/ACSD.2017.23. URL `https://doi.org/10.1109/ACSD.2017.23`

[35] A. Lomuscio, L. Maganti, An approach to reachability analysis for feed-forward relu neural networks, arXiv preprint arXiv:1706.07351 (2017).

[36] R. Seiger, C. Seidl, U. Aßmann, T. Schlegel, A capability-based framework for programming small domestic service robots, in: Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering, ACM, 2015, pp. 49–54.

[37] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, Ros: an open-source robot operating system, in: ICRA workshop on open source software, Vol. 3, Kobe, Japan, 2009, p. 5.

[38] E. De Coninck, S. Bohez, S. Leroux, T. Verbelen, B. Vankeirsbilck, B. Dhoedt, P. Simoens, Middleware platform for distributed applications incorporating robots, sensors and the cloud, in: 2016 5th IEEE International Conference on Cloud Networking (Cloudnet), IEEE, 2016, pp. 218–223.

[39] S. Bohez, E. De Coninck, T. Verbelen, P. Simoens, B. Dhoedt, Enabling component-based mobile cloud computing with the aiolos middleware, in: Proceedings of the 13th Workshop on Adaptive and Reflective Middleware, ACM, 2014, p. 2.

[40] M. O'Kelly, V. Sukhil, H. Abbas, J. Harkins, C. Kao, Y. V. Pant, R. Mangharam, D. Agarwal, M. Behl, P. Burgio, et al., F1/10: An open-source autonomous cyber-physical platform, arXiv preprint arXiv:1901.08567 (2019).

[41] C. Liu, P. Jiang, A cyber-physical system architecture in shop floor for intelligent manufacturing, Procedia Cirp 56 (2016) 372–377.

[42] T. Samad, S. Iqbal, A. W. Malik, O. Arif, P. Bloodsworth, A multi-agent framework for cloud-based management of collaborative robots, International Journal of Advanced Robotic Systems 15 (4) (2018) 1729881418785073.

[43] S. Karaman, A. Anders, M. Boulet, J. Connor, K. Gregson, W. Guerra, O. Guldner, M. Mohamoud, B. Plancher, R. Shin, et al., Project-based, collaborative, algorithmic robotics for high school students: Programming self-driving race cars at mit, in: 2017 IEEE Integrated STEM Education Conference (ISEC), IEEE, 2017, pp. 195–203.

[44] M. G. Bechtel, E. McEllhiney, H. Yun, DeepPicar: A low-cost deep neural network-based autonomous car, arXiv preprint arXiv:1712.08644 (2017).

[45] W. Roscoe, Donkey car: An opensource diy self driving platform for small scale cars.

[46] J. S. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, in: Advances in neural information processing systems, 2011, pp. 2546–2554.

[47] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, Journal of Machine Learning Research 13 (Feb) (2012) 281–305.

[48] C. J. Willmott, K. Matsuura, Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance, Climate research 30 (1) (2005) 79–82.

[49] J. Canny, A computational approach to edge detection, IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8 (6) (1986) 679–698. doi:10.1109/TPAMI.1986.4767851.

[50] J. Illingworth, J. Kittler, A survey of the hough transform, Computer vision, graphics, and image processing 44 (1) (1988) 87–116.

[51] K. McFall, Using visual lane detection to control steering in a self-driving vehicle, in: Smart City 360°, Springer, 2016, pp. 861–873.

[52] C. Goutte, E. Gaussier, A probabilistic interpretation of precision, recall and f-score, with implication for evaluation, in: European Conference on Information Retrieval, Springer, 2005, pp. 345–359.

[53] C. J. Watkins, P. Dayan, Q-learning, Machine learning 8 (3-4) (1992) 279–292.

[54] L. Fridman, B. Jenik, B. Reimer, Arguing machines: Perceptioncontrol system redundancy and edge case discovery in real-world autonomous driving, arXiv preprint arXiv:1710.04459 (2017).

[55] G. Biswas, H. Khorasgani, G. Stanje, A. Dubey, S. Deb, S. Ghoshal, An approach to mode and anomaly detection with spacecraft telemetry data, International Journal of Prognostics and Health Management (2016).

[56] M. Davis, M. Smith, J. Canny, N. Good, S. King, R. Janakiraman, Towards context-aware face recognition, in: Proceedings of the 13th annual ACM international conference on Multimedia, ACM, 2005, pp. 483–486.

[57] H. Cao, D. H. Hu, D. Shen, D. Jiang, J.-T. Sun, E. Chen, Q. Yang, Context-aware query classification, in: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval, ACM, 2009, pp. 3–10.

[58] A. G. Ozkil, Z. Fan, S. Dawids, H. Aanes, J. K. Kristensen, K. H. Christensen, Service robots for hospitals: A case study of transportation tasks in a hospital, in: 2009 IEEE International Conference on Automation and Logistics, IEEE, 2009, pp. 289–294.

[59] K. Bhasin, P. Clark, How amazon triggered a robot arms race, Bloomberg Techology (2016).

[60] Raspberry Pi frequency management, `https://www.raspberrypi.org/documentation/hardware/raspberrypi/frequency-management.md`.

[61] G. Rodola, psutil documentation; 2017.

[62] iPerf: A tool for measuring network performance, `https://iperf.fr/`.

[63] G. A. Agha, Actors: A model of concurrent computation in distributed systems., Tech. rep., Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab (1985).

[64] M.-W. Jang, S. Reddy, P. Tosic, L. Chen, G. Agha, An actor-based simulation for studying uav coordination, in: 15th European Simulation Symposium, 2005, p. 323.

[65] A. Dubey, G. Karsai, P. Volgyesi, M. Metelko, I. Madari, H. Tu, Y. Du, S. Lukic, Device access abstractions for resilient information architecture platform for smart grid, IEEE Embedded Systems Letters (2018) 1–1doi:10.1109/LES.2018.2845854.

[66] N. Mahadevan, A. Dubey, G. Karsai, Model-based software health management for real-time systems, in: IEEE Aerospace Conference(AERO), Vol. 00, 2011, pp. 1–18. doi:10.1109/AERO.2011.5747559. URL `doi.ieeecomputersociety.org/10.1109/AERO.2011.5747559`

[67] S. M. Pradhan, A. Dubey, A. Gokhale, M. Lehofer, Chariot: A domain specific language for extensible cyber-physical systems, in: Proceedings of the workshop on domain-specific modeling, ACM, 2015, pp. 9–16.