DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car

Michael Garrett Bechtel, Elise McEllhiney, Heechul Yun
{m783b224, e908m429, heechul.yun}@ku.edu
University of Kansas, USA

***Abstract*—We present DeepPicar, a low-cost deep neural network (DNN) based autonomous car platform. DeepPicar is a small scale replication of a real self-driving car called Dave-2 by NVIDIA, which drove on public roads using a deep convolutional neural network (CNN), that takes images from a front-facing camera as input and produces car steering angles as output. DeepPicar uses the exact same network architecture—9 layers, 27 million connections and 250K parameters—and can be trained to drive itself, in real-time, using a web camera and a modest Raspberry Pi 3 quad-core platform. Using DeepPicar, we analyze the Pi 3's computing capabilities to support end-to-end deep learning based real-time control of autonomous vehicles. We also systematically compare other contemporary embedded computing platforms using the DeepPicar's CNN based real-time control software as a workload. We find all tested platforms, including the Pi 3, are capable of supporting deep-learning based real-time control, from 20 Hz up to 100 Hz depending on hardware platform. However, shared resource contention remains an important issue that must be considered in applying deep-learning models on shared memory based embedded computing platforms.**

I. Introduction

Autonomous cars have been a topic of increasing interest in recent years as many companies are actively developing related hardware and software technologies toward fully autonomous driving capability with no human intervention. Deep neural networks (DNNs) have been successfully applied in various perception and control tasks in recent years. They are important workloads for autonomous vehicles as well. For example, Tesla Model S was known to use a specialized chip (MobileEye EyeQ), which used a deep neural network for vision-based real-time obstacle detection and avoidance. More recently, researchers are investigating DNN based end-to-end control of cars [3] and other robots. It is expected that more DNN based Artificial Intelligence workloads may be used in future autonomous vehicles.

Executing these AI workloads on an embedded computing platform poses several additional challenges. First, many AI workloads in vehicles are computationally demanding and have strict real-time requirements. For example, latency in a vision-based object detection task may be directly linked to safety of the vehicle. This requires a high computing capacity as well as the means to guaranteeing the timings. On the other hand, the computing hardware platform must also satisfy cost, size, weight, and power constraints, which require a highly efficient computing platform. These two conflicting requirements complicate the platform selection process as observed in [14].

To understand what kind of computing hardware is needed for AI workloads, we need a testbed and realistic workloads. While using a real car-based testbed would be most ideal, it is not only highly expensive, but also poses serious safety concerns that hinder development and exploration. Therefore, there is a strong need for safer and less costly testbeds. There are already several relatively inexpensive RC-car based testbeds, such as MIT's RaceCar [16] and UPenn's F1/10 racecar [1]. However, these RC-car testbeds still cost more than $3,000, requiring considerable investment.

Instead, we want to build a low cost testbed that still employs the state-of-the art AI technologies. Specifically, we focus on a end-to-end deep learning based real-time control system, which was developed for a real self-driving car, NVIDIA DAVE-2 [3], and use the same methodology on a smaller and less costly setup. In developing the testbed, our goals are (1) to analyze real-time issues in DNN based end-to-end control; and (2) to evaluate real-time performance of contemporary embedded platforms for such workload.

In this paper, we present DeepPicar, a low-cost autonomous car platform for research. From a hardware perspective, DeepPicar is comprised of a Raspberry Pi 3 Model B quad-core computer, a web camera and a RC car, all of which are affordable components (less than $100 in total). The DeepPicar, however, employs state-of-the-art AI technologies, including a vision-based end-to-end control system that utilizes a deep convolutional neural network (CNN). The network receives an image frame from a single forward looking camera as input and generates a predicted steering angle value as output at each control period in *real-time*. The network has 9 layers, about 27 million connections and 250 thousand parameters (weights). The network architecture is identical to that of NVIDIA's DAVE-2 self-driving car [3], which uses a much more powerful computer (Drive PX computer [12]) than a Raspberry Pi 3. We chose to use a Pi 3 not only because of its affordability, but also because it is representative of today's mainstream low-end embedded multicore platforms found in smartphones and other embedded devices.

We apply a standard imitation learning methodology to train the car to follow tracks on the ground. We collect data for training and validation by manually controlling the RC car and recording the vision (from the webcam mounted on the RC-car) and the human control inputs. We then train the network offline using the collected data on a desktop computer, which is equipped with a NVIDIA GTX 1060 GPU. Finally, the trained network is copied back to the Raspberry Pi, which is then used to perform inference operations—locally on the Pi—in the car's main control loop in real-time. For real-time control, each inference operation must be completed within the desired control period (e.g., 50 ms period for 20 Hz control frequency).

Using the DeepPicar platform, we systematically analyze its real-time capabilities in the context of deep-learning based real-time control, especially on real-time deep neural network inferencing. We also evaluated other, more powerful, embedded computing platforms to better understand achievable real-time performance of DeepPicar's deep-learning based control system and the impact of computing hardware architectures.

We find that the DeepPicar is capable of completing control loops in under 50 ms, or 20 hz, and can do so almost 99% of the time. Other tested embedded platforms, Intel Up and NVIDIA TX2, offer even better performance, capable of supporting deep-learning based real-time control up to 100 Hz control frequency on the TX2 when the GPU is used. However, in all platforms, shared resource contention remains an important issue as we observe up to 9.6X control loop execution time increase, mostly due to increase in the nueral network inferencing operation, when memory performance intensive applications are co-scheduled on idle CPU cores.

The **contributions** of this paper are as follows:

- We present the design and implementation of a low-cost autonomous vehicle testbed, DeepPicar, which utilizes state-of-the-art artificial intelligence techniques.
- We provide an analysis and case-study of real-time issues in the DeepPicar.
- We systematically compare real-time computing capabilities of multiple embedded computing platforms in the context of vision-based autonomous driving.

The remaining sections of the paper are as follows: Section II provides a background of applications in autonomous driving and related works. Section III gives an overview of the DeepPicar platform, including the high-level system and the methods used for training and inference. Section IV presents our evaluation of the platform and how different factors can affect performance. Section V gives a comparison between the Raspberry Pi 3 and other embedded computing platforms to determine their suitably for autonomous driving research. The paper finishes with conclusions in Section VI.

II. Background and Related Work

In this section, we provide background and related work on the application of deep learning in robotics, particularly autonomous vehicles.

A. *End-to-End Deep Learning for Autonomous Vehicles*

To solve the problem of autonomous driving, a standard approach has been decomposing the problem into multiple sub-problems, such as lane marking detection, path planning, and low-level control, which together form a processing pipeline [3]. Recently, researchers are exploring another approach that dramatically simplifies the standard control pipeline by applying deep neural networks to directly produce control outputs from senor inputs [11]. Figure 1 shows the differences between two approaches.

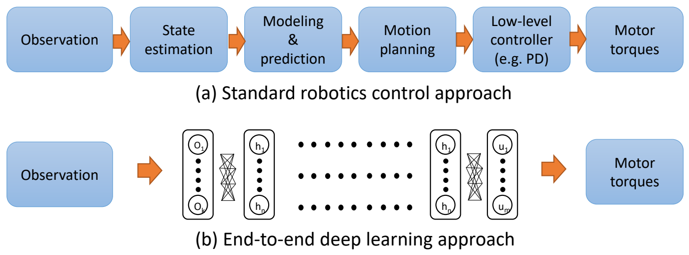
Observation
State estimation
Modeling & prediction
Motion planning
Low-level controller (e.g. PD)
Motor torques
(a) Standard robotics control approach
Observation
Motor torques
(b) End-to-end deep learning approach

Fig. 1: Standard robotics control vs. DNN based end-to-end control. Adopted from [10].

The use of neural networks for end-to-end control of autonomous car was first demonstrated in late 1980s [15], using a small 3-layer fully connected neural network; and subsequently in a DARPA Autonomous Vehicle (DAVE) project in early 2000s [9], using a 6 layer convolutional neural network (CNN); and most recently in NVIDIA's DAVE-2 project [3], using a 9 layer CNN. In all of these projects, the neural network models take raw image pixels as input and directly produce steering control commands, bypassing all intermediary steps and hand-written rules used in the conventional robotics control approach. NVIDIA's latest effort reports that their trained CNN autonomously controls their modified cars on public roads without human intervention [3].

Using deep neural networks involves two distinct phases [13]. The first phase is *training* during which the weights of the network are incrementally updated by backpropagating errors it sees from the training examples. Once the network is trained—i.e., the weights of the network minimize errors in the training examples—the next phase is *inferencing*, during which unseen data is fed to the network as input to produce predicted output (e.g., predicted image classification). In general, the training phase is more computationally intensive and requires high throughput, which is generally not available on embedded platforms. The inferencing phase, on the other hand, is relatively less computationally intensive and latency becomes as important, if not moreso, as computational throughput, because many use-cases have strict real-time requirements (e.g., search query latency).

B. *Embedded Computing Platforms for Real-Time Inferencing*

Real-time embedded systems, such as an autonomous vehicle, present unique challenges for deep learning, as the computing platforms of such systems must satisfy two often conflicting goals [14]: The platform must provide enough computing capacity for real-time processing of computationally expensive AI workloads (deep neural networks); The platform

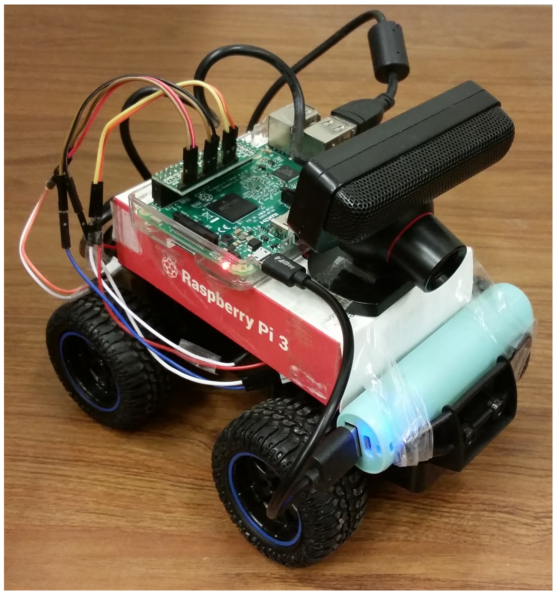

Fig. 2: DeepPicar platform.

must also satisfy various constraints such as cost, size, weight, and power consumption limits.

Accelerating AI workloads, especially inferencing operations, has received a lot of attention from academia and industry in recent years as applications of deep learning are broadening to areas of real-time embedded systems such as autonomous vehicles. These efforts include the development of various heterogeneous architecture-based system-on-a-chips (SOCs) that may include multiple cores, GPU, DSP, FPGA, and neural network optimized ASIC hardware [7]. Consolidating multiple tasks on SoCs with lots of shared hardware resources while guaranteeing real-time performance is also an active research area, which is orthogonal to improving raw performance. Consolidation is necessary for efficiency, but unmanaged interference can nullify the benefits of consolidation [8]. For these reasons, finding a good computing platform is a non-trivial task, one that requires a deep understanding of the workloads and the hardware platform being utilized.

The primary objectives of this study are to understand (1) the necessary computing performance for applying AI technology-based robotics systems, and (2) what kind of computing architecture and runtime supports are most appropriate for such workload. To achieve these goals, we implement a low-cost autonomous car platform as a case-study and systematically conduct experiments, which we will describe in the subsequent sections.

III. DeepPicar Overview

In this section, we provide an overview of our DeepPicar platform. In developing DeepPicar, one of our primary goals is to replicate the NVIDIA's DAVE-2 system on a smaller scale with using a low cost multicore platform, Raspberry Pi 3. Because the Pi 3's computing performance is much lower than that of the DRIVE PX platform used in DAVE-2, we are interested in if, and how, we can process computationally expensive neural network operations in real-time. Specifically, inferencing (forward pass processing) operations must be completed within each control period duration—e.g., a WCET of 50ms for 20Hz control frequency—locally on the Pi 3 platform, although training of the network (back-propagation for weight updates) can be done offline and remotely using a desktop computer.

Item	Cost ($)
Raspberry Pi 3 Model B	35
New Bright 1:24 scale RC car	10
Playstation Eye camera	7
Pololu DRV8835 motor hat	8
External battery pack & misc.	10
Total	70

TABLE I: DeepPicar's bill of materials (BOM)

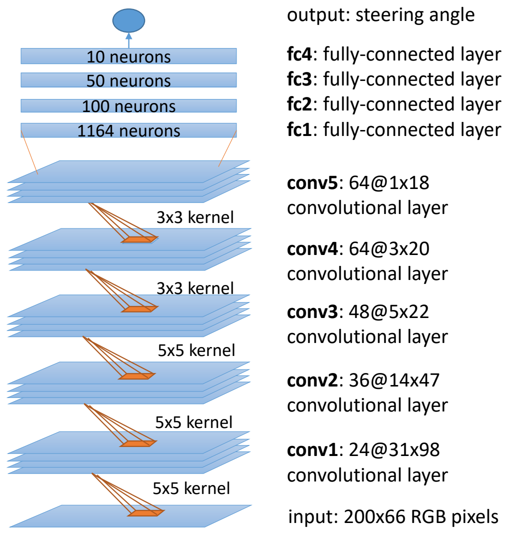
output: steering angle
10 neurons
50 neurons
100 neurons
1164 neurons
fc4: fully-connected layer
fc3: fully-connected layer
fc2: fully-connected layer
fc1: fully-connected layer
conv5: 64@1x18 convolutional layer
3x3 kernel
conv4: 64@3x20 convolutional layer
3x3 kernel
conv3: 48@5x22 convolutional layer
5x5 kernel
conv2: 36@14x47 convolutional layer
5x5 kernel
conv1: 24@31x98 convolutional layer
5x5 kernel
input: 200x66 RGB pixels

Fig. 3: DeepPicar's neural network architecture: 9 layers (5 convolutional, 4 fully-connected layers), 27 million connections, 250K parameters. The architecture is identical to the one used in NVIDIA's real self-driving car [3].

Figure 2 shows the DeepPicar, which is comprised of a set of inexpensive components: a Raspberry Pi 3 Single Board Computer (SBC), a Pololu DRV8835 motor driver, a Playstation Eye webcam, a battery, and a 1:24 scale RC car. Table I shows the estimated cost of the system.

For the neural network architecture, we adopt a TensorFlow version of NVIDIA DAVE-2's convolutional neural network (CNN), published by Dr. Fridman at MIT [1]. As in DAVE-2, the CNN takes a raw color image (200x66 RGB pixels) as input and produces a single steering angle value as output. Figure 3 shows the network architecture, which is comprised of 9 layers, 250K parameters, and about 27 million connections.

To collect the training data, a human pilot manually drives the RC car on a small track we created (Figure 4) to record timestamped videos and contol commands. The stored data is

[1]https://github.com/lexfridman/deeptesla

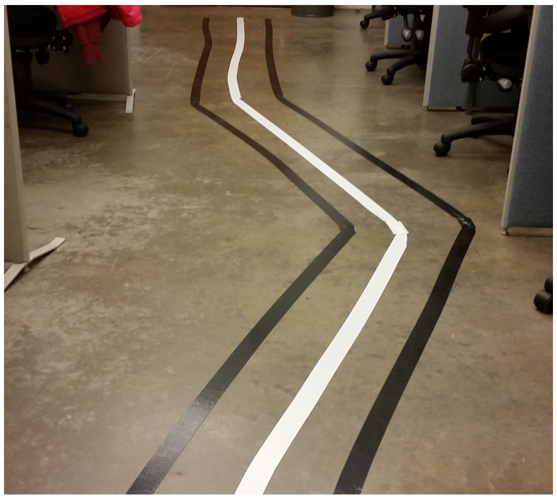

Fig. 4: One of the custom tracks used for training/testing.

```
while True:
  # 1. read from the forward camera
  frame = camera.read()
  # 2. convert to 200x66 rgb pixels
  frame = preprocess(frame)
  # 3. perform inferencing operation
  angle = DNN_inferencing(frame)
  # 4. motor control
  steering_motor_control(angle)
  # 5. wait till next period begins
  wait_till_next_period()
```

Fig. 5: Control loop

then copied to a desktop computer, which is equipped with a NVIDIA GTX 1060 GPU, where we train the network to accelerate training speed. For comparison, training the network on the Raspbeery Pi 3 takes approximately 4 hours, whereas it takes only about 5 minutes on the desktop computer using the GTX 1060 GPU.

Once the network is trained on the desktop computer, the trained model is copied back to the Raspberry Pi. The network is then used by the car's main controller, which feeds a image frame from the web camera as input to the network. In each control period, the produced steering angle output is then converted as the PWM values of the steering motor of the car. Figure 5 shows simplified pseudo code of the controller's main loop. Among the five steps, the 3rd step, network inferencing, is the most computationally intensive and dominates the execution time.

Note that although the steering angle output of the network $angle$ is a continuous real value, the RC car platform we used unfortunately only supports three discrete angles—left (-30°), center (0°), and right (+30°)—as control inputs. Currently, we approximate the network generated real-valued angle to the closest one of the three angles, which may introduce inaccuracy in control. In the future, we plan to use a different (more expensive) RC car platform that can precisely control

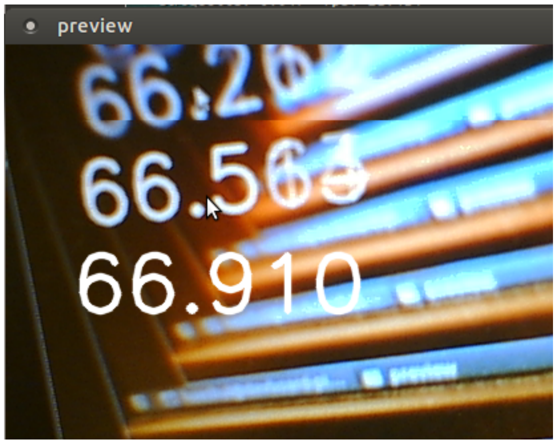

Fig. 6: Measuring DeepPicar's camera latency.

the car's steering angle. We would like to stress, however, that the use of different RC car platforms has no impact on the computational aspects of the system, and that our main focus of this study is not in improving network accuracy but in closely replicating the DAVE-2's network architecture and its real-time characteristics.

Another issue we observed in training/testing the network, which could affect the network performance, is camera latency. In DeepPicar's context, the camera latency is from the time the camera sensor observes the scene to the time the computer actually reads the digitized image data. Unfortunately, this time can be significantly long depending on the camera and the performance of the Pi. We experimentally measure DeepPicar's camera latency as follows [2]. First, we point the DeepPicar's camera towards a computer monitor, which is conntected to the Raspberry Pi 3's HDMI output. We then launch a program that overlays the current time with the camera's preview screen on the monitor. Then, an infinite number of time strings will be shown on the monitor screen and the time difference between two successive time strings represent the camera latency. Figure 6 shows the captured moniotor screen. The measured camera latency is about 300-350 ms. This is significantly higher than the latency of human perception, which is known to be as fast as 13 ms [17]. Higher camera latency could negatively affect control performance, because the DNN would analyze stale scenes. In the future, we plan to identify and use low-latency cameras.

Despite these issues, the trained models were still able to achieve a reasonable degree of accuracy, successfully navigating several different tracks we trained. The source code, build instruction, and a collection of self-driving videos of the DeepPicar can be found at: https://github.com/heechul/picar.

IV. Evaluation

In this section, we experimentally analyze various real-time aspects of the DeepPicar. This includes (1) measurement based worst-case execution time (WCET) analysis of deep

[2]We follow the method and the tool found at https://www.makehardware.com/2016/03/29/finding-a-low-latency-webcam/

Operation	Mean	Max	99pct.	Stdev.
Image capture	2.28	4.94	4.54	0.52
Image pre-processing	3.09	4.60	3.31	0.10
DNN inferencing	37.30	51.03	45.48	2.75
Total Time	42.67	56.37	50.70	2.80

TABLE II: Control loop timing breakdown.

learning inferencing, (2) the effect of using multiple cores in accelerating inferencing, (3) the effect of co-scheduling multiple deep neural network models, and (4) the effect of co-scheduling memory bandwidth intensive co-runners.

A. Setup

The Raspberry Pi 3 Model B platform used in DeepPicar equips a Boardcom BCM2837 SoC, which has a quad-core ARM Cortex-A53 cluster, running at up to 1.2GHz. Each core has 16K private I&D caches, and all cores share a 512KB L2 cache. The chip also includes Broadcom's Videocore IV GPU, although we did not use the GPU in our evaluation due to the lack of sofware support (TensorFlow is not compatible with the Raspberry Pi's GPU). For software, we use Ubuntu MATE 16.04, TensorFlow 1.1 and Python 2.7. We disabled DVFS (dynamic voltage frequency scaling) and configured the clock speed of each core statically at the maximum 1.2GHz.

B. Inference Timing for Real-Time Control

For real-time control of a car (or any robot), the control loop frequency must be sufficiently high so that the car can quickly react to the changing environment and its internal states. In general, control performance improves when the frequency is higher, though computation time and the type of the particular physical system are factors in determining a proper control loop frequency. While a standard control system may be comprised of multiple control loops with differing control frequencies—e.g., an inner control loop for lower-level PD control, an outer loop for motion planning, etc.—DeepPicar's control loop is a single layer, as shown earlier in Figure 5, since a single deep neural network replaces the traditional multi-layer control pipline. (Refer to Figure 1 on the differences between the standard robotics control vs. end-to-end deep learning approach). This means that the DNN inference operation must be completed within the inner-most control loop update frequency. To understand achievable control-loop update frequencies, we experimentally measured the execution times of DeepPicar's DNN inference operations.

Figure 7 shows the measured control loop processing times of the DeepPicar over 1000 image frames (one per each control loop). We omit the first frame's processing time for cache warmup. Table II shows the time breakdown of each control loop. Note that all four CPU cores of the Raspberry Pi 3 were used by the TensorFlow library when performing the DNN inference operations.

First, as expected, we find that the inference operation dominates the control loop execution time, accounting for about 85% of the execution time. Second, and more importantly, we also find that the measured average execution time of

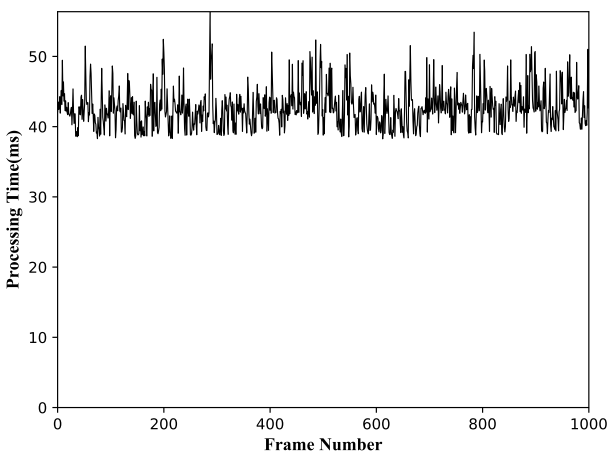

Fig. 7: DeepPicar's control loop processing times over 1000 input image frames.

a single control loop is 42.67 ms, or 23.4 Hz and the 99 percentile time is 50.70 ms. This means that the DeepPicar can operate at about a 20 Hz control frequency in real-time using only the on-board Raspberry Pi 3 computing platform, as no remote computing resources were necessary. We consider these results respectable given the complexity of the deep neural network, and the fact that the inference operation performed by TensorFlow only utilizes the CPU cores of the Raspberry Pi 3 (its GPU is not supported by Tensorflow).

In comparison, NVIDIA's DAVE-2 system, which has the exact same neural network architecture, reportedly runs at 30 Hz [3], which is just a bit faster than the DeepPicar. Although we believe it was not limited by their computing platform (we will experimentally compare performance differences among multiple embedded computing platforms, including NVIDIA's Jetson TX2, later in Section V), the fact that the low-cost Raspberry Pi 3 can achieve similar real-time control performance is surprising.

C. Effect of the Core Count to Inference Timing

In this experiment, we investigate the scalability of performing inference operations of DeepPicar's neural network with respect to the number of cores. As noted earlier, the Raspberry Pi 3 platform has four Cortex-A53 cores and TensorFlow provides a programmable mechanism to adjust how many cores are to be used by the library. Leveraging this feature, we repeat the same experiment described in the previous subsection with varying numbers of CPU cores—from one to four.

Figure 8 shows the average execution time of the control loop as we vary the number of cores used by TensorFlow. As expected, as we assign more cores, the average execution time decreases—from 61.96 ms on a single core to 42.67 ms on four cores (a 30% improvement). However, the improvement is far from an ideal linear scaling. In particular, from 2 cores to 3 cores, the improvement is mere 2.38 ms (or 4%). In short, we

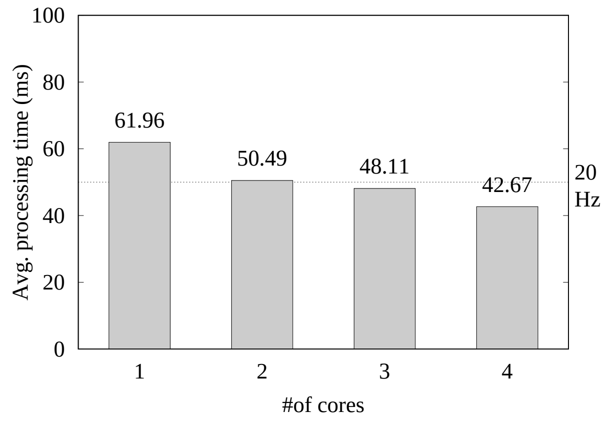

Fig. 8: Average control loop execution time vs. #of CPU cores.

find that the scalability of DeepPicar's deep neural network is not ideal on the platform. We do not know whether it is due to the limitations of TensorFlow's multicore implementation or if it's the model's inherent characteristics.

The poor scalability opens up the possibility of consolidating multiple different tasks or different neural network models rather than allocating all cores for a single neural network model. For example, it is conceivable to use four cameras and four different neural network models, each of which is trained separately and executed on a single dedicated core. Assuming we use the same network architecture for all models, then one might expect to achieve up to 15 Hz using one core (given 1 core can deliver 62 ms average execution time). In the next experiment, we investigate the feasibility of such a scenario.

D. Effect of Co-scheduling Multiple DNN Models

In this experiment, we launch multiple instances of DeepPicar's DNN model at the same time and measure its impact on their inference timings. In other words, we are interested in how shared resource contention affects inference timing.

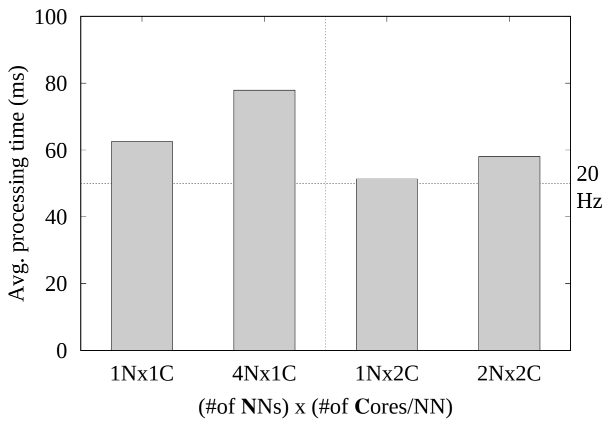

Fig. 9: Timing impact of co-scheduling multiple DNNs. 1Nx1C: one DNN model using one core; 4Nx1C: four DNN models each using one core; 1Nx2C: one DNN model using two cores; 2Nx2C: two DNN models each using two cores.

Figure 9 shows the results. In the figure, the X-axis shows the system configuration: #of DNN models x #of CPU cores/DNN. For example, '4Nx1C' means running four DNN models each of which is assigned to run on one CPU core, whereas '2Nx2C' means running two DNN models, each of which is assigned to run on two CPU cores. The Y-axis shows the average inference timing. The two bars on the left show the impact of co-scheduling four DNN models. Compared to executing a single DNN model on one CPU core (1Nx1C), when four DNN models are co-scheduled (4Nx1C), each model suffers an average inference time increase of approximately 15 ms, ∼24%. On the other hand, when two DNN models, each using two CPU cores, are co-scheduled (2Nx2C), the average inference timing is increased by about 7 ms, or 10%, compared to the baseline of running one model using two CPU cores (1Nx2C).

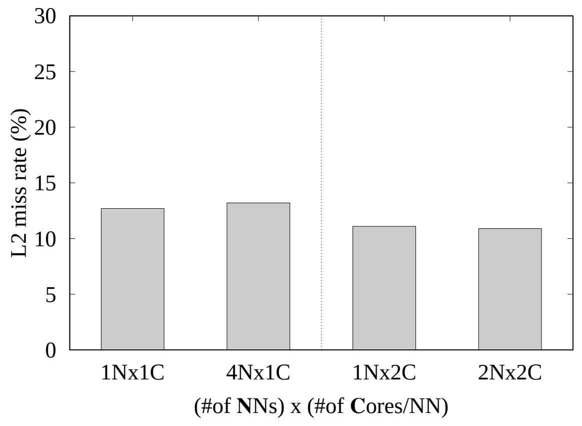

Fig. 10: L2 cache miss rates of different neural network and core assignments. X-axis is the same as Figure 9.

These increases in inference times in the co-scheduled scenarios are expected and likely caused by contention in the shared hardware resources, such as the shared L2 cache and/or the DRAM controller. To further analyze the source of contention, we use hardware performance counters of the processor. Specifically, we measure L2 miss rates of the DNN models first in isolation and then after co-scheduling other models. If the shared L2 cache is the primary source of inteference, then the measured L2 miss rates will increase. Figure 10 shows the results. As can be see in the figure, L2 miss rates are largely unchanged regardless of whether multiple models are co-scheduled or not. This suggests that the shared L2 cache is not the bottleneck that caused execution time increases. In other words, DNN models don't appear to be sensitive to the shared L2 cache space. Instead, we hypothesize that it is likely caused by the memory controller—the only other major shared hardware source—where memory requests from different CPU cores contend, which would result in increased memory access latency. While some Intel processors provide incore hardware counters that can measure average memory access latency [19], we were not able to identify whether such hardware counters exist in the BCM2837 processor of Raspberry Pi 3 due to the lack of documentation. Instead, in the next experiment, we use memory intensive synthetic benchmarks to test the hypothesis.

E. Effect of Co-scheduling Memory Performance Hogs

In order to determine how contended DRAM requests affect the DNN inference timing of the DeepPicar, we use a synthetic memory intensive benchmark from the IsolBench suite [18].

We run a single DNN model one core, and co-schedule an increasing number of the memory intensive synthetic benchmarks [3], on the remaining idle cores.

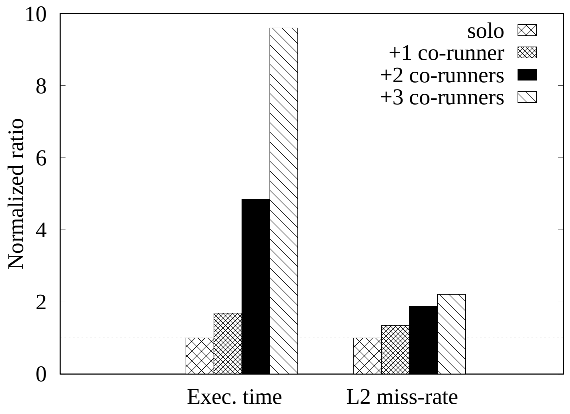

Fig. 11: Effect of memory performance hogs on the DNN inferencing. The DNN model uses Core 0 and memory-hog co-runners use the rest of the cores.

Figure 11 shows the normalized execution time and L2 miss-rate of the DNN model running on the Core 0 as a function of the number of co-scheduled memory intensive synthetic benchmarks. First, as we increase the number of co-runners, the DNN model's execution times are increased—by up to 9.6X—even though the DNN model is running on a dedicated core (Core 0). On the other hand, the DNN model's L2 cache-miss rates do not increase as much. This suggests that the DNN model's exeuction increase cannot be fully explained by increases in L2 cache-misses. Instead, as we hypothesized in the previous experiment, the increased memory pressure from the co-scheduled memory intensive benchmarks is likely the primary cause of the DNN model's execution time increase. Therefore, we conclude that DeepPicar's DNN model is more senstive to DRAM access latency than L2 cache space.

This observation suggests that shared cache partitioning techniques [5], [8] may not be effective isolation solutions for DeepPicar, as its AI workload is more sensitive to memory performance. Instead, memory controller focused isolation solutions, either hardware or software-based ones (e.g., [6], [21]), may be more important. Although our observation is made on a single hardware platform running on a single DNN workload, we suspect that many AI workloads may exhibit similar characteristics.

[3]We use the *Bandwidth* benchmark in the IsolBench suite, with the following command line parameters: `$ bandwdith -a write -m 16384`

F. Summary of the Findings

So far, we have evaluated DeepPicar's real-time characteristics from the perspective of end-to-end deep learning based real-time control, and made several observations.

First, we find that DeepPicar's computing platform, the Raspberry Pi 3 Model B, offers adquate computing capacity to perform real-time control of the RC car at 20 Hz frequency (or 50ms per control loop). Given the complexity of the DNN used, we were pleasantly suprised by this finding. The time breakdown shows that the DNN inferencing operation, performed by the Tensorflow library, dominates the execution time, which is expected.

Second, we find that scalability of Tensorflow's DNN implementation is limited. We find that using all four cores is only about 30% better than using just a single core.

Third, we find that consolidating multiple DNN models—on different CPU cores—is feasible as we find: (1) DNN performance using a single core is not much worse than using multiple cores; (2) multiple DNN models running simultaneously do not cause severe interference with each other.

Lastly, we find that consolidating memory (DRAM) performance intensive applications could jeopadize DNN performance, because DNN performance appears to be very sensitive to memory performance; we observe up to 9.6X slowdown in DNN performance by co-scheduling synthetic memory bandwidth intensive applications on idle cores.

V. EMBEDDED COMPUTING PLATFORM COMPARISON

In this section, we compare three computing platforms—the Raspberry Pi 3, the Intel UP [4] and NVIDIA Jetson TX2 [5]—from the point of view of supporting end-to-end deep learning based autonomous vehicles. Table III shows architectural features of the three platforms [6].

Our basic approach is to use the same DeepPicar software, and repeat the experiments in Section IV on each hardware platform and compare the results. For the Jetson TX2, we have two different system configurations, which differ in whether TensorFlow is configured to use its GPU or only the CPU cores. Thus, the total four system configurations are compared.

Figure 12 shows the average control loop completion timing of the four system configurations we tested as a function of the number of CPU cores used. First, both the Intel UP and Jetson TX2 exhibit superior performance when compared with the Raspberry Pi 3. When all four CPU cores are used, the Intel UP is 2.53X faster than the Pi 3, while the TX2 (CPU) and TX2 (GPU) are 3.6X and 7.6X times faster, respectively, than the Pi 3. As a result, they are all able to satisfy the 50 ms WCET by a clear margin, and, in case of TX2, 50 Hz or even 100 Hz real-time control is feasible with the help of its GPU. Another observation is that TX2 (GPU) does not change

[4]http://www.up-board.org/up/

[5]http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html

[6]The GPUs of the Raspberry Pi 3 and Intel Up are not used in evaluation due to the lack of software (TensorFlow) support. Also, the two Denver cores in Tegra TX2 are not used in evaluation due to TensorFlow issues.

Item	Raspberry Pi 3 (B)	Intel UP	NVIDIA Jetson TX2
CPU	BCM2837 4x Cortex-A53@1.2GHz/512KB L2	X5-Z8350 (Cherry Trail) 4x Atom@1.92GHz/2MB L2	Tegra X2 4x Cortex-A57@2GHz/2MB L2 2x Denver@2.0GHz/2MB L2 (not used)
GPU	VideoCore IV (not used)	Intel HD 400 Graphics (not used)	Pascal 256 CUDA cores
Memory	1GB LPDDR2	2GB DDR3L	8GB LPDDR4

TABLE III: Compared embedded computing platforms

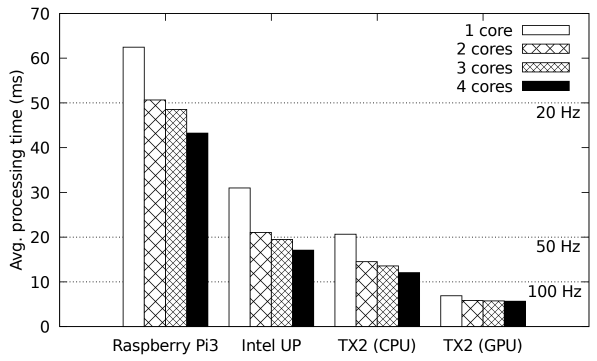

Fig. 12: Average control loop execution time.

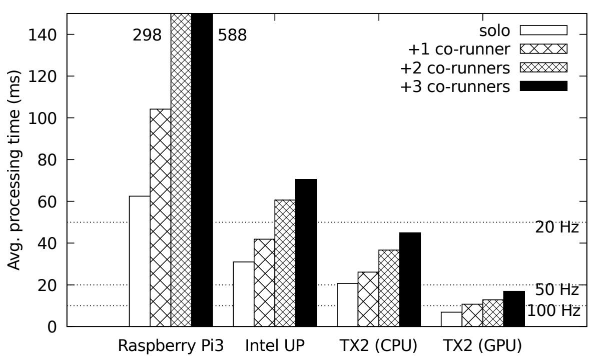

Fig. 14: Average control loop execution time in the presence of an increasing number of memory intensive applications on idle CPU cores.

much, as most of the neural network computation is done at the GPU.

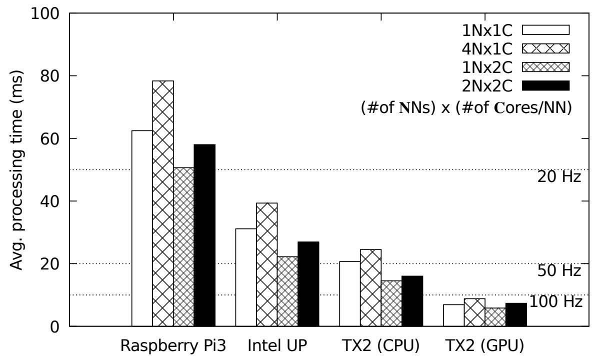

Fig. 13: Average control loop execution time when multiple DNN models are co-scheduled.

The Intel UP board and Jetson TX2 also perform much better when multiple DNN models are co-scheduled. Figure 13 shows the results of the multi-model co-scheduling experiment. Once again, they can comfortably satisfy 20 Hz real-time performance for all of the co-scheduled DNN control loops, and in the case of the TX2 (GPU), 100 Hz real-time control is still feasible. Given that the GPU must be shared among the co-scheduled DNN models, the results suggest that the TX2's GPU has sufficient capacity to accomodate multiple instances of the DNN model we tested.

Finally, we compare the effect of co-scheduling memory bandwidth intensive synthetic benchmarks on the DNN control loop timing. Figure 14 shows the results. As discussed in Section IV-E, we observed dramatic execution time increases, up to 9.4X, in Raspberry Pi 3 as we increased the number of co-scheduled tasks. We also observe increased control loop execution timing in the Intel Up and Jetson TX2, but the degree of the increase is not as dramatic as the Pi 3. Compared to their respective solo timings (i.e., the model runs on a single core in isolation), Intel UP suffers up to 2.3X execution time increase; TX2 (CPU) and TX2 (GPU) suffer up to 2.2X and 2.5X increases, respectively. This is somewhat suprising because the Raspberry Pi 3's cores are in-order architecture based while the cores in the Intel Up and NVIDIA TX2 are out-of-order architecture based, and that the memory intensive tasks on out-of-order cores can generate more memory traffic. We suspect that this is because the memory subsystems in the Intel UP and TX2 platforms provide higher bandwidth and fairness than the memory subsystem of the Pi 3.

Another interesting observation is that the TX2 (GPU) also suffers considerable execution time increase (2.5X) despite the fact that the co-scheduled synthetic tasks do not utilize the GPU. In other words, the DNN model has dedicated access to the GPU. This is, however, a known characteristic of integrated CPU-GPU architectue based platforms where both CPU and GPU share the same memory subsystem [2]. As a result, the TX2 (GPU) fails to meet the 10ms deadline for 100 Hz control that would have been feasible if there was no contention between the CPU cores and the GPU.

In summary, we find that today's embedded computing platforms, even as inexpensive as a Raspberry Pi 3, are powerful enough to support vision and end-to-end deep learning based real-time control applications. Furthermore, availability of CPU cores and GPU on these platforms allow consolidating mutiple deep neural network based AI workloads. However, shared resource contention among these diverse computing resources remains an important issue that must be understood and controlled, especially for safety-critical applications.

VI. Conclusion

We presented DeepPicar, a low cost autonomous car platform that is inexpensive to build, but is based on state-of-the-art AI technology: End-to-end deep learning based real-time control. Specifically, DeepPicar uses a deep convolutional neural network to predict steering angles of the car directly from camera input data in real-time. Importantly, DeepPicar's neural network architecture is identical to that of NVIDIA's real self-driving car.

Despite the complexity of the neural network, DeepPicar uses a low-cost Raspberry Pi 3 quad-core computer as its sole computing resource. We systematically analyzed the real-time characteristics of the Pi 3 platform in the context of deep-learning based real-time control appilcations, with a special emphasis on real-time deep neural network inferencing. We also evaluated other, more powerful, embedded computing platforms to better understand achievable real-time performance of DeepPicar's deep-learning based control system and the impact of computing hardware architectures. We find all tested embedded platforms, including the Pi 3, are capable of supporting deep-learning based real-time control, from 20 Hz up to 100 Hz, depending on the platform and its system configuration. However, shared resource contention remains an important issue that must be considered in applying deep-learning models on shared memory based embedded computing platforms.

As future work, we plan to apply shared resource management techniques [21], [20] on the DeepPicar platform and evaluate their impact on the real-time performance of the system. We also plan to improve the prediction accuracy by feeding more data and upgrading the RC car hardware platform to enable more precise steering angle control.

References

[1] F1/10 autonomous racing competition. http://f1tenth.org.

[2] W. Ali and H. Yun. Work-in-progress: Protecting real-time GPU applications on integrated CPU-GPU SoC platforms. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 141–143, 2017.

[3] M. Bojarski et al. End-to-End Learning for Self-Driving Cars. *arXiv:1604*, 2016.

[4] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 9:249–256, 2010.

[5] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Computing Surveys*, 48(2):1–36, nov 2015.

[6] D. Guo and R. Pellizzoni. A requests bundling DRAM controller for mixed-criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 247–258. IEEE, apr 2017.

[7] N. P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, 2017.

[8] N. Kim, B. C. Ward, M. Chisholm, C.-y. Fu, J. H. Anderson, and F. D. Smith. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *Real-Time Technology and Applications Symposium, IEEE*. IEEE, 2016.

[9] Y. Lecun, E. Cosatto, J. Ben, U. Muller, and B. Flepp. DAVE: Autonomous off-road vehicle control using end-to-end learning. Technical Report DARPA-IPTO Final Report, Courant Institute/CBLL, http://www.cs.nyu.edu/˜yann/research/dave/index.html, 2004.

[10] S. Levine. Deep Reinforcement Learning. http://rll.berkeley.edu/deeprlcourse/f17docs/lecture_1_introduction.pdf, 2017.

[11] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 39:1–40, 2016.

[12] NVIDIA. The AI Car Computer for Autonomous Driving. http://www.nvidia.com/object/drive-px.html.

[13] NVIDIA. GPU-Based Deep Learning Inference : A Performance and Power Analysis. Technical Report November, 2015.

[14] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 353–364. IEEE, apr 2017.

[15] D. a. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in Neural Information Processing Systems 1*, pages 305–313, 1989.

[16] R. Shin, S. Karaman, A. Ander, M. T. Boulet, J. Connor, K. L. Gregson, W. Guerra, O. R. Guldner, M. Mubarik, B. Plancher, et al. Project based, collaborative, algorithmic robotics for high school students: Programming self driving race cars at mit. Technical report, MIT Lincoln Laboratory Lexington United States, 2017.

[17] Thomas Burger. How Fast Is Realtime? Human Perception and Technology — PubNub, 2015.

[18] P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.

[19] Y. Ye, R. West, J. Zhang, and Z. Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 179–190. IEEE, 2016.

[20] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[21] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

Appendix

A. DNN Training and Testing

We have trained and tested the deep neural network with several different track conditions, different combinations of input data, and different hyper parameters. In the following paragraphs, we describe details on two of the training methods that performed reasonably well.

In the first method, we trained the neural network model across a set of 30 completed runs on the track seen in Figure 4 by a human pilot. Half of the runs saw the car driving one way along the track, while the remaining half were of the car driving in the opposite direction on the track. In total, we collected 2,556 frames for training and 2,609 frames for validation. The weights of the network are initialized using a Xavier initializer [4], which is known to provide better initial values than the random weight assignment method. In each training step, we use a batch size of 100 frames, which are randomly selected among all the collected training images, to optimize the network. We repeat this across 2,000 training steps. When a model was trained with the aformentioned data, the training loss was 0.0188 and the validation loss was

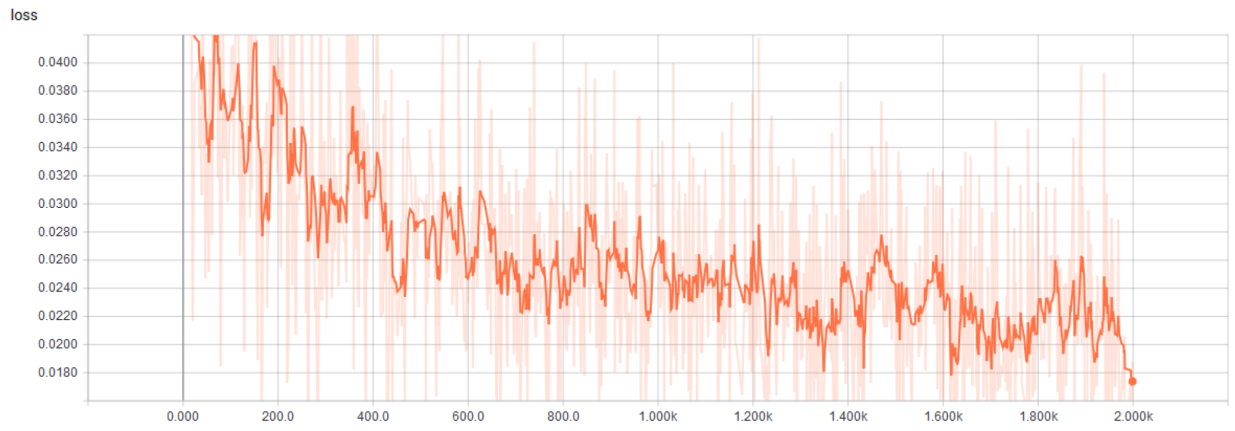

Fig. 15: Change in loss value throughout training.

0.0132. The change of the loss value over the course of model training can be seen in Figure 15.

In the second method, we use the same data and parameters as above except that now images are labeled as 'curved' and 'straight' and we pick an equal number of images from each category at each training step to update the model. In other words, we try to remove bias in selecting images. We find that the car performed better in practice by applying this approach as the car displayed a greater ability to stay in the center of the track (on the white tape). However, we find that there is a huge discrepency between the training loss and the validation loss as the former was 0.009, while the latter was 0.0869—a 10X difference—indicating that the model suffers from an overfitting problem.

We continue to investigate ways to achieve better prediction accuracy in training the network, as well as improving the performance of the RC car platform, especially related to precise steering angle control.

B. System-level Factors Affecting Real-Time Performance

In using the Raspberry Pi 3 platform, there are a few system-level factors, namely power supply and temperature, that need to be considered to achieve consistent and high real-time performance.

In all our experiments on the Raspberry Pi 3, the CPU operated at a preferred clock speed of 1.2 GHz. However, without care, it is possible for the CPU to operate at a lower frequency.

An important factor is CPU thermal throttling, which can affect CPU clock speed if the CPU temperature is too high (Pi 3's firmware is configured to throttle at 85C). DNN model operations are computationally intensive, thus it is possible for the temperature of the CPU to rise quickly. This can be especially problematic in situations where multiple DNN models are running simultaneously on the Pi 3. If the temperature reaches the threshold, the Pi 3's thermal throttling kicks in and decreases the clock speed down to 600MHz—half of the maximum 1.2GHz—so that the CPU's temperature stays at a safe level. We found that without proper cooling solutions (heatsink or fan), prolonged use of the system would result in CPU frequency decrease that may affect evaluation.

Another factor to consider is power supply. From our experiences, the Pi 3 frequency throttling also kicks in when the power source can not provide the required minimum of 2A current. In experiments conducted with a power supply that only provided 1 Amp, the Pi was unable to sustain a 1.2 GHz clock speed, and instead, fluctuated between operating at 600 MHz and 1.2 GHz. As a result, it is necessary, or at least highly recommended, that the power supply used for the Raspberry Pi 3 be capable of outputting 2 Amps, otherwise optimal performance isn't guaranteed.

Our initial experiment results suffered from these issues, after which we always carefully monitored the current operating frequencies of the CPU cores during the experiments to ensure the correctness and repeatability of the results.