DeepPicarMicro: Applying TinyML to Autonomous Cyber Physical Systems

Michael Bechtel, QiTao Weng, Heechul Yun
University of Kansas, USA.
{mbechtel, wengqt, heechul.yun}@ku.edu

***Abstract*—Running deep neural networks (DNNs) on tiny Micro-controller Units (MCUs) is challenging due to their limitations in computing, memory, and storage capacity. Fortunately, recent advances in both MCU hardware and machine learning software frameworks make it possible to run fairly complex neural networks on modern MCUs, resulting in a new field of study widely known as TinyML. However, there have been few studies to show the potential for TinyML applications in cyber physical systems (CPS).**

In this paper, we present DeepPicarMicro, a small self-driving RC car testbed, which runs a convolutional neural network (CNN) on a Raspberry Pi Pico MCU. We apply a state-of-the-art DNN optimization to successfully fit the well-known PilotNet CNN architecture, which was used to drive NVIDIA's real self-driving car, on the MCU. We apply a state-of-art network architecture search (NAS) approach to find further optimized networks that can effectively control the car in real-time in an end-to-end manner. From an extensive systematic experimental evaluation study, we observe an interesting relationship between the accuracy, latency, and control performance of a system. From this, we propose a joint optimization strategy that takes both accuracy and latency of a model in the network architecture search process for AI enabled CPS.

***Index Terms*—Real-time, Autonomous Car, Convolutional Neural Network, Microcontroller, Case Study, TinyML**

I. Introduction

Autonomous cyber physical systems (CPS), such as self-driving cars and drones, are a topic with much interest in recent years. The premise is that by employing recent advances in machine learning (ML) algorithms such as deep neural networks (DNNs), CPS can become more intelligent and safer, which benefits society.

However, executing DNN models is computationally expensive. This limits their applicability to many CPS with significant size, weight, power, cost, and real-time constraints. Therefore, there are increasing research efforts to reduce the computational requirements of employing DNN models. In particular, many researchers and companies are putting significant effort to support DNNs in tiny micro-controller units (MCUs) due to their low cost and low power consumption, despite their obvious limitations in terms of available computing and memory resources [3], [4], [20], [21].

In this paper, we present DeepPicarMicro, a low-cost autonomous car testbed to study the feasibility of AI enabled CPS on tiny MCUs. DeepPicarMicro employs an end-to-end deep learning approach, which utilizes a convolutional neural network (CNN) to directly control the physical plant from the camera based sensory input as in our prior work DeepPicar [5]. The main difference is that DeepPicarMicro utilizes a Raspberry Pi Pico MCU (a dual Cortex-M0+ MCU) as the main computing platform, which is significantly less capable than the Raspberry Pi 3/4 computing platforms used in the DeepPicar. Note that DeepPicar's Raspberry Pi 3/4 platforms were capable enough to run the full unmodified PilotNet model [5], which was used in NVIDIA's real self-driving car [7], in real-time. Table I shows the differences between a Raspberry Pi 4 and a Raspberry Pi Pico MCU, the latter of which features orders of magnitude smaller computing power and memory/storage availability.

Part	Raspberry Pi 4	Raspberry Pi Pico
CPU	BCM2837 4x Cortex-A72@1.5GHz	RP2040 2x Cortex-M0+@133MHz
Memory	48BK(I)/32KB(D) L1 cache 1MB L2 (16-way) L2 cache 4GB LPDDR4	264KB SRAM
Storage	8GB+ micro-SD	2MB Flash
Power	3A	<100mA

TABLE I: Comparison of hardware resources on a Raspberry Pi 4 microprocessor and a Raspberry Pi Pico MCU.

Using DeepPicarMicro, we want to answer the following questions: (1) Can we run a full-sized PilotNet on a micro-controller? (2) How can we find optimized neural network architectures for a target micro-controller? (3) What are the relationships between accuracy, latency, and control performance of an end-to-end DNN model in controlling CPS?

From an extensive systematic experimental evaluation study, we made the following observations. First, to our surprise, we find that the full sized PilotNet can run on a Pico MCU using a specialized ML framework, namely Tensorflow-lite micro (TFLM) [14], and standard optimization techniques such as 8-bit quantization. However, the unmodified (except quantization) PilotNet model's latency was more than 3 seconds, which is not acceptable for real-time control of CPS. Clearly, there is a strong need to further optimize the network to be able to run on a tiny MCU.

Second, we apply a state-of-the-art neural architecture search (NAS) approach [21] to find smaller variants of the PilotNet by varying the input resolution, depth, width of the full model. In addition, we also employed depthwise-separable convolutions [24] in place of the standard 2D convolutions to further reduce the latency of the models. As a result, we found many PilotNet variants that meet the real-time constraints of

the system and achieve high accuracy. Interestingly, however, we observe that less accurate DNN models with lower latency often performed better in practice than more accurate models with higher latency. Even when we compare similarly accurate models, we observe that lower latency models perform better, even if we set the control frequency of the models to be identical (all meeting the same deadline). This is because the model's latency affects the reaction time of the CPS system it controls and the quality of the network's output degrades as the network's input becomes stale. This suggests that in a CPS system, not only a network's accuracy but also its latency must be taken into account to predict the model's true performance. Therefore, the standard NAS approach that treats latency as a constraint may not be ideal to find best performing models.

Third, we evaluate a simple joint optimization strategy, which uses a normalized sum of the DNN model's latency and accuracy to compare a model's predicted performance in a real CPS system. In both simulation and in real-world experiments, we find the joint optimization strategy is effective in predicting a network's real-world performance in controlling the RC car.

In summary, we make the following contributions:

- We present DeepPicarMicro, a MCU-based autonomous car testbed that employs a CNN-based end-to-end real-time control loop.
- We present extensive experimental evaluation results showing the possibility of using MCU for AI enabled CPS.
- We propose a simple joint optimization strategy that takes both accuracy and latency of a model in the network architecture search process for AI enabled CPS.

The remainder of the paper is organized as follows. Section II provides a background on MCUs and TinyML. Section III gives an overview of the DeepPicarMicro testbed and our initial evaluations with it. Section IV describes the first NAS approach we use for finding a TinyML model that can run on the DeepPicarMicro. Section V presents extensive CPS control performance evaluation results on a real-world environment, in addition to our modified NAS approach. We review related work in Section VI and conclude in Section VII.

II. Background

In this section, we provide background on autonomous vehicles, MCUs, and TinyML.

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

Self-driving cars have been a topic of increasing interest over the past several years. A standard approach is to split the task into multiple specialized sub tasks, such as planning and perception [17]. On the other hand, an end-to-end deep learning approach uses a single neural network to produce control outputs directly from the raw sensor input data, which dramatically simplifies the control pipeline [19]. First introduced in 1989 by Pomerleau [23], many systems have since employed DNN-based control loops to much success [1], [5], [7], [15], [16].

In a DNN based end-to-end control loop based system, training and inference are typically performed separately. In general, training a neural network model is computationally expensive, so it is often done on more powerful PC systems equipped with hardware accelerators (e.g. GPUs). On the other hand, inference operations require relatively less computing power and can thus be run on smaller embedded platforms. However, on such platforms, the model's inference latency becomes an important factor as many systems and applications have real-time constraints. In this paper, we explore the capability of executing a deep neural network on a small microcontroller platform in real-time.

B. Microcontroller Units (MCUs)

A microcontroller is a small computer that integrates simple CPU core, SRAM and flash memory into a single integrated chip. MCUs are inexpensive and consume very little power, often in the range of milliamps (mA). As such, they are used in a wide variety of applications, ranging from toys to cars. Unlike powerful microprocessors, which typically employ complex operating systems and other runtime frameworks to perform sophisticated tasks, MCUs are designed for relatively simple tasks and often do not employ operating systems. This allows MCUs to have far more predictable temporal behaviors than microprocessors, as they do not suffer from non-determinism typically seen in standard OSes (e.g. virtual memory, page faults, etc.). On the other hand, MCUs have very limited computing, memory, and storage capacity, which pose a challenge for complex applications that require large amount of resources, such as machine learning algorithms.

C. Tiny Machine Learning (TinyML)

Recently, there are increasing interests and effort to enable ML applications in MCUs, which is collectively known as Tiny Machine Learning or TinyML for short [3], [4], [20], [21]. In TinyML, a major goal is to execute machine learning algorithms, such as DNN models, locally on an MCU, instead of relying on communications with larger PC or cloud that requires high energy consumption and suffers long latency. Lately, major MCU vendors as well as big tech companies such as Google have developed machine learning frameworks specially tailored for MCUs.

In this paper, we primarily use the Tensorflow Lite Micro (TFLM) deep learning framework, which is optimized for neural network inference for MCUs [14]. TFLM uses a runtime interpreter architecture for portability and supports a wide range of MCUs, including the MCU we used in this study. TFLM supports 8-bit quantified models, which are converted to C char arrays to be directly compiled for the target MCU. To efficiently utilize the limited memory in a MCU, TFLM uses a single pool of statically allocated memory called an "arena" that holds intermediate buffers and computations [2]. The size of the arena it can allocate determines the maximum activation size of the neural network model it can support. In this paper, we utilize the TFLM framework to execute a CNN model that controls a small-scale RC car autonomously.

III. DeepPicarMicro

DeepPicarMicro is a small self-driving RC car that employs an end-to-end deep learning approach utilizing a deep convolutional neural network (CNN) to directly control the motors from the raw camera input data. Such an end-to-end learning approach has been demonstrated in many prior works, including NVIDIA's real self-driving car DAVE-2 [7] as well as in our prior work DeepPicar [5]. Our main difference compared to all other prior works is that we realized this CNN based end-to-end control system on a tiny MCU.

A. Hardware Platform and Track

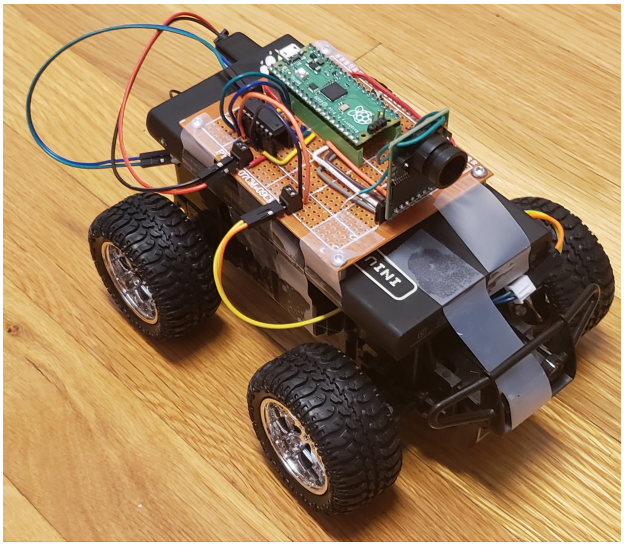

Fig. 1: DeepPicarMicro platform.

Component	DeepPicar	DeepPicarMicro
Compute platform	Raspberry Pi 3/4	Raspberry Pi Pico
Car platform	New Bright 1:24 scale RC car	New Bright 1:24 scale RC car
Camera	Playstation Eye	Arducam Mini 2MP Plus
Motor control	Pololu DRV8835	L293D
Power source	External battery pack	External battery pack

TABLE II: Hardware components used in DeepPicar vs DeepPicarMicro

Figure 1 shows the DeepPicarMicro, which is comprised of the following components: a Raspberry Pi Pico MCU, a L293D motor driver, an Arducam Mini 2MP Plus, an external battery pack, and a 1:24 scale RC car. Table II shows a comparison of the hardware used in the DeepPicar and DeepPicarMicro platforms. The camera used on the DeepPicarMicro is able to capture images at a frequency of 7.5Hz (∼133 ms per image), so we use this as our control frequency. In other words, our control loop has a deadline of 133ms per iteration. Note that the deadline for control systems is often derived from the system's dynamics, though in our case it is determined by hardware limitations. In addition, the camera can only capture images where both the width and height are multiples of four, so we modify all CNN models we find to fit this constraint before loading them onto the DeepPicarMicro.

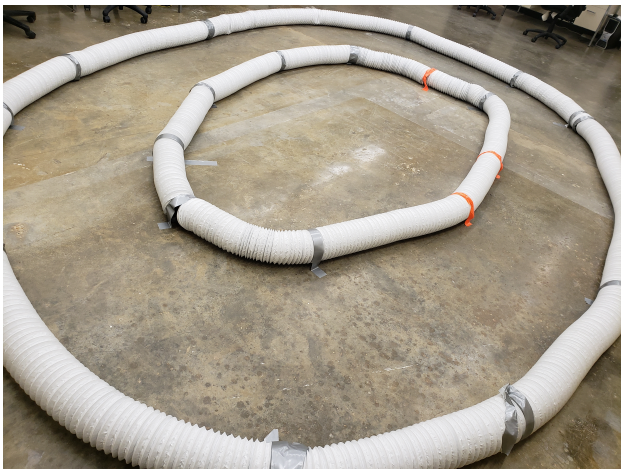

Fig. 2: Real-world track used to collect training data and evaluate the control performance of the DeepPicarMicro.

For the environment, we construct a simple track, shown in Figure 2, that we use for all of our real-world evaluations in this paper.

B. The PilotNet Architecture

For the neural network architecture, we begin by implementing NVIDIA's PilotNet CNN in TensorFlow using the Keras API. After we train a model, we then use the Tensorflow Lite Micro (TFLM) framework to load and execute the model on the DeepPicarMicro. Additionally, we use integer 8-bit quantization-aware training as the TFLM framework only supports quantized models at the time of writing.

Layer	Input size	Output size	Weights	MACs
Conv1	66x200x3	31x98x24	1.8K	5.5M
Conv2	31x98x24	14x47x36	21.6K	14.2M
Conv3	14x47x36	5x22x48	43.2K	4.8M
Conv4	5x22x48	3x20x64	27.7K	1.7M
Conv5	3x20x64	1x18x64	36.9K	663.6K
FC1	1152	100	115.3K	115.2K
FC2	100	50	5.1K	5K
FC3	50	10	510	500
FC4	10	1	11	51
Total			252.2K	26.9M

TABLE III: PilotNet [7] architecture

C. Data Collection, Pre-processing and Training

We manually collect a dataset of 10,000 frame and steering angle pairs around the track, which we will henceforth refer to as the DeepPicarMicro dataset[1]. We categorize all steering angles to one of three output classes (left, center, and right) to match the discrete control output space of the DeepPicarMicro. We use 7,500 pairs for training and the remaining 2,500 pairs

[1]We used the DeepPicar platform for data collection as the current iteration of the DeepPicarMicro does not have capability to store the collected data.

for validation. To improve the consistency of the training process we also employ the following techniques:

- When generating the train and validation sets, we use a constant seed such that the output sets are the same every time they are generated.
- We stratify the train and validation sets so that they are both equally proportionate to the output class distribution of the overall dataset.
- In our dataset, the majority of the samples are of the car going straight. As such, we perform class balancing so that all three output classes have an equal effect in the changes made to the model's final weight values. Specifically, we assign higher weight values to the left and right output classes and a lower weight to the straight output class.

In terms of hardware, the cameras used for the DeepPicar and DeepPicarMicro differ in their image capture properties (e.g. zoom, etc.). As a result, the two cameras will capture different images for the same scene. To account for this during training, we perform an additional translation scheme when pre-processing the input frames. To be more specific, for each image captured on the DeepPicar, we flip the image and crop it such that the final image closely resembles one captured from the DeepPicarMicro's camera.

D. Platform Resource Constraints

To evaluate the performance of the PilotNet model, we initially test three important metrics: memory usage, accuracy, and inference latency. We begin with the memory utilization of the network to determine whether PilotNet with 8-bit quantization can fit on the Pico MCU. We perform a theoretical analysis of PilotNet's per-layer memory usage by calculating the total input and output activation buffer sizes.

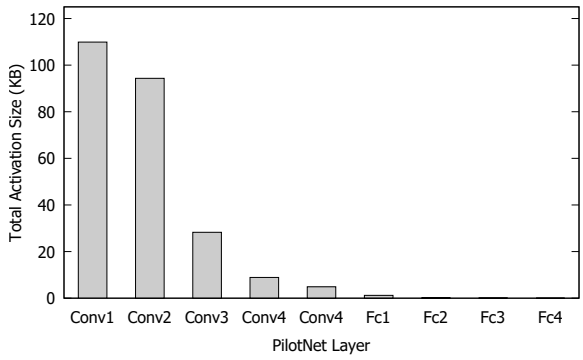

Fig. 3: SRAM requirement for each PilotNet layer.

Figure 3 shows the results of this analysis. Importantly, we find that the largest layer only requires 110KB which easily fits inside the 264KB SRAM available on the Pico MCU. Furthermore, because the TFLM Arena only needs to be as big as the largest layer [2], we find that the full PilotNet model can successfully be run on the Pico. We also find that the initial layers of the network require far more memory than the remainder of the network, which is consistent with prior studies [20].

E. Quantization, Accuracy and Latency

To test the model's accuracy, we re-feed all 2,500 samples from the validation set to the model. We again categorize the model's predictions to be a left, center, or right output. We then compare the predictions for all images to their respective ground truth outputs and measure the number of samples where the two matched (i.e. the model's prediction was "correct"). Using this method, we test the accuracy for the full PilotNet model, both with and without quantization enabled. Without quantization, PilotNet achieves an accuracy of *87.6%*, whereas it has an accuracy of *86.9%* with 8-bit quantization. From this, we find that PilotNet can be quantized and run on the DeepPicarMicro while still achieving comparable accuracy to original 32-bit floating point model. However, we find that the performance is highly undesirable as it takes over *3 seconds* to process each frame.

F. Depth-wise Separable Convolutional Layer

Based on PilotNet's temporal performance, there is little chance it would achieve good control performance. Even though it can run on the Pico with high accuracy, it would still face significant issues in reacting to external stimuli before the system fails (e.g. the car crashes). To address this, we first try to replace the standard 2D convolutional layers (i.e. Conv2D) with well-known *Depthwise Separable* layers [13], [24] to reduce the total multiply-accumulate (MAC) operations. As adopted in many recent TinyML architectures, depthwise separable layers significantly reduce network MACs by separating Conv2D layers into two different operations: (1) a depthwise convolution, and (2) a pointwise convolution. This reduces the computational cost of a convolution from

$$C * O_h * O_w * O_d * K^2 \tag{1}$$

to

$$C * O_h * O_w * K^2 + C * O_h * O_w * O_d \tag{2}$$

where C denotes input channels, O denotes output dimensions, and K denotes the kernel size. For PilotNet, we replace all five Conv2D layers with equivalent depthwise separable layers and train a new model on the DeepPicarMicro dataset. We henceforth refer to both PilotNet models as either the Conv2D model or Depthwise model, based on the type of convolution they employ. We then perform the same accuracy and inference latency measurement tests for the Depthwise model.

	Conv2D	**Depthwise**
Weights	252.2K	133.7
MACs	26.9M	2.1M
Val. Loss	0.027	0.032
Accuracy (%)	86.9	85.7
Latency (ms)	3025	525

TABLE IV: Comparison of PilotNet models with Conv2D and depthwise separable layers.

Table IV shows the model characteristics for the Conv2D and Depthwise models. Notably, the Depthwise model has ∼12.7X fewer MACs and ∼5.8X faster inference latency compared to the original Conv2D model. At the same time, both models have comparable accuracy, with the Depthwise model's accuracy only being ∼1% smaller.

However, we still find the Depthwise model's performance to be unsatisfactory. Even with fewer MACs, it still takes the Depthwise model *>500 ms* to process a single frame, which is greater than the target 133 ms control period. Due to this, we next explore the potential for reducing PilotNet's size without overly sacrificing accuracy. For this, we perform a Neural Architecture Search (NAS).

IV. Neural Architecture Search

In this section, we describe the NAS approach we perform on the PilotNet architecture. For the NAS, our goal is to find the model with the highest accuracy while also satisfying the different physical and temporal constraints required by the DeepPicarMicro. In particular, we hold that the model must be small enough to fit in the Pico MCU's SRAM and Flash and that its inference latency be <133 ms, the DeepPicarMicro's control period. Importantly, due to the performance gains seen in Table IV, we perform our NAS on the PilotNet model with depthwise separable layers.

A. Latency Prediction

While we can calculate the SRAM and Flash usage for a given model layout, the same can not be said for its inference latency. That is, without profiling a model to measure its latency, we can not directly determine if it meets the 133 ms constraint. However, prior studies have shown that a model's MAC operations corresponds to its inference latency [3]. To find this relationship for the Pico MCU, we run 50 CNN models with MAC operations ranging from ∼54.4K to ∼2.1M, and measure their respective inference latencies.

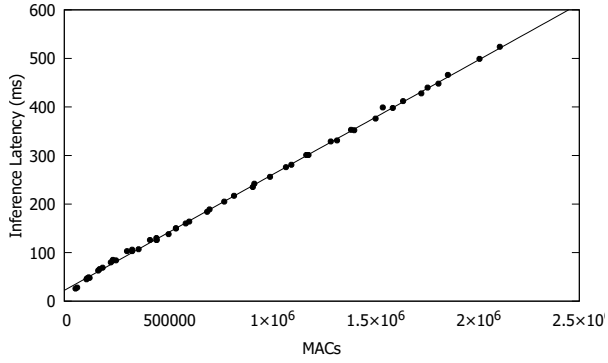

Fig. 4: Number of MAC operations vs. inference latency on the Raspberry Pi Pico. We use Linear Regression to find the dotted line of best fit (R^2=0.9996).

$$Latency = 0.000236 * MACs + 22.189388. \quad (3)$$

Figure 4 shows the inference latencies for each model on the Pico MCU. Using these data points, we use Linear Regression to derive Equation 3, which can be used to predict a model's latency. For example, with this equation an inference latency of 133 ms on the Pico MCU roughly correlates to ∼470,000 MACs. Note that this function would not work for models with Conv2D layers because they have have different numbers of operations per convolution overhead [3].

B. Search Space

Now that we can estimate a model's inference latency, we perform the NAS on PilotNet. We employ a NAS methodology largely influenced by the state-of-the-art MCUNet approach [21]. That is, we vary PilotNet's architectural properties that affect the total number of MAC operations. We refer to these properties as *reduction parameters*. However, compared to the MCUNet NAS, we substantially limit the number of network layouts we search to reduce the total execution time of the NAS. Using PilotNet with depthwise separable layers as a backbone, we define a search space with the following two reduction parameters:

- *Width* multiplier for all layers, ranging from [0.2, 0.4, 0.7, 0.8, 0.9, 1.0].
- Network *Depth*, ranging from a minimum of three layers to maximum of nine layers. We always keep the input convolutional and output fully-connected layers, but vary all unique combinations of the seven middle layers with at least one additional convolutional layer. In total, we evaluate 120 different network layouts in this parameter.

In addition, we configure all network layouts to have an input resolution size of 68x68x1. We choose this resolution as it is the smallest that can (1) be used for the full unaltered PilotNet, and (2) be captured by the DeepPicarMicro's camera, as discussed in Section III. Using this search space, we then perform a two-step NAS. First, we construct a model for every network layout in the search space and calculate its number of MAC operations. If the model has ≤470,000 MACs then we keep that model, otherwise we discard it. In total, we search 720 different network layouts in the first step, and keep 349 of them. In the second step, we train all remaining models on the DeepPicarMicro dataset. For each model, we attempt to train it up to five times to account for random weight initialization. To optimize this process, we define two validation loss thresholds: a target threshold and a fail threshold. At the end of each training iteration, we perform two checks on the model's current validation loss. If the validation loss is less than the target threshold, or greater than the fail threshold, then we do not perform any additional training iterations and keep the current model. Likewise, if the model is trained five times without passing either check, then we stop and move on to the next model. In our NAS, we use a target threshold of 0.0350, which roughly correlates to 80% accuracy, and a fail threshold of 0.0450. In the end, we found substantial variance in the validation losses of the 349 models we train, from 0.0287 to 0.0836. In terms of accuracy, this resulted in a range of 62.6% to 86.6% accuracy.

Model #	Layers	Width	Weights	MACs	Latency (ms)	Val. Loss	Accuracy (%)	Score	Laps w/o Crash
1	4	0.2	3.0K	64.8K	37	0.031	84.9	0.04	7
2	7	0.2	26.8K	151.3K	58	0.031	85.0	0.27	9
3	8	0.7	8.8K	267.6K	85	0.032	85.2	0.50	7
4	3	0.7	5.9K	214.5K	73	0.042	82.1	0.56	7
5	6	0.2	1.0K	70.5K	39	0.060	75.3	0.58	0
6	5	0.9	13.3K	358.0K	107	0.033	83.8	0.75	4
7	7	0.8	82.7K	371.4K	110	0.032	85.1	0.84	5
8	4	0.4	1.6K	128.1K	52	0.073	67.7	0.92	0
9	7	0.8	9.0K	315.8K	97	0.050	78.2	0.96	0
10	5	0.8	5.9K	312.8K	96	0.052	76.5	0.98	2
11	3	0.2	1.6K	63.5K	37	0.084	65.6	1.00	0
12	5	0.4	4.6K	276.4K	87	0.057	75.0	1.03	1
13	6	0.9	53.8K	421.3K	122	0.044	80.4	1.16	2
14	6	0.7	7.0K	265.5K	85	0.072	69.8	1.23	0
15	3	0.4	23.8K	374.2K	111	0.066	72.8	1.56	0
16	6	0.9	10.7K	378.2K	111	0.083	62.7	1.71	0

TABLE V: Model statistics and performance for each real-world test case, in order of increasing heuristic score. All models have the same input resolution of 68x68x1, and the latency values are calculated using Equation 3.

C. Performance Prediction

In the process of searching for an optimal TinyML model, most state-of-the-art NAS approaches will optimize their search on a performance based metric, such as accuracy. However, in the context of CPS, it has been shown that inference latency can also have a notable impact on control performance [22], [29]. Based on these findings, we propose a joint optimization strategy to better predict the control performance of the CNN models we found. In our strategy, we assign a heuristic score to each model that is calculated as follows:

$$Score = norm(ValLoss) + norm(Latency) \quad (4)$$

In this function, we normalize the validation loss and inference latency values for all of the models to be between 0 and 1. For each model, we then sum the two normalized values together to get a heuristic score between 0 and 2. With this strategy, the intuition is that models with relatively smaller validation losses and inference latencies often perform better. Therefore, the smaller a model's heuristic score is, the better it should perform. Note that we use estimated latencies based on Equation 3 when generating the heuristic scores.

Now that we have CNN models that can effectively run on the DeepPicarMicro in real-time, we next test their control performance in a real-world environment, as well as the effectiveness of our joint optimization strategy.

V. Evaluation

In this section, we evaluate the control performance of the CNN models from NAS both in a real track and in a simulated track.

A. Performance in Real Track

To begin, we perform a more in-depth evaluation of the models found in Section IV. As it would be time consuming and inefficient to test all 349 models, we select a sample subset of 16 models with differing validation losses and inference latencies. Using Equation 4, we calculate the heuristic scores for each model. We next evaluate the real-world control performance of each model. For this, we use the DeepPicarMicro testbed on our handmade track from Figure 2. For each CNN model, we attempt to run it on the DeepPicarMicro for ten individual laps and measure the total number of laps the car is able to finish without crashing.

Table V shows the statistics, heuristic scores, and laps completed for each tested model. As expected that some CNN models performed very well (green colored) and completed the majority of laps without crashing while some models perform poorly (red colored). An important observation is that a model's accuracy alone was not a sufficient indicator to predict the system's true performance in the track. For example, the best model (#2) we tested completed 9 laps without a single crash, but another similarly accurate—in terms of validation loss and accuracy—model (#7) was only able to complete 5 labs without crash. Note also that models #6 and #7 achieve good accuracy yet perform worse than significantly less accurate model #4. When we consider latency into account, however, it is clear that these highly accurate models did not work well as their latencies are significantly higher than others. As such, we find that our heuristic score that considers both accuracy and latency into account generally performed well in predicting each model's true relative performance. That being said, our joint optimization strategy is not perfect and can incorrectly predict the performance of some models. For example, the model #5 in the table, which has a relatively good score of 0.58, performed very poorly in the real world and couldn't complete any laps around the track. Unlike other models with low (good) scores, all of which had accuracy of $>80\%$, model #5 had a relatively low accuracy of $\sim 75\%$. This indicates that accuracy is too low, even with a fast inference time, it can lead to undesirable results.

B. Performance in Udacity Simulator

In order to better evaluate and understand the relationship between both model accuracy and latency with respect to control performance, we conduct a systematic simulation study using the Udacity self-driving car simulator [26].

We run the simulator on a desktop computer that is equipped with a Nvidia GTX 1060 GPU and is running Ubuntu 20.04 for its OS. Using the simulator, we evaluated various models that could meet the resource and latency constraints of the DeepPicarMicro (i.e., Raspberry Pi Pico MCU's constraints). For this, we perform the same general workflow to find and train CNN models as we did for the DeepPicarMicro. We first manually collect training data by driving the simulated car around the first default track available in the simulator. Figure 5 shows an overview of the track we use for our simulation environments.

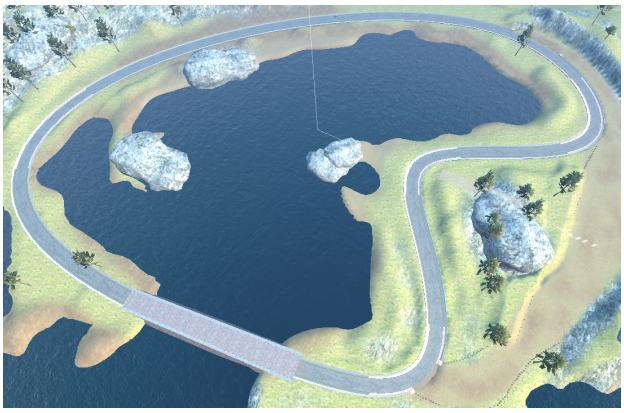

Fig. 5: Udacity simulator's first default track.

In this case, we collect a dataset of 14,468 samples. We then perform the same NAS approach as in Section IV, and obtain models with varying validation losses. In total, we train 84 models with validation losses ranging from 0.0259 to 0.0651.

We next evaluate the impacts that both validation loss and inference latency have on control performance. To accomplish this, we select a subset of six models with varying validation losses. For each model, we then add synthetic delays in the range of [0, 20, ..., 100] ms after each model inference to simulate longer model latencies and, by proxy, control actuations. Note that the actual inference times of the models on the PC were relatively negligible ($\sim$100-200μs). For each validation loss and inference latency combination, we adopt a similar methodology as in real-world experiments. This time, however, we measure how many seconds the car can drive before it crashes, more commonly known as the car's *Time to Crash (TtC)*. Due to the increased size of the simulated track, we measure average TtC to better analyze control performance. For each test case, we measure the car's TtC value across five different runs, with each run having a maximum length of five minutes (300 seconds). Finally, we calculate the average TtC for each test case. This gives us a total of 30 data points.

Table VI shows the results from the simulator tests. As expected, we find that both validation loss and inference latency play a vital role in the simulated car's control performance. For instance, the model with the lowest validation loss (highest accuracy) fails to stay on track if the inference latency is over 80 ms. As validation loss increases, this reduction in control performance (lower TtC) becomes even more apparent.

		Validation Loss				
		0.026	0.030	0.035	0.040	0.045
Latency (ms)	0	300	300	92	81	29
	20	300	300	84	79	30
	40	300	300	81	78	27
	60	300	300	79	76	28
	80	93	59	55	60	29
	100	43	56	39	45	28

TABLE VI: Average TtC (seconds) performance for each test case on the Udacity simulator.

		Validation Loss				
		0.026	0.030	0.035	0.040	0.045
Latency (ms)	0	0.00	0.12	0.28	0.46	0.64
	20	0.20	0.32	0.48	0.66	0.84
	40	0.40	0.52	0.68	0.86	1.04
	60	0.60	0.72	0.88	1.06	1.24
	80	0.80	0.92	1.08	1.26	1.44
	100	1.00	1.12	1.28	1.46	1.64

TABLE VII: Heuristic scores (between 0 to 2) for each test case on the Udacity simulator.

Likewise, models with validation losses $\geq$0.035 also fail to remain on track, and crash before the allotted five minutes.

To validate our joint optimization strategy, we calculate the heuristic scores for each of the simulator test cases. In this case, we normalize the validation losses for all 84 models found in our NAS of the simulator dataset, as well as the synthetic delays that we added in our testing.

Table VII shows the heuristic scores for the 30 test cases. Similar to the real-world experiments, we find that the function does a relatively good job of predicting relative control performance. That being said, the function does not directly correlate to the control performance for the test cases, meaning that there is indeed room for improvement in our strategy. We leave this for future work.

VI. Related Work

There are several RC-car based autonomous car testbeds. MIT's RaceCar [25] and the F1Tenth car [16] are both based on a Traxxas 1/10 scale RC car. Similar to the DeepPicar, the DonkeyCar also employs an end-to-end CNN-based control loop that runs on an embedded Raspberry Pi platform [15]. The development of small-scale autonomous vehicle testbeds has also been seen in industry. For example, Amazon developed its own autonomous 1/18th scale RC car platform called DeepRacer [1]. However, all of these platforms employ microprocessor class computing platforms for their computational needs. In this paper, we instead introduce and evaluate of an MCU-based autonomous vehicle testbed.

In terms of the PilotNet architecture we use in this paper [7], there has been work to improve its performance for autonomous vehicles [6]. This includes a new architecture that utilizes a combination of residual layers, convolutional layers, and fully connected layers. Recent experiments also explored many avenues to improve performance, including data collection, pre-processing, and the use of a real-world representative simulator.In our case, we use the original PilotNet architecture

due to its popularity and simplicity, but plan to evaluate the newer iterations of PilotNet in future work.

With the goal of executing complex DNN-based algorithms on MCUs and other Edge devices, there has been a plethora of work in the TinyML sector [3], [4], [8]–[12], [20], [21], [28]. Due to the relative infancy of the field, though, some works have focused on developing standards that can be used for benchmarking future works. For example, the TinyMLPerf benchmark suite was introduced in order to better enable TinyML-focused research [4]. In addition, many machine learning frameworks have been developed that target MCUs. Apart from TFLM [14], there are other frameworks like CMSIS-NN [18], and uTensor [27]. In academia, the MCUNet framework has found much success in optimizing neural network discovery and inferencing on MCUs [21], achieving SOTA performance on image classification tasks by proposing an intelligent NAS approach and a highly optimized custom ML runtime. They have since extended this work to the MCUNetV2 framework, which instead prioritizes and optimizes peak memory usage for CNN models, thus allowing even bigger models to be deployed on MCUs [20]. In our work, we adopted best-practices in TinyML research and applied them to CNN based end-to-end control of MCU-based autonomous CPS.

VII. Conclusion

We presented DeepPicarMicro, an autonomous RC car platform, which employs a deep-learning based end-to-end control on a tiny MCU. We applied several DNN optimization techniques to execute the well-known PilotNet CNN architecture, which was used to drive NVIDIA's real self-driving car, on the platform's MCU. We also applied a state-of-the-art network architecture search (NAS) approach to find further optimized networks that can effectively control the car in real-time on the MCU. From an extensive systematic experimental and simulation study, we observed an interesting relationship between the accuracy, latency, and control performance of a system. Based on the insights, we proposed a joint optimization strategy that takes both accuracy and latency of a model in the network architecture search process for AI enabled CPS.

For future work, we plan to evaluate more complex state-of-the-art CNN architectures on various MCUs. We also plan to investigate more fine-grained methods for estimating real-world control performance of AI enabled CPS systems and develop effective optimization strategies for MCUs.

Acknowledgements

This research is supported in part by NSF grant CNS1815959, CPS-2038923 and NSA Science of Security initiative contract no. H98230-18-D-0009.

References

[1] Amazon. AWS DeepRacer. https://aws.amazon.com/deepracer/.

[2] ARM. Difference between Memory Arena and Tensor Arena. https://developer.arm.com/documentation/ka004688/latest.

[3] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough. Micronets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. *Proceedings of Machine Learning and Systems*, 2021.

[4] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, et al. Benchmarking TinyML Systems: Challenges and Direction. *arXiv preprint arXiv:2003.04821*, 2020.

[5] M. G. Bechtel, E. McEllhiney, M. Kim, and H. Yun. Deeppicar: A low-cost deep neural network-based autonomous car. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2018.

[6] M. Bojarski, C. Chen, J. Daw, A. Değirmenci, J. Deri, B. Firner, B. Flepp, S. Gogri, J. Hong, L. Jackel, Z. Jia, B. Lee, B. Liu, F. Liu, U. Muller, S. Payne, N. K. N. Prasad, A. Provodin, J. Roach, T. Rvachov, N. Tadimeti, J. van Engelen, H. Wen, E. Yang, and Z. Yang. The nvidia pilotnet experiments, 2020.

[7] M. Bojarski et al. End-to-End Learning for Self-Driving Cars. *arXiV*, 2016.

[8] H. Cai, C. Gan, J. Lin, et al. Network Augmentation for Tiny Deep Learning. In *ICLR*, 2021.

[9] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once for All: Train One Network and Specialize it for Efficient Deployment. In *ICLR*, 2020.

[10] H. Cai, C. Gan, L. Zhu, and S. Han. TinyTL: Reduce Memory, Not Parameters for Efficient On-Device Learning. In *NeurIPS*, 2020.

[11] H. Cai, J. Lin, Y. Lin, Z. Liu, H. Tang, H. Wang, L. Zhu, and S. Han. Enable Deep Learning on Mobile Devices: Methods, Systems, and Applications. *TODAES*, 2022.

[12] H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *ICLR*, 2019.

[13] F. Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[14] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, et al. TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. *Proceedings of Machine Learning and Systems*, 2021.

[15] DonkeyCar. DonkeyCar. http://www.donkeycar.com/.

[16] F1Tenth. F1/10 autonomous racing competition. http://f1tenth.org.

[17] T. A. Foundation. Autoware. https://www.autoware.org/autoware.

[18] L. Lai, N. Suda, and V. Chandra. CMSIS-NN: Efficient Neural Network Kernels for ARM Cortex-M CPUs. *arXiv preprint arXiv:1801.06601*, 2018.

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

[20] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han. MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning. In *NeurIPS*, 2021.

[21] J. Lin, W.-M. Chen, J. Cohn, C. Gan, and S. Han. MCUNet: Tiny Deep Learning on IoT Devices. In *NeurIPS*, 2020.

[22] S. Park, J. Choi, S. Hwang, and C.-G. Lee. ROS2 Extension of Functionally and Temporally Correct Real-Time Simulation of Cyber Systems for Automotive Systems. In *ISEEIE*, 2021.

[23] D. a. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems (NIPS)*, 1989.

[24] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[25] 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.

[26] Udacity. Udacity Self-Driving Car Simulator.

[27] Hand on Embedded Machine Learning. https://utensor.github.io/website/.

[28] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, and S. Han. APQ: Joint Search for Nerwork Architecture, Pruning and Quantization Policy. In *CVPR*, 2020.

[29] K.-S. We, S. Kim, W. Lee, and C.-G. Lee. Functionally and Temporally Correct Simulation of Cyber-Systems for Automotive Systems. In *RTSS*. IEEE, 2017.