2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN) | 978-1-6654-9624-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPSN54338.2022.00049

Demo Abstract: Capuchin: A Neural Network Model Generator for 16-bit Microcontrollers

Le Zhang, Yubo Luo, and Shahriar Nirjon
University of North Carolina at Chapel Hill
lezhang@unc.edu,yubo@cs.unc.edu,nirjon@cs.unc.edu

ABSTRACT

Resource-optimized deep neural networks (DNNs) nowadays run on microcontrollers to perform a wide variety of audio, image and sensor data classification tasks. Despite comprehensive support for deep learning tools for 32-bit microcontrollers, performing deep learning inferences on 16-bit microcontrollers still remains a challenge. Although there are some tools for implementing neural networks on 16-bit systems, generally, there is a large gap in efficiency between the development tools for 16-bit microcontrollers and 32-bit (or higher) systems. There is also a steep learning curve that discourages beginners inexperienced with microcontrollers and programming in C to develop efficient and effective deep learning models for 16-bit microcontrollers. To fill this gap, we have created a neural network model generator that (1) automatically transfers parameters of a pre-trained DNN or CNN model from commonly used frameworks to a 16-bit microcontroller, and (2) automatically implements the model on the microcontroller to perform on-device inference. The optimization of data transfer saves time and minimizes chances of error, and the automatic implementation reduces the complexity to implement DNNs and CNNs on ultra-low-power microcontrollers.

KEYWORDS

Neural Network, Machine Learning, Microcontroller, Edge Computing, Model Generation, On-device Classification.

1 INTRODUCTION

In recent years, we see an increased number of microcontroller-based embedded sensing systems running scaled-down versions of deep and convolutional neural networks to perform on-device inferences for simple audio or image classification tasks [5, 7]. While on-device inference is practical, on-device training is generally unrealistic on these devices due to computational power, memory, and energy limitations. As a result, developers usually train neural network models on a high-end machine and then load the model onto microcontrollers. Although machine learning frameworks like TensorFlow Lite [4] provide solutions for 32-bit microcontrollers, the inconvenience of transferring pre-trained model parameters onto 16-bit microcontrollers troubles the developers. The common solution includes a series of complicated operations such as designing and training the model using a high-level programming language (e.g., python), formatting pre-trained neural network weights and parameters, copying and pasting the weights and parameters in arrays declared inside programs independently written for microcontrollers in a low-level language (e.g., C), and writing code layer-by-layer to configure the inference model (Figure 1). This process is time-consuming and error-prone. While there are a few neural network development frameworks for 16-bit ultra-low-power microcontrollers [5, 7], an efficient tool connecting 16-bit microcontrollers to large-scale machine learning frameworks like TensorFlow is missing.

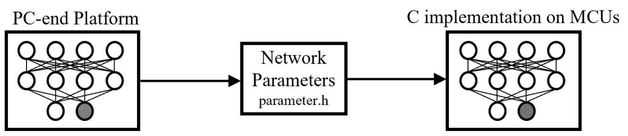

Figure 1: Traditional Workflow. Solid arrows represent manual implementation.

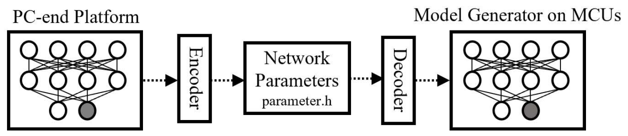

Figure 2: Proposed Workflow. Dotted arrows represent automatic model generation.

In this paper, we introduce Capuchin, which is a deep neural network model generator for 16-bit microcontrollers, such as MSP430 series MCUs, which consists of two parts: (1) a python encoder module collaborated with the TensorFlow framework to generate a header file with pre-trained neural network parameters, and (2) a framework that automatically generates models for the target microcontroller using the header file provided by the encoder module. Capuchin improves the efficiency of developing neural network models on microcontrollers by freeing the programmer from the time consuming, tedious, and error-prone copying-and-pasting job and saves the development overhead due to transferring the weights and biases from TensorFlow models to microcontroller by over 90% – from minutes to a few seconds. At the same time, the automation of header file composing, parsing, and model generation minimizes the chance of error by reducing programmer's engagement. As shown in Figure 2, all underlying data processing mechanisms are transparent to the programmer except the only step to move a file from one system to the other. Programmers are only allowed to use the highly abstract function calls to initiate these mechanisms without the necessity to change anything underlying. Beyond that, the model-level and layer-level abstraction function calls scaffold a framework for programmers to perform creative work on microcontrollers beyond single model inferences, e.g., on-device training and multitask learning. Besides, Capuchin decreases the demands of understanding the microcontroller platform and the skills of programming in C. The code generator allows users to touch zero C code, which benefits beginners unfamiliar with or reluctant to program and debug in C.

2 DESIGN

The design of Capuchin is constrained by the limited resources of the target platform. In particular, we use TI MSP430FR5994 16-bit/16MHz ultra-lower-power (1.8V-3.6V, 0.7μA idle, 118μA/MHz active) microcontrollers which come with 8KB SRAM (volatile memory) and 256KB FRAM (non-volatile memory) [6]. These microcontrollers are quite capable of executing memory-optimized DNNs [5, 9, 10]. These microcontrollers are programmed in C. Hence, to create a bridge between python-based modern deep learning frameworks and these microcontrollers, we use C header files as the file format for data transferring and address two major concerns pertaining to memory usage: (1) the commonly used Python-to-C compilers cause massive redundancy after compiling, and (2) other file formats cannot be directly compiled with C and thus require extra external parser libraries. We implement an encoder module to interface with TensorFlow and to transfer neural network parameters into a header file. The microcontroller parses these parameters from the header file to generate neural network models.

Data Encoding. To extract the data from TensorFlow neural network models, we design our encoder module to interface with Keras APIs, one of the most popular machine learning frameworks. Specifically, by iterating the layers of a Keras [1] model object, the encoder module extracts information, including input shapes, output shapes, filter sizes, pool sizes, stride sizes and activation functions, along with weights and biases of each layer. Then the module processes data into formats corresponded with our microcontroller framework, serializes the data into an array, and writes the array into a header file. Users can import the encoder module into their code and call the encode function with the desired pre-trained model as the parameter.

Data Transfer. In our current implementation, data transfer is done manually between TensorFlow and the microcontroller – which are two separate systems. Programmers are prompted to copy a header file generated by the encoder module to the designated directory in the microcontroller framework. The header file is later compiled with the framework for inference tasks. In the next version of Capuchin, we aim at automating this process.

Model Generation. Capuchin is built on the top of [7] which provides support for fixed-point operations, matrix operations, and dense layers. We develop additional functions, including filter operations for convolutional layers, max-pooling layers, and the pipelining design for model generation. Specifically, we employ MSP430 Low-Energy Accelerator (LEA) [6] to optimize both the inference time and energy consumption by about 10%. Our model generator parses the array in the header file into each layer's parameters, weights, and biases and applies these data to generate an inference model. Given the limited memory and static memory allocation, we implement a pipeline using two static buffer arrays, input buffer feeding input for each layer and output buffer saving the result, and copying output buffer to input buffer for the next layer. Thus, we optimize memory usage by reusing two static arrays for inputs and outputs of all layers.

Examples. We provide examples of models generated with Capuchin and their performance in Table 1. The details of the networks can be found here [14].

Dataset	Size(kB)	Time(s)	Accuracy(%)
mini-GSC [12, 13]	44	4.9	79
ESC-50 [11]	69	5.9	69
CIFAR-10 [8]	78	5.4	61
VWW [3]	2	2.5	77
HAR [2]	16	1.0	94

Table 1: Performance over popular datasets.

3 DEMONSTRATION

We will demonstrate how a real-world sensing and inference problem can be solved using Capuchin. We will also do a live demo to show the inference performance. We will implement a sensor (e.g., microphone, camera, accelerometer, etc.) to collect real-time data and do the classification onsite. The demonstration will happen in two phases. First, the audience will be shown step-by-step how to use Capuchin to create the classifier. We will use a pre-trained model developed in TensorFlow and use Capuchin to transfer it and generate a binary file for MSP430. The audience will be allowed to choose or modify the neural network architecture to see how that change is reflected when Capuchin generates the model for the microcontroller. Second, there will be a live demo where the system samples and classifies sensor data onsite. Besides the classification result, we will also show the CPU and memory usage and the real-time performance of the classifier to help the audience understand the difference between different model choices.

REFERENCES

[1] 2015. Keras. https://keras.io.

[2] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, Jorge Luis Reyes-Ortiz, et al. 2013. A public domain dataset for human activity recognition using smartphones.. In *Esann*, Vol. 3. 3.

[3] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. 2019. Visual wake words dataset. *arXiv preprint arXiv:1906.05721* (2019).

[4] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, et al. 2020. Tensorflow lite micro: Embedded machine learning on tinyml systems. *arXiv preprint arXiv:2010.08678* (2020).

[5] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 199–213.

[6] Texas Instruments Inc. 2021. MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers. https://www.ti.com/lit/ds/symlink/msp430fr5994.pdf

[7] Tejas Kannan and Henry Hoffmann. 2021. Budget RNNs: Multi-Capacity Neural Networks to Improve In-Sensor Inference Under Energy Budgets. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[8] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[9] Seulki Lee, Bashima Islam, Yubo Luo, and Shahriar Nirjon. 2019. Intermittent Learning: On-device Machine Learning on Intermittently Powered System. *Proc. of the ACM IMWUT* 3, 4 (2019).

[10] Seulki Lee and Shahriar Nirjon. 2020. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 175–190.

[11] Karol J Piczak. 2015. ESC: Dataset for environmental sound classification. In *Proceedings of the 23rd ACM international conference on Multimedia*. 1015–1018.

[12] Tensorflow. [n.d.]. Simple audio recognition: Recognizing keywords. https://www.tensorflow.org/datasets/catalog/speech_commands

[13] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209* (2018).

[14] Le Zhang. 2022. Capuchin Github. https://github.com/leleonardzhang/Capuchin