RCE-NN: A Five-Stage Pipeline to Execute Neural Networks (CNNs) on Resource-Constrained IoT Edge Devices

Bharath Sudharsan
Confirm SFI Centre for Smart Manufacturing
Data Science Institute
National University of Ireland Galway
b.sudharsan1@nuigalway.ie

John G. Breslin
Confirm SFI Centre for Smart Manufacturing
Data Science Institute
National University of Ireland Galway
john.breslin@nuigalway.ie

Muhammad Intizar Ali
Confirm SFI Centre for Smart Manufacturing
School of Electronic Engineering
Dubin City University
ali.intizar@dcu.ie

ABSTRACT

Microcontroller Units (MCUs) in edge devices are resource constrained due to their limited memory footprint, fewer computation cores, and low clock speeds. These limitations constrain one from deploying and executing machine learning models on MCUs. To fit, deploy and execute Convolutional Neural Networks (CNNs) for any IoT use-case on small MCUs, a complete design flow is required. *Resource Constrained Edge - Neural Networks* (RCE-NN) is the name given to our proposed design flow, with a five-stage pipeline that developers can follow for executing CNNs on MCUs. In this pipeline, the initial model architecture and training stage consists of four well-defined tasks on model size, workload, operations and quantization awareness, which maps the desired CNN as captured in an executable specification to a resource-constrained MCU's specification. The next quantization and conversion stage reduces model size, saves memory, and simplifies calculations without much impact on the accuracy. In the third stage, the quantized version of the model is translated into a *c-byte* array since the MCUs lack native file-system support. The translated *c-byte* array is fused with the main program of an IoT use-case and binaries are built using techniques from the fourth stage. Finally, the method presented in the last deployment stage is used to flash the built binaries onto MCUs, as this method allows the memory of the MCU to be fully utilized by the CNN and its operations. We evaluated RCE-NN using eight popular MCU boards. The results show that, when users realize all five pipeline stages, they can fit, deploy and execute multiple CNNs across multiple open-source MCU boards. The RCE-NN pipeline components quantize and compress the CNNs to $1/10^{th}$ of their original size, enabling the CNNs to fit on MCUs with no or minimal loss in performance, both after quantization and compression, and also during runtime.

CCS CONCEPTS

• **Computing methodologies → Machine learning**; • **Computer systems organization → Embedded hardware**;

KEYWORDS

Quantization-aware training, Post-training quantization, Model translation, Kernel optimization, RSS prediction, WiFi, BLE, Embedded systems, Edge intelligence, CNN edge deployment.

ACM Reference Format:
Bharath Sudharsan, John G. Breslin, and Muhammad Intizar Ali. 2020. RCE-NN: A Five-Stage Pipeline to Execute Neural Networks (CNNs) on Resource-Constrained IoT Edge Devices. In *Proceedings of 10th International Conference on the Internet of Things, Malmö, Sweden, October 6–9, 2020 (IoT '20),* 8 pages. https://doi.org/10.1145/3410992.3411005

1 INTRODUCTION

In the past few years, CNNs have been used as the principal approach to solving a variety of significant problems in cyber security [27], smart wearables [26], localization [29, 30], resilient communication [17], consumer electronics [19], computer vision [28, 31], etc. Although CNNs are renowned for their excellent performance, they are also computationally intensive, consuming hundreds of MB of memory and MFLOPS of computing power [20, 32].

An edge device is an embedded system with a small microcontroller unit (MCU) as its brain [18]. The Arduino Nano, an 8-bit ATmega328 microcontroller with a 16 MHz clock, 2 kB of SRAM and 32 kB of ISP flash memory, and the NUCLEO-F303K8, a 32-bit ARM Cortex-M4 microcontroller with a 72 MHz clock and 64 kB of flash memory, are two examples of small MCU boards. Many such MCU boards do not provide hardware support for floating-point operations, and billions of such tiny boards have been deployed in the world. Before deploying such an MCU-based edge device for any IoT use-case, the IoT application code data is burnt onto an MCU's flash memory, leaving only a few kB for storing the feature extractor code, trained CNN, and associated sensor data [22, 25]. After burning the code and deploying the MCU for a use-case, the only writable memory available is limited to a few kBs which is not sufficient to hold even a single feature vector.

There are a few other facts which constrain executing CNNs on MCU-based edge devices [24, 25]. Firstly, the memory footprint (SRAM, FLASH and EEPROM) is limited to a few MB. Next, the computation core (commonly a single ARM Cortex-M CPU) runs only up to a few hundred MHz resulting in low operations per second. Next, is the absence of native filesystem support. Finally, it has an inability to perform parallel processing due to the absence of multiple execution units. However, given the potential of building intelligent IoT applications for edge analytics using CNN, there is a strong need for a mechanism which allows IoT applications

IoT '20, October 6–9, 2020, Malmö, Sweden

ACM ISBN 978-1-4503-8758-3/20/10...$15.00
https://doi.org/10.1145/3410992.3411005

to fit, deploy and execute CNNs on edge devices. We therefore present *Resource Constrained Edge - Neural Networks* (RCE-NN), our proposed design flow with a five-stage pipeline, that developers can follow for executing CNNs on resource-constrained edge devices.

In this work, we address the challenges from both the algorithm and the device perspectives. At the device level, the CNN execution capabilities of these MCUs are improved by optimizing the low-level computation kernels to achieve performance improvements and a reduced memory footprint when executing CNN workloads, enabling MCUs to handle larger and complex CNNs. Also, the algorithm level design and optimization of CNNs are presented in this work concerning the targeting MCU platform by network architecture exploration. Both these hardware and software co-optimization components for executing multiple CNNs on multiple small MCUs are designed as a five-stage pipeline. Our main contributions in this paper are: (i) We provide a complete design flow which is the RCE-NN's five-stage pipeline, that developers can follow for executing CNNs for any IoT use-case on small MCUs. Our pipeline also provides the necessary components for building an end-to-end CNN optimizing compiler; (ii) We provide third-party library independent kernel optimization tasks that are applicable across a wide range of MCUs for guaranteeing no runtime performance bottlenecks; (iii) Out of the dozens of post-training quantization techniques studied, we have selected the best-suited choices for MCUs and small CPUs and summarised their benefits. This provides readers with a guideline for selecting the best use-case and edge device-aware quantization technique; (iv) Lastly, we provide a practical method to fuse the optimized version of the CNN along with the main program of the IoT use-case, followed by the best deployment method for maximum utilization of the full program space (flash) of the MCU only for the CNN and its operations.

2 RELATED WORK

The MCU hardware platform has recently become an attractive target to run models due to TensorFlow Micro and MCU-targetted optimized kernels from CMSIS [9]. In this domain of enabling learning for resource-constrained devices, the authors in [8] have implemented a tree-based algorithm, called Bonsai, for efficient prediction on IoT devices. High accuracy predictions were obtained in milliseconds even on slow microcontrollers and were able to fit in a kB of memory. Similarly, ProtoNN, a k-Nearest Neighbor (KNN) inspired algorithm with several orders lower storage and prediction complexity was proposed in [2] to address the problem of real-time and accurate prediction on resource-scarce devices. Both [8] and [2] are tailored prediction algorithms that can fit in resource-scarce devices and show superior performance. However the design flow used by them cannot be applied for executing other CNN based applications on resource-constrained devices.

Another set of articles proposes compression techniques to reduce the size of the model's weights using quantization and weight pruning techniques, resulting in a reduction of both model size and inference time without considerable accuracy loss. CONDENSA [7] is a system for users to programmatically compose simple operators to build complex model compression strategies. Given a strategy and a user-provided objective, CONDENSA automatically infers desirable sparsity ratios. Two new compression methods that jointly leverage weight quantization and the distillation of larger networks was proposed in [13]. Both methods were validated on convolutional and recurrent architectures, and have demonstrated that their method reaches similar accuracy levels to full-precision models while providing up to an order of magnitude compression. In [7, 13] and other similar articles proposing compressing [3, 4] and optimization [1, 5, 9, 10, 21, 23] techniques do not include an MCU-aware CNN tuning step, a C byte array conversion after model quantization and conversion, a way in which the model can be fused with the main program of the IoT application, or a method for how the binaries for the final model and IoT app are generated and deployed on MCUs.

3 RCE-NN: FIVE-STAGE PIPELINE

We propose a five-stage pipeline which helps to execute neural networks on resource-constrained IoT edge devices. Our proposed RCE-NN's design flow contains a complete stepwise pipeline with hardware and software co-optimization at different stages of the pipeline. As shown in Figure 1, there are multiple components at each stage to implement the application design flow and execution on small MCUs. Before training any model for an IoT edge device, the techniques we provided in *Stage One* should be used for an MCU-aware CNN architecture design. Next, after performing direct training or quantization-aware training using the technique we provided, the model should go through any one of the post-training quantization schemes from *Stage Two* to obtain a quantized version of the actual model. Then the quantized model is translated into a C byte array using the command we provided in *Stage Three*. In *Stage Four*, we add the translated model to the root of the project directory along with the necessary files that should be fused within the IoT application. In the same stage, binaries are generated from the project's source file system and loaded into the MCU's flash memory via the serial port using the technique from *Stage Five*.

3.1 Stage One: Architecture and Training

We provide model architecture and training as stage one. In the context of the RCE-NN pipeline, the components from this stage perform the task of mapping the desired functionality, as captured in an executable specification, to resource-constrained system components. For example, the design constraints related to hardware size, power, performance, and monetary cost are satisfied. Our approach for model architecture and training consists of four well-defined tasks on model size, workload, operations and quantization awareness.

3.1.1 Model Size. Models to run at the edge must fit within the target device's memory alongside the IoT application both at the runtime and as a binary. One possible method for fitting is to create a smaller model using fewer and smaller layers. In most use-cases, small models lead to under-fitting. Therefore, for the majority of real-life problems, a larger model suits well where the number of input features and outputs are already decided based on the use-cases. Hence, we explored other areas to look into where optimizations to the model size can be applied. We provide a list of tasks as a summary of our findings to reduce model size. Below we provide the list of tasks;

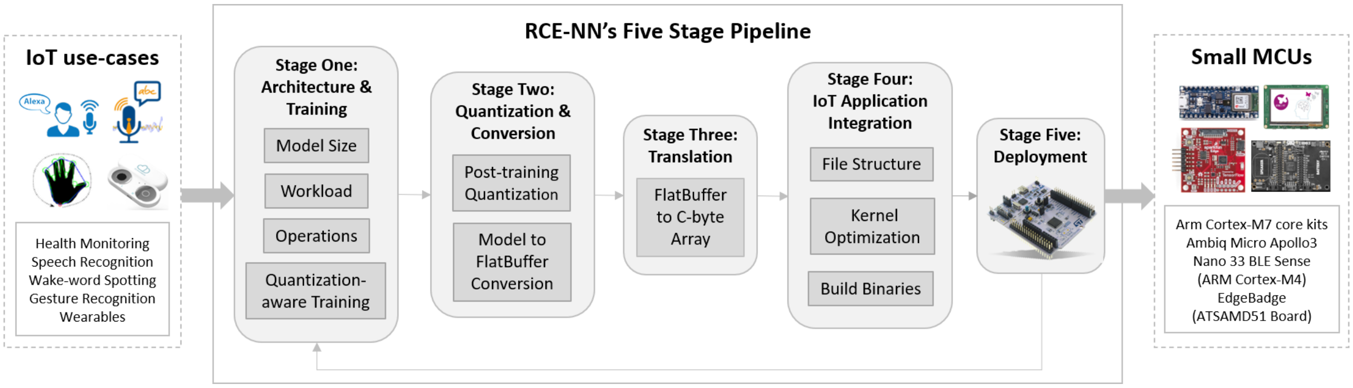

Figure 1: Architecture and Components of RCE-NN's Five-stage pipeline: Design flow to fit deploy and execute CNNs on small MCU's at the IoT edge.

- Task1: Limit the number of neurons in the hidden layers since the memory requirements, model size and computational complexity grows exponentially with it.
- Task2: Multiple variations of the model for the same use-case should be evaluated experimentally for acceptable latency and accuracy, matching resource capabilities (especially memory) of the MCU in the edge device.
- Task3: Pruning edges with negligibly small weights should be performed to achieve a lower count on model parameters.
- Task4: A combination of max pooling or global average pooling layers can be used as an alternative to fully connected layers.
- Task5: Activation function simplification; Since most MCU's do not have built-in floating-point capabilities, realizing functions such as tanh and sigmoid is expensive. The accuracy of these functions could be sacrificed to a reasonable level, reducing the inference time and model size.
- Perform an advanced level of optimizations using hardware-oriented model approximation methods [3]. Here, a given CNN should be analyzed in the context of numerical resolution used in representing weights and outputs of convolutional and fully connected layers.

3.1.2 Workload. The complexity and size of the model impacts the workload (larger models will lead to increased processor workload). Denser models result in a higher duty cycle, hence the MCU's processor spends more time working and less time idle resulting in elevated power consumption and heat output. In this section, we analyzed the linear algebraic properties of CNNs and proposed recommendations (without influencing the inference results) to reduce the computational workloads of CNNs, so it can comfortably run within the resources of edge devices. Our proposed methods apply globally, i.e, not biased towards local performance optimizations for a single operation as in many previous works. Below, we provide the list of methods used to reduce workloads:

- Input data reduction; when the sampling rates of sensors are high, an inference cannot be made by the MCU from the acquired data within a sampling interval. In such scenarios, computationally inexpensive low pass filters should be applied to reduce the volume of data, which also improves the quality of the data (reduced noise), and allows a sufficient time interval to perform inferencing and also reduce workload.
- Boost the CNN algorithm using hardware accelerators, e.g., co-processing units and field-programmable gate arrays (FPGAs), while also reducing the theoretical number of basic operations needed for CNN computation.
- Limit the number of threads initialized by CNNs for computation.
- Perform low-level optimization of convolution operations using [10]. This method adds flexibility in searching for the best implementation of a specific convolution workload on a particular architecture and allows us to optimize the whole computation graph by choosing proper data layouts between operations to eliminate unnecessary data layout transformation overheads.
- Analyze the linear algebraic properties of the CNN designed for a particular use-case, and apply algorithms such as Strassen Gaussian elimination, Winograd's minimal filtering [16], etc. to reduce the computational workload, resulting in increased available memory.

3.1.3 Operations. When designing a CNN to run on edge devices, only a limited subset of operations can be used for keeping the operational costs low, which impacts the designed CNN's architecture. Hence, to execute CNNs on MCUs by reducing operations, without impacting the architecture, we recommend using the weight factorization approach, which significantly reduces the memory footprint of models by optimizing the parameter space of the fully connected layers and also reduces the overall number of operations needed while improving the inference time. We also recommend another method [1], which is to find a sparse representation of the fully connected layers and use separate filters to separate the convolutional kernels, reducing the number of parameters and convolutional operations required to execute a CNN.

3.1.4 Quantization-Aware Training. Here we select and give a brief overview of a method that can be practically realized on CNNs trained for small MCUs. Also, it falls in line (matches up) with other pipeline stages and their components while providing higher accuracy than post quantization training schemes. We first consider the quantized weights in full precision representation to simulate and inject quantization error [6] into training, thus enabling the weights to be optimized against quantization errors. This quantization error is modeled using fake quantization nodes, which simulates the effect of quantization in both forward and backward passes. These fake quantization operations are added to all required locations in the model by rewriting the training graph (by creating a fake quantized training graph).

For the majority of use cases, this method significantly improves the latency-versus-accuracy tradeoffs. In cases when performance is not improving, then the user should directly train model and perform post-training quantization using any of the methods we provide in Section 3.2.1 since it is broadly applicable to all models and does not require training data during quantization.

3.2 Stage Two: Quantization and Conversion

After performing direct training or quantization-aware training, the model should go through any of the post-training quantization schemes we provide, followed by its conversion into a FlatBuffer.

3.2.1 Post-training Quantization. The quantize and train scheme from Section 3.1.4 required model modifications, in terms of adding fake quantization nodes. In contrast, in this section we quantize the existing pre-trained model by reducing the precision of the model's weights to save memory and simplify calculations often without much impact on accuracy. From the multiple available techniques to choose from, a summary of schemes which suits MCUs and small CPU devices are provided along with their benefits.

Weight-only quantization: Out of the various weight-only quantization schemes studied, a simple approach we provide is to just reduce the weight's precision in the network from a float to 8-bit precision by using any command-line tool, without requiring validation data (which reduces the model size by at least four times). RCE-NN realizes the most efficient approach to quantize the CNN's weight w to a Q-bit fixed-point number $quant(w)$, by using the quantization function (Eqn. 1).

$$quant(w) = clip_{[-1,1)}(2^{-(Q-1)}.round(w.2^{(Q-1)})) \qquad (1)$$

Here $clip_{[-1,1)}(x) = max(a, min(x, b))$, and the corresponding INT-Q representation of a CNN's weights is $W = quant(w).2^{(Q-1)}$. The same Eqn. 1 is applied to any activation values as well. In this work, RCE-NN converts the weights and activations to an Int-8 data type since they are the most natural type to fit in 32-bit MCU registers.

Weights and activations quantization: Similar to weights, activations alone can also be quantized to 8-bits with almost no accuracy loss. We studied multiple techniques and found out that Symmetric per-channel, Asymmetric per-layer, and Asymmetric per-channel are well-suited techniques to quantize both weights and activations. We recommend users to select Per-channel quantization with asymmetric ranges over other techniques since it provides close-to-floating point accuracy for a wide range of networks. RCE-NN quantizes both weights and activations to Int-8 values. Hence the convolution in CNNs takes the following form:

$$\psi(w, x) = 2^{-2(Q-1)} \sum_{i \epsilon D} W_i X_i \doteq 2^{-2(Q-1)}.\phi(W, X) \qquad (2)$$

In Eqn. 2, D is the number of input channels, ψ is CNN's convolution operation, and $\phi(W, X)$ is an accumulator containing high precision values, in our case, Int-32 for Int-8 operands. We used this method to convert the CNN's weights and activations to 8-bit integers. For CNNs, the inference time improves after quantization since the inference is carried out using integer-only arithmetic. We selected this Int-8 quantization method since it outperforms the Fine-Grained Quantization (FGQ) method (2 bits for weight quantization) and Incremental Network Quantization (INQ) method (5-bit weight floating-point activation) by preserving accuracy, while also providing run-time improvements.

Float16 quantization: Here, we quantize the original CNN's Float32 weights and activations to Float16 values. Users can use this Float16 quantization when they want to achieve reasonable compression rates without loss of precision. Also, Float16 models run on small CPUs without modification.

3.2.2 Model to FlatBuffer Conversion. This is the second component of the second stage which the user has to follow for converting the post-training quantized models into FlatBuffer, using FlatBuffer's cross-platform serialization library[1] or by also using the TensorFlow Lite converter[2]. After conversion, the resulting FlatBuffer format file contains direct data structures of the trained CNN. This data structure contains information arrays with a graph consisting of subgraphs, where each subgraph consists of a list of tensors and operators. After this stage, since the flat buffers of the CNNs for IoT use-cases are memory-mapped, they can be utilised directly from disk/flash without any loading or parsing tasks, and with zero additional memory requirements for accessing the data (the only memory required to access data is that of the buffer).

3.3 Stage Three: Translation

Most MCUs in edge devices do not have native filesystem support, hence we convert the quantized version of the trained model into a C array, and compile it along with the program for the IoT application which is to be executed on the edge device. In our pipeline, to perform this conversion we use a UNIX command as shown in Figure 2.a, which generates the C source file containing the quantized model as a char array.

3.4 Stage Four: IoT Application Integration

In stage four, we first provide a method to fuse trained CNNs with the main program for an IoT use-case. Then, to execute this CNN-fused-IoT program on MCUs without runtime performance bottlenecks, we provide third-party library independent kernel optimization tasks that are applicable across a wide range of MCUs. Finally, we will briefly describe a technique to build binaries of the CNN-fused-IoT program which will be used in the subsequent deployment stage (Stage Five).

3.4.1 File Structure. Here, we explain how to use the C byte array (obtained using the technique from Section 3.3) of the trained model, when the main program of an IoT use-case requires predictions/inference results. Initially, users have to fit the blocks we provide in Figure 2.c along with the flow of the main program, in

[1] https://google.github.io/flatbuffers/flatbuffers_white_paper.html

[2] https://www.tensorflow.org/lite/r1/convert

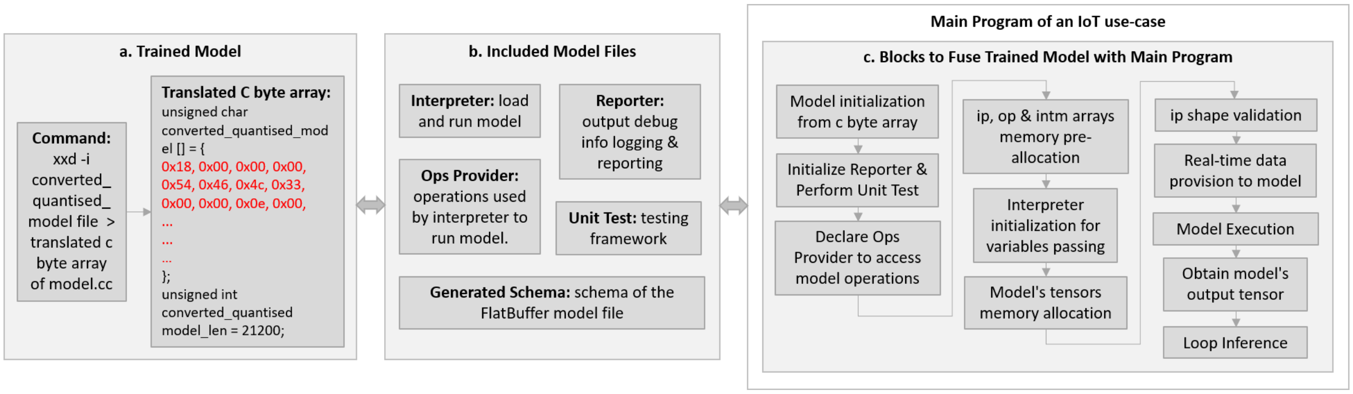
a. Trained Model
Command:
xxd -i converted_quantised_model file > translated c byte array of model.cc
Translated C byte array:
unsigned char converted_quantised_model [] = {
0x18, 0x00, 0x00, 0x00,
0x54, 0x46, 0x4c, 0x33,
0x00, 0x00, 0x0e, 0x00,
...
...
...
};
unsigned int converted_quantised model_len = 21200;
b. Included Model Files
Interpreter: load and run model
Reporter: output debug info logging & reporting
Ops Provider: operations used by interpreter to run model.
Unit Test: testing framework
Generated Schema: schema of the FlatBuffer model file
Main Program of an IoT use-case
c. Blocks to Fuse Trained Model with Main Program
Model initialization from c byte array
Initialize Reporter & Perform Unit Test
Declare Ops Provider to access model operations
ip, op & intm arrays memory pre-allocation
Interpreter initialization for variables passing
Model's tensors memory allocation
ip shape validation
Real-time data provision to model
Model Execution
Obtain model's output tensor
Loop Inference

Figure 2: a. Translating the trained model into a C byte array. b. Files required to load and execute the model's C byte array on MCU boards. c. Blocks to fuse the C byte array of the trained model with the main program in an IoT use-case.

the order shown in Figure 2.c. When the MCU executes our blocks during the execution of the main program, these blocks will use the Included Model Files (shown in Figure 2.b) to perform their intended tasks. We import these Included Model Files from the TensorFlow Micro library. These files need to be included in the root of the project along with the model's C byte array. Below we provide and describe the blocks to fuse the C byte array of a trained CNN with the main program for an IoT use-case (i.e. the IoT application running on MCU-based edge devices):

i Model initialization from C byte array: To initiate the compressed and translated model from the C byte array shown in Figure 2.a.
ii Initialize reporter and perform unit test: To invoke the "Unit Test and Reporter" file to perform a unit test for ensuring that the entire flow works correctly.
iii Declare Ops Provider to access model operations: To invoke the "Ops Provider" file to initialize the operators used by the model.
iv Ip, op and intm arrays memory pre-allocation: This block pre-allocates memory for these arrays.
v Interpreter initialization for passing variables: To invoke the interpreter file for initializing the interrupter to load the model and pass variables.
vi Model's tensors memory allocation: This block allocates the memory for the model's tensors.
vii Ip shape validation: A block to confirm whether the input's shape and type are as expected.
viii Real-time data provision to model, model execution, obtain model's output tensor and loop inference blocks: After executing the previous blocks, the real-time data that the sensors are seeing is received by the MCU's peripherals and fed to the model for inference. Thus, obtained inference results are the model's output tensors, which are fed to the main program of the IoT application. The process as outlined is looped to obtain continuous inferences for its corresponding real-time sensor data.

3.4.2 Kernel Optimization. The general pure C/C++ implemented reference kernels for MCUs need MCU platform-specific hardware optimizations for achieving reduced data movement overhead, reduced inference time, faster convolution, loop parallelism, temporal parallelism, improved stability, etc. At the time of writing, according to the authors, the versions of the best-optimized kernels for MCUs are provided for implementation in CMSISNN [9]. Other libraries such as NNPack provide manually optimized CNN operators on ARM CPUs; likewise Android NN is available for efficient inference on Android. To the best of our knowledge, such off-the-shelf libraries cannot optimize kernels of models trained using different deep learning frameworks and targeted to be deployed on a wide range of MCUs. In this section, we provide library independent kernel optimization tasks that are generic across a wide range of MCUs for guaranteeing no runtime performance bottlenecks.

- Task 1: Remove excess modules and components inside the project directory before building, using the technique we provided in 3.4.3. This reduces the size of the compiled kernel, and also aids the MCU to boot faster.
- Task 2: Group multiple operators together within a single kernel. Performing this task will improve efficiency due to better memory locality.
- Task 3: Matrix multiplication is a computationally intensive task, yet the most important computation kernel that needs to be used during convolution operations. In the context of matrix multiplication on MCUs, out of the various schemes studied, we provide the best optimization methods. LIBXSMM [5], which goes deep into the assembly code level for improving small matrix multiplication tasks, should be used for improving kernel performance. Also, the matrix multiplication kernel should be implemented with 2x2 kernels to save on the total number of load instructions while also enabling some data re-usage.
- Task 4: The computationally intense convolutions traverse their operands many times during computation. Hence, managing the layout of the data fed to CNNs is critical for reducing memory access overheads. We also recommend applying algorithms to optimize convolution in a single thread (thread optimization) to reduce memory access overheads.
- Task 5: The convolution should be partitioned into disjoint pieces to achieve parallelism. At the CPU level, off-the-shelf

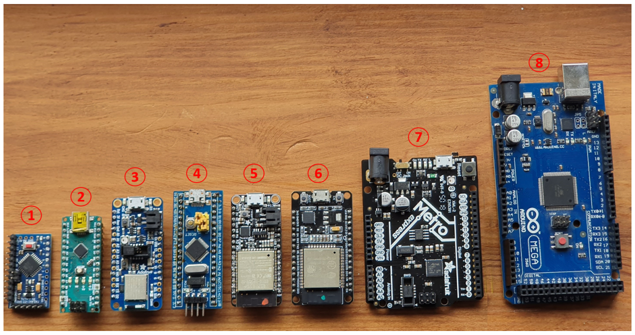

Figure 3: MCU-based development boards chosen to evaluate RCE-NN.

multithreading solutions such as OpenMP (used by the Intel MKL-DNN kernel library) are used to achieve parallelism via shared memory multiprocessing. But for MCUs, such approaches are not suited. For MCUs, self-customized thread pooling techniques should be used to reduce overheads while launching and suppressing threads, to reduce performance jitters while adding threads. Using such a self-customized thread pool provides full control of the IoT application while maintaining performance across different MCU platforms.

3.4.3 Build Binaries. In this Section, we give an overview of a technique to build binaries and use them during deployment (Stage Five). Binaries are the machine runnable code that needs to be generated from the project's source file system, which we load into the MCU's flash memory via the serial port. In our pipeline, to build the MCU executable binary, the users need to use the GNU make -f [3] command. By default, the make command compiles for the host system. Hence the users need to select a target architecture depending on the target MCU, using the TARGET parameter of the GNU make command. In the end, the user should ensure that the binaries do not exceed the MCU's flash memory size.

3.5 Stage Five: Deployment

In this stage, the binaries are generated from the project's source file system using the technique we provided in Section 3.4.3, and should be ready to be loaded into the MCU's flash memory via the serial port. In general, these binaries are loaded into the MCU with the help of a bootloader, that allows uploading of the compiled binary for the IoT use case without the need for an external programmer, but at the cost of consuming memory to occupy the bootloader. To avoid this and achieve maximum utilization of the full program space (flash) of the chip only for the CNN and its operations, we provide a methodology to burn/flash the generated binaries into the MCU. In our method, it is mandatory not to flash the bootloader on the MCU. Therefore, to perform the binaries upload/flash task, our method uses an external In-System Programmer (ISP) or a ParallelProgrammer, which does not occupy bootloader space and also avoids the bootloader delay.

Table 1: Specification of boards chosen to evaluate RCE-NN.

Board	MCU & Board Name	Specification					
		Bits	EEP ROM	SRAM	Flash	Clock (MHz)	FP
#1 #2	ATmega328P Arduino Pro Mini, Nano	8	1kB	2kB	32kB	16	✗
#3	nRF52840 Adafruit Feather	32	-	256kB	1MB	64	✓
#4	STM32f103c8 Blue Pill	32	-	20kB	128kB	72	✗
#5 #6	Adafruit HUZZAH32, Generic ESP32	32	-	520kB	4MB	240	✓
#7	ATSAMD21G18 Adafruit METRO	32	-	32kB	256kB	48	✗
#8	ATmega2560 Arduino Mega	8	4kB	8kB	256kB	16	✗

4 EVALUATION

We have explained the five-stages of RCE-NN with its hardware and software co-optimization components that enable us to fit, deploy and execute multiple CNNs from the laboratory environment on multiple MCUs. In this section, we focus on the end-to-end evaluation of the RCE-NN pipeline, that answers: (i) Can RCE-NN be used to execute CNNs on various MCU-based development boards? (ii) How does RCE-NN influence the accuracy and size of models?

4.1 Datasets and Evaluation procedure

We evaluate RCE-NN using eight MCU-based boards, as shown in Figure 3, with their specifications given in Table 1. We categorize these boards into two groups, based on their resource constraints. The first group contains boards 1, 2, 7 and 8. From all of the boards chosen, the boards with the better resources form the second group (boards 3, 4, 5 and 6), where the boards 3, 5 and 6 from the second group support floating-point operations. Having these two groups of boards, we chose one model per group, aiming to fit and execute on all boards of that group. For the first group, we designed a CNN to perform RSS (Received Signal Strength) prediction of BLE signals. This CNN was designed using the instructions provided in RCE-NN's Stage One (Section 3.1) and trained using the BLE RSS [11] dataset. Then we quantized and converted CNNs using techniques from Stage Two (Section 3.2). Next, we follow the instructions provided from Stage Three (Section 3.3) to Stage Five (Section 3.5) to translate the model into C code, followed by integrating it into an IoT application and deploying it on all the boards from the first group. The second group of boards have better FLASH and SRAM resources, so we designed a CNN with more hidden layers to perform the RSS prediction of WiFi signals. This CNN was trained using the kthrss [12] dataset, followed by realizing the instructions provided in the subsequent pipeline stages (Stages Two to Five).

4.2 Executing CNNs on MCUs

After realizing pipeline Stage Five, the BLE and WiFi RSS prediction CNNs are loaded on all the boards shown in Figure 3. The boards from the first group were powered on individually, then multiple unseen series of BLE RSS data from the finalized datasets were fed to the input layer of the loaded CNNs via their serial port. For the second group of boards, unseen series of WiFi RSS data from the

[3] https://www.gnu.org/software/make/manual/make.html

Table 2: Executing CNNs on MCU boards using RCE-NN. CNN size versus performance after pipeline Stage Two. Performance comparisons of actual BLE and WiFi RSS prediction CNNs with their Quantized versions, Quantized and Converted versions, and Real-time performance when executing on MCU boards.

CNN	Size (kB)			Evaluation Datasets	Performance MAE (dBm)			Performance on MCU Boards MAE (dBm)			
	Actual	Quant	Quant & Conv		Actual MAE	Quant MAE	Quant & Conv MAE	#1	#2	#7	#8
BLE RSS Prediction	63.3	15.2	6.0	BLE RSS [11]	4.227014	4.227019	4.151393	4.128201	4.128206	4.158090	4.162391
				BLEBeacon [15]	4.464411	4.464414	4.527941	4.531073	4.531071	4.543889	4.539941
								#3	#4	#5	#6
WiFi RSS Prediction	148.6	38.0	13.6	kthrss [12]	3.210762	3.210761	3.209073	3.221350	3.232757	3.206426	3.206420
				phonelab-wifi [14]	3.022681	3.022679	3.032625	3.039942	3.035064	3.035308	3.035316

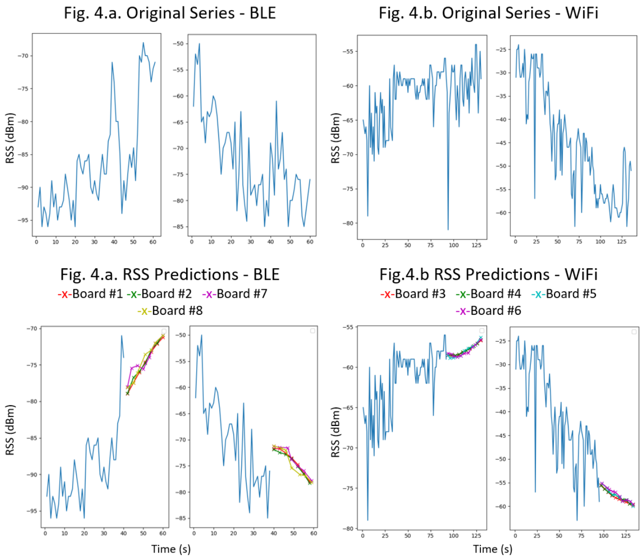

Figure 4: Executing CNNs on boards after realizing all five pipeline stages: Plotting BLE and WiFi RSS predictions obtained from MCUs.

finalized datasets was fed to them. The CNNs running on their respective boards predicted the future RSS values and sent them via the serial port. We received these predictions from the MCU boards and used them to calculate the performance (MAE) of the CNN during its execution on their respective MCUs. This real-time performance across all the boards is provided in Table 2. For visualization, in Figure 4, we took two original unseen data series from the BLE and WiFi datasets, plotted them, and compared them with the RSS predictions produced by all the MCU boards. Here, the predicted RSS is sketched in the colors mentioned in the graph's legend. The trend/slope of the predicted lines (for all boards) is very close to the trend of the raw/original series shown.

It is apparent from Figure 4 that after realizing all five pipeline stages, the trained models can be executed on all the popular MCU boards shown in Figure 3. We estimate that our RCN-NN pipeline can be used to fit, deploy and execute multiple CNNs across thousands of open-source MCU boards supported by Arduino IDE.

4.3 CNN Size

Before realizing the components from pipeline Stage 2 (Section. 3.2), the actual BLE RSS prediction CNN size was 63.3 kB, which exceeds the available memory resource of boards 1, 2, 7 and 8 from the first group. Similarly, the WiFi RSS prediction CNN size was 148.6 kB, which cannot be fit onto board 4 from the second group. Therefore, to reduce the size without increasing the Mean Absolute Error (MAE), we statically quantize the weights and activations from floating-point to 8-bits of precision. We chose this technique since 8-bit calculations help RSS prediction CNNs run faster on MCUs while using less power. It also enables executing of CNNs on boards 1, 2, 4, 7 and 8 from Table 1 that cannot run floating-point code, while saving approximately 70% of the memory bandwidth.

It is apparent from Table 2 that the components of Stage Two quantize and compress the CNNs from 63.3 kB to 6.0 kB and 148.6 kB to 13.6 kB, which enables the CNNs to fit on such small devices shown in Figure 3 with almost no loss on their performance (MAE) both after Quantization and Compression and also during run-time.

4.4 CNN Performance

To measure the performance of the RSS prediction models, Mean Absolute Error (MAE) was used as an evaluation metric. We used MAE over other metrics (accuracy, mean squared error, F1 score, etc.) since for RSS signal data, MAE is more robust as it is less sensitive to fluctuations (outliers). MAE was calculated using Eqn. 3, where z_{k+p} is the true future RSS values, and $\hat{z}^{\mu}_{k+p}$ is the mean of the predicted RSS value. The model running on the boards, used to predict the RSS of WiFi and BLE, was tested on unused columns of data from the finalized datasets and was also validated using new datasets (BLEBeacon [15] for BLE and phonelab-wifi [14] for WiFi). During execution on MCUs, the CNNs were able to achieve an MAE of less than 3.23 dBm for WiFi and 4.54 dBm for BLE across all boards as shown in Table 2.

$$MAE = AE^{\mu} = |\hat{z}^{\mu}_{k+p} - z_{k+p}| \tag{3}$$

From Table 2, it can also be noticed that the MAE obtained by executing CNNs across all boards is the same until its first decimal point: 4.1 dBm across all boards for BLE RSS dataset, 4.5 dBm for BLEBeacon, 3.2 dBm for kth_rss, and 3.0 dBm for phonelab-wifi. From this, it is clear that when RCE-NN's pipeline is used to fit, deploy and execute CNNs across various MCU boards, CNNs perform the same, irrespective of the target MCUs.

5 CONCLUSION

We presented RCE-NN, a design flow that contains a five-stage pipeline with hardware and software co-optimization components that developers can follow to fit, deploy and execute multiple CNNs from the laboratory environment on multiple resource-constrained embedded systems like FPGAs, MCUs, etc. Using our pipeline it will be possible to fit CNNs into resource-constrained devices and execute a variety of IoT edge-based applications and use cases (health monitoring, speech recognition, wake word spotting, gesture recognition, etc.). We evaluated RCE-NN by using its pipeline to execute CNNs that predict the RSS of WiFi and BLE on eight popular MCU boards. The pipeline components quantize and compress the CNNs to $1/10^{th}$ of their original size while also preserving accuracy until its first decimal point across all boards.

ACKNOWLEDGEMENTS

This publication has emanated from research supported by research grants from Science Foundation Ireland (SFI) under Grant Number SFI/16/RC/3918 (Confirm) and SFI/12/RC/2289_P2 (Insight), co-funded by the European Regional Development Fund.

REFERENCES

[1] Sourav Bhattacharya and Nicholas D. Lane. 2016. Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables. In *Conference on Embedded Network Sensor Systems (SenSys)*.

[2] Chirag Gupta, Prateek Jain, et al. 2017. ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*.

[3] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented Approximation of Convolutional Neural Networks. *arXiv preprint*.

[4] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint*.

[5] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. In *Proceedings of the International Conference for High Performance Computing*.

[6] Benoit Jacob, Dmitry Kalenichenko, et al. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Conference on Computer Vision and Pattern Recognition*.

[7] Vinu Joseph, Saurav Muralidharan, Animesh Garg, Michael Garland, and Ganesh Gopalakrishnan. 2019. A Programmable Approach to Model Compression. *arXiv*.

[8] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In *ICML*.

[9] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint*.

[10] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *USENIX Annual Technical Conference*.

[11] Germán Martín Mendoza-Silva, Miguel Matey-Sanz, Joaquín Torres-Sospedra, and Joaquín Huerta. 2018. *BLE RSS measurements database and supporting materials*. https://doi.org/10.5281/zenodo.1618692

[12] Ramviyas Parasuraman, Sergio Caccamo, Fredrik Baberg, and Petter Ogren. 2016. CRAWDAD dataset kth/rss (v. 2016-01-05). https://doi.org/10.15783/C7088F

[13] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. *arXiv preprint arXiv: 1802.05668*.

[14] Jinghao Shi, Chunming Qiao, Dimitrios Koutsonikolas, and Geoffrey Challen. 2016. CRAWDAD dataset buffalo/phonelab-wifi (v. 2016-03-09). https://doi.org/10.15783/C7QS37

[15] Dimitrios Sikeridis, Ioannis Papapanagiotou, and Michael Devetsikiotis. 2018. BLEBeacon: A real-subject trial dataset from mobile Bluetooth low energy beacons. *arXiv preprint*.

[16] V Strassen. 1969. Gaussian elimination is not optimal. *numerical mathematics*.

[17] Bharath Sudharsan, John G. Breslin, and Muhammad Intizar Ali. 2020. Adaptive Strategy to Improve the Quality of Communication for IoT Edge Devices. In *IEEE 6th World Forum on Internet of Things (WF-IoT)*.

[18] Bharath Sudharsan, John G Breslin, and Muhammad Intizar Ali. 2020. Edge2train: A framework to train machine learning models (svms) on resource-constrained iot edge devices. In *10th International Conference on the Internet of Things*.

[19] Bharath Sudharsan, Sweta Malik, Peter Corcoran, Pankesh Patel, John G Breslin, and Muhammad Intizar Ali. 2021. Owsnet: Towards real-time offensive words spotting network for consumer iot devices. In *IEEE 7th World Forum on Internet of Things (WF-IoT)*.

[20] Bharath Sudharsan and Pankesh Patel. 2021. Machine learning meets internet of things: From theory to practice. *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*.

[21] Bharath Sudharsan, Pankesh Patel, John G Breslin, and Muhammad Intizar Ali. 2021. Enabling Machine Learning on the Edge using SRAM Conserving Efficient Neural Networks Execution Approach. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*.

[22] Bharath Sudharsan, Pankesh Patel, John G Breslin, and Muhammad Intizar Ali. 2021. SRAM optimized porting and execution of machine learning classifiers on MCU-based IoT devices: demo abstract. In *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems (ICCPS)*.

[23] Bharath Sudharsan, Pankesh Patel, John G. Breslin, and Muhammad Intizar Ali. 2021. Ultra-fast Machine Learning Classifier Execution on IoT Devices without SRAM Consumption. In *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*.

[24] Bharath Sudharsan, Pankesh Patel, Abdul Wahid, Muhammad Yahya, John G Breslin, and Muhammad Intizar Ali. 2021. Demo Abstract: Porting and Execution of Anomalies Detection Models on Embedded Systems in IoT. *Proceedings of the ACM/IEEE Conference on Internet of Things Design and Implementation (IoTDI)*.

[25] Bharath Sudharsan, Simone Salerno, Duc-Duy Nguyen, Muhammad Yahya, Abdul Wahid, Piyush Yadav, John G Breslin, and Muhammad Intizar Ali. 2021. TinyML benchmark: Executing fully connected neural networks on commodity microcontrollers. In *IEEE 7th World Forum on Internet of Things (WF-IoT)*.

[26] Bharath Sudharsan, Dineshkumar Sundaram, John G. Breslin, and Muhammad Intizar Ali. 2020. Avoid Touching Your Face: A Hand-to-face 3D Motion Dataset (COVID-away) and Trained Models for Smartwatches. In *10th International Conference on the Internet of Things Companion*. ACM.

[27] Bharath Sudharsan, Dineshkumar Sundaram, Pankesh Patel, John Breslin, and Muhammad Intizar Ali. 2021. Edge2Guard: Botnet Attacks Detecting Offline Models for Resource-Constrained IoT Devices. *IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*.

[28] Piyush Yadav. 2019. High-performance complex event processing framework to detect event patterns over video streams. In *20th International Middleware Conference Doctoral Symposium*.

[29] Piyush Yadav and Edward Curry. 2019. Vekg: Video event knowledge graph to represent video streams for complex event pattern matching. In *International Conference on Graph Computing (GC)*.

[30] Piyush Yadav and Edward Curry. 2019. Vidcep: Complex event processing framework to detect spatiotemporal patterns in video streams. In *International conference on big data*.

[31] Piyush Yadav, Dibya Prakash Das, and Edward Curry. [n.d.]. State summarization of video streams for spatiotemporal query matching in complex event processing. In *18th International Conference On Machine Learning And Applications (ICMLA)*.

[32] Piyush Yadav, Dhaval Salwala, and Edward Curry. 2021. VID-WIN: Fast Video Event Matching with Query-Aware Windowing at the Edge for the Internet of Multimedia Things. *IEEE Internet of Things Journal*.