Enabling Machine Learning on the Edge Using SRAM Conserving Efficient Neural Networks Execution Approach

Bharath Sudharsan[1](✉), Pankesh Patel[2], John G. Breslin[1], and Muhammad Intizar Ali[3]

[1] Confirm SFI Research Centre for Smart Manufacturing, Data Science Institute, NUI Galway, Galway, Ireland
{bharath.sudharsan,john.breslin}@insight-centre.org

[2] Artificial Intelligence Institute, University of South Carolina, Columbia, USA
ppankesh@mailbox.sc.edu

[3] School of Electronic Engineering, DCU, Dublin, Ireland
ali.intizar@dcu.ie

Abstract. Edge analytics refers to the application of data analytics and Machine Learning (ML) algorithms on IoT devices. The concept of edge analytics is gaining popularity due to its ability to perform AI-based analytics at the device level, enabling autonomous decision-making, without depending on the cloud. However, the majority of Internet of Things (IoT) devices are embedded systems with a low-cost microcontroller unit (MCU) or a small CPU as its brain, which often are incapable of handling complex ML algorithms.

In this paper, we propose an approach for the efficient execution of already deeply compressed, large neural networks (NNs) on tiny IoT devices. After optimizing NNs using state-of-the-art deep model compression methods, when the resultant models are executed by MCUs or small CPUs using the model execution sequence produced by our approach, higher levels of conserved SRAM can be achieved. During the evaluation for nine popular models, when comparing the default NN execution sequence with the sequence produced by our approach, we found that 1.61–38.06% less SRAM was used to produce inference results, the inference time was reduced by 0.28–4.9 ms, and energy consumption was reduced by 4–84 mJ. Despite achieving such high conserved levels of SRAM, our method 100% preserved the accuracy, F1 score, etc. (model performance).

Keywords: Edge AI · Resource-constrained devices · Intelligent microcontrollers · SRAM conservation · Offline inference

1 Introduction

Standalone execution of problem-solving AI on IoT devices produces a higher level of autonomy and also provides a great opportunity to avoid transmitting data collected by the devices to the cloud for inference. However, at the core of a problem-solving AI is usually a Neural Network (NN) with complex and large architecture that demands a higher order of computational power and memory than what is available on most IoT edge devices. Majority of IoT devices like smartwatches, smart plugs, HVAC controllers, etc. are powered by MCUs and small CPUs that are highly resource-constrained. Hence, they lack multiple cores, parallel execution units, no hardware support for floating-point operations (FLOPS), low clock speed, etc.

The IoT devices are tiny in form factor (because FLASH, SRAM, and processor are contained in a single chip), magnitude power-efficient, and cheapest than the standard laptop CPUs and mobile phone processors. During the design phase of IoT devices, in order to conserve energy and to maintain high instruction execution speeds, no secondary/backing memory is added. For example, adding a high-capacity SD card or EEPROM can enable storing large models even without compression. But such an approach will highly affect the model execution speed since the memory outside the chipset is slow and also requires $\approx$100x more energy to read the thousands of outside-located model parameters.

The memory footprint (SRAM, Flash, and EEPROM) and computation power (clock speed and processor specification) of IoT devices are orders of magnitude less than the resources required for the standalone execution of a large, high-quality Neural Network (NN). Currently, to alleviate various critical issues caused by the poor hardware specifications of IoT devices, before deployment the NNs are optimized using various methods [12] such as pruning, quantization, sparsification, and model architecture tuning etc. Even after applying state-of-the-art optimization methods, there are numerous cases where the models after deep compression/optimization still exceed a device's memory capacity by a margin of just a few bytes, and users cannot optimize further since the model is already compressed to its maximum. In such scenarios, the users either have to change the model architecture and re-train to produce a smaller model (wasting GPU-days and electricity), or upgrade the device hardware (for a higher cost).

In this paper, we propose an efficient model execution approach to execute the deep compressed NNs. Our approach can comfortably accommodate a more complex/larger model on the tiny IoT devices which were unable to accommodate the same NNs without using our approach. The contributions of this paper can be summarised as follows:

- Our proposed approach shows high model execution efficiency since it can reduce the peak SRAM usage of a NN by making the onboard inference procedure follow a specific model execution sequence.
- Our approach is applicable to various NN architectures, and models trained using any datasets. Thus, users can apply it to make their IoT devices/products efficiently execute NNs that were designed and trained to solve problems in their use-case. We also implemented and made our approach freely available online.
- When the NNs optimized using state-of-the-art deep compression sequences exceed the device's memory capacity just by a few bytes margin, the users

cannot additionally apply any optimization approach since the model might be already maximum compressed or the users cannot find a study that contains methods compatible to the previous optimizations. In such scenarios, when our approach is used, the same NNs that couldn't fit on the user's device (due to SRAM overflow), can be comfortably accommodated due to the fact that our approach provides a model execution sequence that consumes less SRAM during execution.

- Orthogonal to the existing model memory optimization methods, our approach 100% preserves the deployed model's accuracy since it does not alter any properties and/or parameters of models, neither alter the standard inference software. Instead it instructs the device to just use the SRAM optimized execution sequence it provides.
- Many IoT devices running large NNs fail due to overheating, fast battery wear, and run-time stalling. The prime reason for such failure causing issues is the exhaustion of device memory (especially SRAM). To accurately estimate the memory consumed by models during execution on IoT devices, we provide a Tensor Memory Mapping (TMM) program that can load any pretrained models like ResNet, NASNet, Tiny-YOLO, etc., and can accurately compute and visualize the tensor memory requirement of each operator in the computation graph of any given model. A part of the approach proposed in this paper relies on the high-accuracy calculation results of TMM.

Outline. The rest of the paper is organized as follows; Sect. 2 briefs essential concepts and related studies. In Sect. 3, we present the complete proposed approach, and in Sect. 4, we perform an empirical evaluation that aims to justify the claims of our approach before concluding our paper in Sect. 5.

2 Background and Related Work

In Subsect. 2.1, we present the top deep model compression techniques that produce the smallest possible model, which can be executed on MCUs and small CPUs using our proposed approach. In Subsect. 2.2, we view the trained NN as a graph and explain its standard execution method, followed by the related studies comparable with our model execution approach.

2.1 Deep Model Compression

The approaches in this category employ various techniques to enable fitting large NNs on IoT devices. For instance, *Model design techniques* emphasize designing models with reduced parameters. *Model compression techniques* use quantization and pruning [12] based approaches. Quantization takes out the expensive floating-point operations by reducing it to a Q-bit fixed-point number, and pruning removes the unnecessary connections between the model layers. Other techniques such as layer decomposition [11], distillation [3], binarisation [5] is also applicable. Also, neural architecture search methods [15] can be used to design a network with only a certain floating-point operation count to fit within the memory budget of the MCUs. If users want to achieve a higher level of size reduction, let's assume when they aim to execute models like Tiny-YOLO and Inception v3 (23.9 MB after post-training quantization) on IoT devices, we recommend performing *Deep Model Compression*. Here the users, in a sequence, have to realize more than one of the briefed model optimization techniques.

After following the deep optimization sequence of their choice, the NNs become friendly enough to be executed on tiny devices. Additionally, when such deep optimized models are executed using our proposed approach, its peak on-device execution memory usage can be reduced.

2.2 Executing Neural Networks on Microcontrollers

A neural network is a graph with defined data flow patterns having an arrangement of nodes and edges, where nodes represent operators of a model, and graph edges represent the flow of data between nodes. The operator nodes in the model graph can be 2D convolutions (Conv2D), or Depthwise separable 2D convolution (DepthwiseConv2D), etc. These operator nodes can take more than one input to produce an output. Recently, a few ML frameworks have released tools to optimize model graphs in order to improve the execution efficiency of NNs. For example, the optimizer tool fuses adjacent operators and converts batch normalization layers into linear operations. In such model computation graphs, buffers are used to hold the input and output tensors before feeding them to the operators during the model execution. After execution, the items in the output buffer will be provided as input to the next operator, and the input buffers can be reclaimed by removing the stored data.

Structure of Computation Graphs. When executing a model, the graph nodes in both the regular graph and its optimized version are executed one by one in a topological fashion/order. For example, the VGG and AlexNet iteratively apply a linear sequence of layers to transform the input data. But, similar to the computation graph shown in Fig. 1, the newer networks like the Inception, NasNet, and MobileNet, etc. are non-linear as they contain branches. For these networks, the input data transformation is performed in divergent paths because the same input is accessible by numerous operators present in several layers i.e., the same input tensors are accessible for processing by several layers and operators. Hence when executing such branched models on MCUs, the execution method can have access to multiple operators.

Mapping Models on the MCU Memory. The typical small CPUs and MCUs based IoT devices have their on-chip memory partitioned into SRAM (read-write) and NOR-Flash (read-only). The complete memory requirement of a NN is mapped to these two partitions. Since SRAM is the only available read-write space, the intermediate tensors generated during model execution are stored here, increasing the peak SRAM usage on MCUs. The model parameters such as trainable weights, layers, constants, etc., do not change during the run-time (immutable in nature). Hence, they are converted into hex code and stored in the static Flash memory along with the application of the IoT use case.

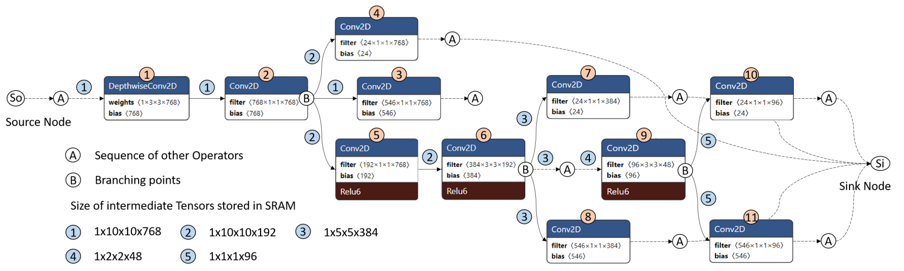

Fig. 1. A part of the COCO SSD MobileNet computation graph with its branched operators: When executing such graphs on IoT devices, our approach reduces the peak SRAM consumption by producing an optimized operators execution sequence. (Color figure online)

The most relevant work to ours are [10] and [9], where a NN execution runtime for MCUs is attached with their NAS. Next is the [1], which proposes a method for optimizing the execution of a given neural network by searching for efficient model layers. i.e., a search is performed to find efficient versions of kernels, convolution, matrix multiplication, etc., before the C code generation step for the target MCU. Both the methods aim to ease the deployment of NNs on MCUs, whereas our approach is to take any deep compressed model and during execution reduce its peak SRAM usage.

3 Efficient Neural Network Execution Approach Design

As discussed earlier, the trained model size and its peak SRAM need to be highly reduced due to the limited Flash and SRAM memory capacity of IoT devices. Here, we present our approach that can reduce the peak SRAM consumed by neural networks. We first describe our Tensor Memory Mapping (TMM) method in Subsect. 3.1. Then in Subsect. 3.2 and 3.3, we present the two parts of our proposed approach, followed by Subsect. 3.4 that combines both the parts and presents the complete approach in the form of an implementable algorithm.

3.1 Tensor Memory Mapping (TMM) Method Design

Before deployment, the memory requirement of models is often unknown or calculated with less accuracy. i.e., there will exist a few MB of deviations in the calculations. When the model is targeted to run on better-resourced devices like smartphones or edge GPUs, these few MB deviations do not cause any issues. But when users target the resource-constrained IoT devices (has only a few MB memory), then the low-accuracy calculation causes run-time memory overflows and/or restrict flashing model on IoT devices due to SRAM peaks. Based on our recent empirical study, we found that many IoT devices that are running large NNs fail due to overheating, fast battery wear, run-time stalling. The prime reason for such failure causing issues is the exhaustion of device memory (especially SRAM). Hence, this inaccurate calculation leads to a horrendous computing resource waste (especially the GPU days) and reduced development productivity. In this section, we thereby present our tensor memory mapping method, which can be realized to accurately compute and visualize the tensor memory requirement of each operator in any computation graph. We use this high-accuracy calculation method in the core algorithm design of our efficient neural network execution approach.

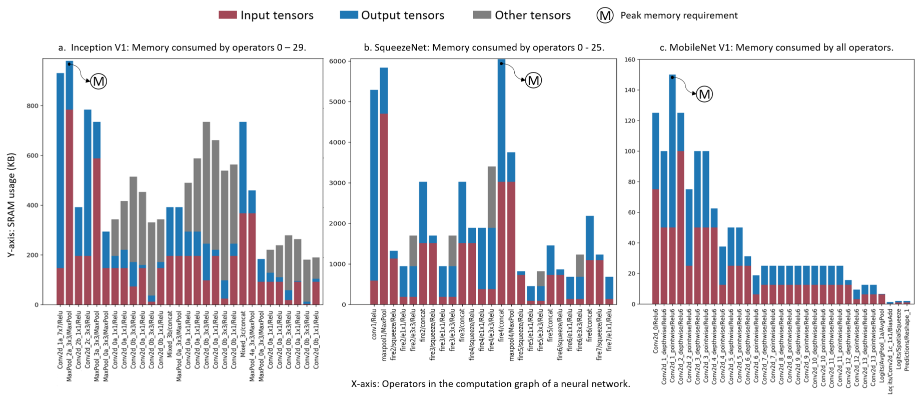

Fig. 2. Accurate computation and visualization of tensor memory requirement for each operator in NN computation graphs (performed using our TMM): The Algorithm 1 reduces the shown memory peaks by reordering operators to produce a new graph execution sequence.

Abstraction and Formalization. We treat the internal of neural networks as mathematical functions and formalize it as *tensor-oriented computation graphs* since the inputs and outputs of graph nodes/operators are a multi-dimensional array of numerical values (i.e., tensor variables). The shape of such a tensor is the element number in each dimension plus element data type. In the below equation, we formally represent a NN as a Directed Acyclic Graph (DAG), and we treat its execution as iterative forward and backward propagation via the graph branches.

$$NN_{DAG} = \langle \{op_i\}_{i=1}^{n}, \{(op_i, op_j)\}, \{p_k\}_{k=1}^{m} \rangle \tag{1}$$

Here op_i are the graph operators, (op_i, op_j) is the connection to transmit output tensor from op_i as an input to op_j, and there are m hyperparameters p_k. Let the topological ordering of operators be $Seq = \langle op_{i_1}, op_{i_2}, \cdots, op_{i_n} \rangle$ that extends from the first graph edge such that $op_{i_i} <_{Seq} op_{i_k} \rightarrow \left(op_{i_k}, op_{i_j}\right) \notin NN_{DAG}$, where Seq is the operator execution sequence (we aim to find a memory friendly sequence in the later sections). In this graph, when visiting a node op, we need to calculate the memory it consumes to store (i) newly assigned tensors, (ii) previously assigned but still in-use tensors, (iii) reserved buffers. To calculate the memory consumption $M_{NN_{DAG}}$ of a graph NN_{DAG} we give the following formulae. We call the first two types of tensors as unreleased tensors.

$$M_{NN_{DAG}} = \max \left\{ MF_{n_{init}}, MF_n\left(op_i\right) \mid op_i \in NN_{DAG} \right\} \tag{2}$$

Here, $MF_{n_{init}} = \sum MT_{sr}(t)$ is the function to compute the initial memory consumption, $MF_n(op) = MU_{res}(op) + MR(op)$ is the current memory consumption, $MU_{res}(op) = \sum_{t \in U_{res}T_{sr}(op)} MT_{sr}(t)$ is the function that computes memory requirement of unreleased tensors, $MR(op)$ function returns memory size of reserved buffers. The set of unreleased tensors are computed using $U_{res}T_{sr}$, and for a given tensor t, function MT_{sr} is used to find its allocated memory size. The Eq. 2 applies to models trained using any ML frameworks like TensorFLow, PyTorch, etc. to estimate the graph memory consumption, and applicable to calculate the memory requirements for any operators execution sequence.

Testing the Design. The implementation of our method is suitable for any pre-trained models like NASNet, Tiny-YOLO, SqueezeNet, etc. For each of the operators in any given model graph, our method computes the total required SRAM. i.e., the space required to store the input tensors + output tensors + other tensors, and then exports the detailed report in CSV format. Our method can also produce images that show the tensor memory requirement of each operator. For example, when we feed the Inception V1 that contains 84 graph nodes/-operators to our method, it produces Fig. 2a. (for brevity, we show only 0–29 operators) along with the detailed CSV report. Similarly, we test our method on SqueezeNet and MobileNet V1 and show the results in Fig. 2b–c. Thus by enabling visualization, our method helps users analyze multiple memory aspects of networks and obtain valuable insights that can guide them to customize their model graph for highly reduced memory. For example, we made the following observations; (i) Most of the Inception V1 nodes consume high memory to accommodate other tensors, whereas the MobileNet does not contain other tensors at all; (ii) Three nodes in SqueezeNet consume significantly higher memory than other nodes. Such nodes can be replaced with cheaper operators that perform the same tasks.

3.2 Loading Fewer Tensors and Tensors Re-usage

In the traditional model execution methods, multiple tensors of various sizes are loaded into the buffer (such bulk loading is the reason for causing peak memory usage) since the traditional methods execute operators requiring different size tensors. In contrast, our approach executes many operators by just loading a minimum number of tensors. This part of our approach also aims to achieve **SRAM conservation by tensors re-usage**. Here, our approach identifies and stores a particular set of tensors in the buffer (buffers are created within SRAM) and first executes the branch of the graph containing operators compatible with the stored tensors. Then in the next iteration, it loads tensors that are suitable as input for the set of operators belonging to the next branch, then performs the execution. After each iteration, the buffers are reclaimed.

For illustration purpose, in Fig. 1, the intermediate tensors of varying size are shown in blue circles ① to ⑤ which need to be stored in SRAM during the graph execution. Here, at the first branching point circled Ⓑ, when the default model execution software is utilized, the two tensors with blue circles ① and ② are loaded on the SRAM. Then it executes all the branched operators with rose circles ③ to ⑤. This method of loading many tensors and executing many operators leads to the most critical SRAM overflow issue, especially in the scenarios where multiple branches are emerging from one branching point.

3.3 Finding the Cheapest NN Graph Execution Sequence

The computational graphs of models perform the inference tasks in a collection of computational steps, where each step depends on the output from a few of the preceding steps. For example, in the graph of MobileNet shown in Fig. 1, these graph steps are the operators and rose circled ① → ② means the second operator depends on the output of the first. Since the computation graphs of most NNs are DAGs, we can enumerate orders/sequences to execute all the operators/computational steps.

As shown in Fig. 1, the modern NNs like MobileNet have divergent data flow paths. i.e., their computation graphs contain branches. As briefed in Subsect. 2.2, due to such a branched design, a given tensor can be accessed by operators in various branches. For example, in Fig. 1, the tensor with a blue circle ① of size $1 \times 10 \times 10 \times 768$ can be accessed by three Conv2D operators due to the presence of a branching point circled Ⓑ. Similarly, the tensor with blue circle ③ of size $1 \times 5 \times 5 \times 384$ is accessible by two Conv2D layers and by another sequence of operators circled Ⓐ. Such branched computation graphs provide freedom for the model execution software to alter the execution order/sequence of the operators. In the rest of this section, we show that any topological execution order of the graph nodes will result in a valid execution scheme; we then explain how our approach leverages this freedom to achieve its SRAM conservation goal.

Does Any Topological Execution Order of the NN Graph Nodes Result in a Valid Execution Scheme? DAGs of models have topological ordering and do not have cycles because the edge into the earliest vertex of a cycle would have to be oriented the wrong way. Therefore, every graph with a topological ordering is acyclic. But for a directed graph that is not acyclic, there can be more than one minimal subgraph with the same reachability relation. Where, for a complete DAG with N nodes, the search space contains $2^{N(N-1)/2}$ possible topological orders/structures. We consider the initial graph node as the *source node*, where the input data (i.e., sensor values that require predictions) is fed into the network, and the ending node as the *sink node*, where the inference results are transmitted to control the real world applications. In the graph of MobileNet from Fig. 1, let us assume the source node to be circle $\textcircled{\text{So}}$ and the sink node to be circle $\textcircled{\text{Si}}$. Since NNs are DAGs, the topological execution numbers assigned to the nodes (operators) increases along the branched path (without forming any cycles) till the sink node. During this coverage, no graph vertices or nodes are skipped.

Formally, we define thus described topological process as $G_0 = (V,\ E)$, with operators $V = \{v_1,\ v_2,\ v_3,\ \ldots,\ v_{n-1},\ v_n\}$ and E are the edges between operators. Here the operator execution order is a sequence containing all the operators $\in V, \{v_{k_1},\ v_{k_2},\ \cdots,\ v_{k_{n-1}},\ v_{k_n}\}$ such that for all $i,\ j$ $(0 \leqslant i, j \leqslant n)$, if there exists a path from v_{k_i} to v_{k_j}, then $i < j$. Briefly, if there is a path from operator v to operator w, then in the execution sequence, v should be set to be executed before w. Hence, the directed computation graph of NNs is a DAG *if and only if* it has a topological ordering. This explanation gives us two independent statements to prove; **First** we need to show if a directed graph follows a topological ordering of operator nodes, it is a DAG. **Second**, we need to show that all DAGs follows a topological ordering of operator nodes.

Proof One. Since a biconditional logical connective exists between the above two statements, either both statements are true or both are false. Hence, proving either the first or the second statement will suffice both. By contrapositive; if we prove that *if a NN graph is not a DAG, it can not have a topological ordering*, we can satisfy the first statement. In the following, we prove this.

When we assume the computation graph of a NN to not be a DAG, there will exist cyclic data flow between operators in the graph. For example, in $\{v_1, v_2, \cdots, v_k, v_1\}$, since there is a path from v_1 to v_2, the operator v_1 must appear before v_2 in the topological ordering scheme. But there is also a path from v_2 to v_1 via v_k making v_2 appear before v_1. If we implicate this scenario in Fig. 1, the execution sequence reaches the sink node (v_k) and then returns back to the source node (v_1), clearly voiding the main ordering principle of a DAG, hence proving the first statement. In the following, we also prove the second statement, but by induction.

Proof Two. We start to prove the second statement in *step one*. Here, we define the base case, which is a graph with just one operator. This graph is a DAG with topological ordering since the execution order starts from the source node, travels via the single operator, and ends at the sink node. In *step two*, we consider a topologically ordered DAG with multiple operators connected by n vertices as the induction hypothesis. In order to prove the second statement, for this induction step two, we need to show that the induction hypothesis implies that a DAG with $n+1$ vertices must have a topological ordering. To prove this, in *step three*, we take a NN graph with $n+1$ vertices/operators having one 0-degree vertex v_0. In *step four*, we remove the 0-degree vertex to obtain a computation graph with n vertices (similar to graph from step two). This resulting graph must be a DAG since the base graph from step two had no cycles, and also, in this step, we removed edges (not added).

According to the induction hypothesis from step two, since the resultant graph from step four is a DAG with n vertices, it also will have a topological ordering. Thus, a topological operators execution sequence can be constructed for the graph from step three that has $n+1$ operators, by prepending v_0 to the topological order of the n vertices DAG from step two.

Algorithm 1. Reducing the peak SRAM consumption by discovering an optimized operators execution sequence.

```
1: Input: Computation graph of the trained model.
2: Output: Cheapest graph execution order with reduced peak SRAM requirement.
3: $const_{tens}$    ▷ Constant tensors
4: $active_{tens}$    ▷ Active tensors that change during graph execution
5: $set_{tens}$    ▷ Set of tensors
6: $rem_{tens}$    ▷ Variable to store the remaining tensors
7: $req_{tens}$    ▷ Tensors required to produce $tens$
8: operator ($tens$)    ▷ The operator that computes to produce tensors $tens$ and $set_{tens}$
9: $k \leftarrow \infty$, $s \leftarrow 0$, $k' \leftarrow 0$    ▷ Variables
10: memory reduction    ▷ Function to find the path that consumes minimum memory to compute all $tens \in set_{tens}$
11:     $const_{tens}$, $active_{tens}$ ← Separate ($set_{tens}$, $tens$ : operator ($tens$) is none)    ▷ Separate constant and active tensors
12:     if no $active_{tens}$ then
13:         return $\sum_{s \in const_{tens}} |c|$    ▷ No remaining operators to reorder. Send sizes of remaining $const_{tens}$
14:     end if
15:     for $tens$ in $active_{tens}$ do
16:         $rem_{tens} \leftarrow active_{tens}$    ▷ Remaining tensors need to be stored in memory
17:         $req_{tens}$ ← operator ($tens$) . $data$
18:         if any ($tens$ is used to produce $rem$ where $rem \in rem_{tens}$) then
19:             $tens$ was used to produce $rem$. So in the future, the operator ($tens$) will be executed    ▷ Result stored for re-use
20:         end if
21:             ▷ At this stage, peak memory will be consumed either by; (i) the operator ($tens$) that produced $rem$. In this case the peak is the memory of input tensors + output tensor + other tensors. (ii) other operators. i.e., recursive case memory reduction ($rem_{tens} \cup req_{tens}$)
22:             $k'$ ← max (memory reduction ($rem_{tens} \cup req_{tens}$), $\sum_{t \in rem_{tens} \cup req_{tens} \cup \{tens\}} |t|$)
23:         $k \leftarrow \min(k, k')$
24:             ▷ The cheapest graph execution order/path is decided here
25:     end for
26: return $\sum_{rem \in rem_{tens}} |rem| + k$
```

SRAM Conservation by Altering Operators Execution Sequence. Having proved that changing execution sequence of operators still produces a valid scheme; our approach achieves its memory conservation goal by intelligently selecting the execution branch that when executed consumes less SRAM (reduces the peak memory consumption) than the default sequence. For illustration purpose, in Fig. 1., if the model execution software follows the default operators execution order; the execution will start at the operator with a rose circle ① and follow the sequence till the operator with a rose circle ⑧, in the order of 1, 2, 3, 4, 5, 6, 7, 8. This unoptimized default order will consume a peak SRAM of 5900 Bytes. Whereas when our efficient execution approach is utilized, the operator execution order is altered to form a new sequence that will require a reduced SRAM of 5200 Bytes. This new order will be 1, 2, 5, 6, 7, 8, 3, 4. Here, the calculated SRAM consumption/requirement is the sum of the size of tensors stored in the operator's input and output buffers added with the tensor size of the output of previous or next operators. As explained in Subsect. 2.2, this third set of stored tensors are the input for the other operators that exist in the graph.

3.4 Core Algorithm

Discovering multiple topological orders of nodes in a computation graph belongs to the literature of graph optimization. The algorithm that we present in this section belongs here since we designed it considering the computation graph of a model as a DAG, and as proved in Subsect. 3.3, the execution of available nodes in any topological order will result in a valid execution sequence. When the computation graph of any given model is loaded into our algorithm, it analyzes the complete network by running through each branch of the network and finally discovering the cheapest graph execution path/sequence. The time consumed by the algorithm to produce the results depends on complexity $\mathcal{T}_{\rfloor}\left(|O|2^{|O|}\right)$, where $|O|$ is the total operators count. Since the latest network architectures contain hundreds of operators, our proposed algorithm is best-suited to run on better-to-high resource devices such as laptop CPUs. Our algorithm-generated optimized graph execution sequence should be used by the inference software when executing the target model on MCU-based IoT devices.

We present our complete approach in Algorithm 1. Here, Lines 10 to 25 is the core *memory reduction* function of our algorithm that performs the required tasks to reduce the peak SRAM usage by reordering operators to produce a new execution sequence. Before the core function, in Line 3 to 9, we declare all the function required variables. In Line 11, we remove the tensors that shall not be used as inputs by the operators in the graph. Also, the tensors that do not contain the operators that produced are taken out. Thus performed removal actions do not affect the model performance since the removed tensors are constant $const_{tens}$.

Next, in Line 18 to 20, we ensure that no operator nodes are executed twice. This is done by checking whether an operator node has produced any of the tensors (rem_{tens}) that are remaining after taking out $const_{tens}$. If such tensors are existing, in the future, the inference software might require to execute again the operators that produced those tensors. To conserve memory, in Line 19, the results of such operators that need to be re-executed are stored in the buffer for reuse. In fact, such re-execution can cause memory peaks. In Line 22, the *memory reduction* function is called multiple times in order to cover all the branches of the computation graph. Finally, in Line 26, the cheapest graph execution path is returned. When executing the thus produced reordered operators sequence on IoT devices, if the scope of loaded tensors is over, we recommend the inference software to reclaim the memory used by such tensors by removing them from the SRAM.

Table 1. Executing original models and its Algorithm 1 optimized versions: Comparing the peak SRAM usage, inference time, and the energy consumed for inference.

Model task/ category	Pre-trained model name	Quantized model without optimization			Quantized model with optimization using Algorithm 1		
		Peak SRAM usage (KB)	Inference time (ms)	Energy used (mJ)	Peak SRAM usage (KB)	Inference time (ms)	Energy used (mJ)
Image classification	MobileNetV1 [7]	98.304	1.6	27.59904	65.536(32 ↓)	0.96 (0.64 ↓)	16.55942 (11 ↓)
	SqueezeNet [6]	6195.200	12.4	213.8926	4816.896 (1378 ↓)	10.62 (1.78 ↓)	183.1886 (30.7 ↓)
	InceptionV1 [13]	1003.520	43.6	752.0738	802.816 (200 ↓)	38.9 (4.7 ↓)	671.0017 (81.0 ↓)
	MnasNet [14]	1605.632	7.4	127.6456	1204.224 (401 ↓)	5.7 (1.7 ↓)	98.32158 (29.3 ↓)
	NASNet mobile [17]	4511.660	63	1086.712	3834.284 (677 ↓)	61.2 (1.2 ↓)	1055.663 (31 ↓)
	DenseNet [4]	8429.568	246.3	4248.527	5221.264 (3208 ↓)	241.4 (4.9 ↓)	4164.005 (84 ↓)
Semantic segmentation	DeepLabv3 [2]	5639.592	38.2	658.927	5548.116 (91 ↓)	37.07 (1.13 ↓)	639.435 (19 ↓)
Pose estimation	PoseNet [8]	6575.904	22.3	384.661	4383.936 (2191 ↓)	19.4 (2.9 ↓)	334.638 (50 ↓)
Text detection	EAST [16]	5324.800	43.38	748.278	3686.400 (1638 ↓)	43.10 (0.28 ↓)	743.449 (4 ↓)

4 Experimental Evaluation

In this section, we perform an empirical evaluation to answer the following questions.

- To what levels can the proposed approach increase the model execution efficiency by reducing the peak SRAM usage of NNs?
- Is the approach suitable to diverse NN architectures and NNs trained using various datasets?
- Can the approach produce an optimized operators execution sequence for already optimized or deep compressed models?
- Does optimization using the proposed approach impact the accuracy or performance of the model?

We start the evaluation by downloading popular pre-trained TensorFlow Lite models (`.tflite` format) from TensorFlow Hub. For comprehensiveness, the models selected to evaluate our approach belong to various problem domains ranging from image classification to text detection and are listed in Table 1. As described in Subsect. 3.4, since the chosen models contain hundreds of operators, the complexity of our algorithm will be high. Hence, we conduct the evaluation on a standard NVIDIA GeForce GPU-based Ubuntu laptop with Intel (R) Core (TM) i7-5500 CPU @ 2.40 GHz. After the download, we first load and execute each model on the same laptop using the default execution sequence of operators and tabulate the corresponding peak SRAM usage, unit inference time, and the energy consumed to execute the model and perform inference.

In the same setup, we next apply the implementation of Algorithm 1 on each model and tabulate the obtained results in Table 1, next to the results obtained when executing models using their default execution sequence. During the evaluation, for statistical validation, the reported inference time and the consumed energy corresponds to the average of 5 runs. In order to perform analysis, in Table 1, we subtract the values reported under *Quantized Model with Optimization using Algorithm 1* with values under *Quantized Model without Optimization* and plot bar-graphs for each model in Fig. 3. Based on this, in the remainder subsections, we analyze and discuss the benefits achieved as a result of optimizing models using our proposed approach.

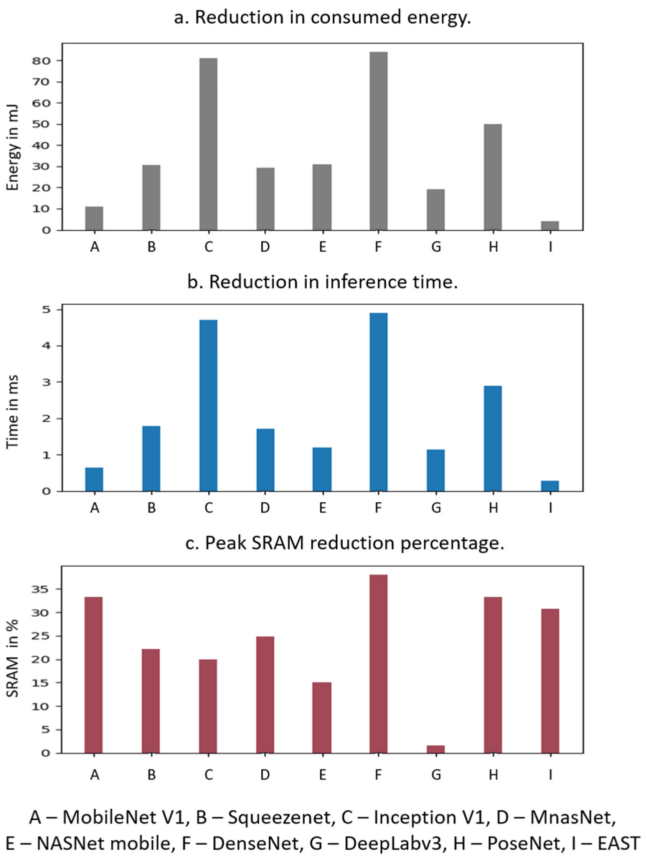

Fig. 3. Benefits achieved after optimization using our proposed approach.

4.1 SRAM Usage

In practice, there are many cases where ML models optimized using state-of-the-art deep compression sequences exceed the target device's SRAM capacity just by a few KB margin. In such cases, users cannot additionally apply any optimization approach since it might not match the previous optimizer components, or the model might already be maximum compressed. So they either have to alter the model architecture and re-train to produce a smaller model (waste of GPU days and electricity) or upgrade the IoT device hardware (loss of money). In the remainder of this section, we show how our approach can enable the accommodation and execution of memory overflow issues causing models on IoT devices.

We take the quantized DenseNet with its default execution sequence and feed it to our TMM program from Sect. 3.1. From the resultant computed memory requirement for each operator in the default graph, the 24^{th} operator showed the peak SRAM consumption of 8429.568 KB. Next, after applying our Algorithm 1 on DenseNet, the resultant memory-friendly graph execution sequence, when evaluated by the TMM program, showed the peak memory of only 5221.264 KB (peak reduced by 38.06%).

Similarly for MobileNet V1, the peak SRAM usage reduced from 98.304 KB to 65.536 KB (see Table 1). Here our approach has reduced the memory peak by 32.76 KB (by 33%). In Fig. 3c, we plot thus calculated peak SRAM reduction percentage for MobileNet V1 (label A in x-axis) and the remaining 8 models selected for evaluation. The maximum peak SRAM reduction of 38.06% was achieved for the DenseNet and the least of 1.61% reduction for DeepLabv3. It is apparent from the results that the execution sequence produced by our approach is applicable for a wide range of ML models that have diverse network architectures. Also, since it reduces the SRAM peaks, the models that are still large after optimization can be accommodated on tiny IoT devices. Thus, our approach eliminates the re-training step that aims to produce small models, and also, the device hardware need not be upgraded to accommodate the models.

4.2 Model Performance

As a part of experimental results, we report that despite the SRAM conservation, the model executed using the SRAM optimized sequence provided by Algorithm 1 showed the same performance (accuracy, F1 score, etc.) as the models when executed with their default sequence. This is because, unlike existing methods, ours does not alter any properties/parameters of models, neither alter the standard inference software (just instructs to use a different model execution sequence). Also, as proved in Sect. 3.3, the SRAM optimized sequence produced by our approach is a valid model execution sequence. This 100% model performance preservation characteristics enable even tiny IoT devices to produce high accuracy offline analytics results.

4.3 Inference Time and Energy Consumption

Here in order to investigate the impact of our approach on inference/model execution performance, we execute each model first with their default execution sequence, then with the memory peak reduced sequence produced by our approach. We report the difference in inference time and consumed energy for both default and optimized sequence in Table 1 and show it in Fig. 3a–b. For the same tasks performed on the same device using the same datasets, the new graph execution sequence for DenseNet shows the maximum inference time reduction of 4.9 ms and the least of 0.28 ms reduction for EAST. We also achieved 4–84 mJ less energy to perform unit inference since executing the model using the SRAM optimized sequence produced by our approach is 0.28–4.9 ms faster than the default sequence.

In realistic scenarios, to infer using a stream of data input, the deployed model is executed in a loop. Here, even the minor inference speedups and energy conservation produced by our approach get multiplied, driving the IoT devices close to producing real-time edge analytics results at a lower power cost. Thus, even the autonomous tiny IoT devices can efficiently control real-world IoT applications by making timely predictions/decisions and also perform offline model inference without affecting the operating time of battery-powered devices.

5 Conclusion

In this paper, we presented an approach to efficiently execute (with reduced SRAM usage) deeply optimized (maximally compressed) ML models on resource-constrained devices. For nine popular models, when comparing the default model execution sequence with the sequence produced by our approach, we showed that 1.61–38.06% less SRAM was used to produce inference results, the inference time was reduced by 0.28–4.9 ms, and energy consumption was reduced by 4–84 mJ. As well as achieving highly conserved SRAM levels, our method 100% preserved the model performance. Thus, when users apply the approach presented in this paper, they can: (i) Execute large-high-quality models on their IoT devices/products without needing to upgrade the hardware or alter the model architecture and re-train to produce a smaller model; (ii) Devices can control real-world applications by making timely predictions/decisions; (iii) Devices can perform high accuracy offline analytics without affecting the operating time of battery-powered devices.

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

References

1. Tinyml - how TVM is taming tiny. https://tvm.apache.org/2020/06/04/tinyml-how-tvm-is-taming-tiny
2. Chen, L.C., Papandreou, G., Schroff, F., Adam, H.: Rethinking atrous convolution for semantic image segmentation. arXiv preprint arXiv:1706.05587 (2017)
3. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 (2015)
4. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2017)
5. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. In: Advances in Neural Information Processing Systems (2016)
6. Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., Keutzer, K.: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 mb model size. arXiv preprint arXiv:1602.07360
7. Jacob, B., et al.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2018)
8. Kendall, A., Grimes, M., Cipolla, R.: PoseNet: a convolutional network for real-time 6-DOF camera relocalization. In: Proceedings of the IEEE International Conference on Computer Vision (2015)
9. Liberis, E., Dudziak, Ł., Lane, N.D.: μNAS: constrained neural architecture search for microcontrollers. arXiv preprint arXiv:2010.14246
10. Lin, J., Chen, W.M., Lin, Y., Cohn, J., Gan, C., Han, S.: MCUNet: tiny deep learning on IoT devices. arXiv preprint arXiv:2007.10319 (2020)
11. Qiu, Q., Cheng, X., Calderbank, R., Sapiro, G.: DCFNet: deep neural network with decomposed convolutional filters. arXiv preprint arXiv:1802.04145 (2018)
12. Sudharsan, B., Breslin, J.G., Ali, M.I.: RCE-NN: a five-stage pipeline to execute neural networks (CNNs) on resource-constrained IoT edge devices. In: Proceedings of the 10th International Conference on the Internet of Things (2020)
13. Szegedy, C., et al.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2015)
14. Tan, M., e al.: MnasNet: platform-aware neural architecture search for mobile. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2019)
15. Tan, M., Le, Q.V.: EfficientNet: rethinking model scaling for convolutional neural networks. arXiv preprint arXiv:1905.11946 (2019)
16. Zhou, X., et al.: East: an efficient and accurate scene text detector. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2017)
17. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2018)