TMM-TinyML: Tensor Memory Mapping (TMM) Method for Tiny Machine Learning (TinyML)

Bharath Sudharsan
GMCI, General Motors
Limerick, Ireland
bharath.sudharsan@gm.com

Sonu Prasad
GMCI, General Motors
Limerick, Ireland
sonu.k.prasad@gm.com

Dan Jose
GMCI, General Motors
Limerick, Ireland
dan.jose@gm.com

John G. Breslin
Data Science Institute
NUI Galway, Ireland
john.breslin@nuigalway.ie

ABSTRACT

TinyML: Tiny in size, big in impact! In this paper, we present a Tensor Memory Mapping (TMM) method, which can accurately calculate the on-device execution memory consumed by a range of ML and TinyML models during execution on small central processing units (CPUs), microcontroller units (MCUs), and single board computers (SBCs).

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; • **Computing methodologies** → **Machine learning**; **Artificial intelligence**.

KEYWORDS

Edge Computing, IoT Devices, Machine Learning, TinyML

ACM Reference Format:
Bharath Sudharsan, Sonu Prasad, Dan Jose, and John G. Breslin. 2022. TMM-TinyML: Tensor Memory Mapping (TMM) Method for Tiny Machine Learning (TinyML). In *The 28th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '22), October 17–21, 2022, Sydney, NSW, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3495243.3558265

1 INTRODUCTION

Standalone execution of problem-solving ML and TinyML models on IoT devices produces a higher level of autonomy and also provides an opportunity to avoid transmitting data collected by the devices to the cloud for inference [1]. However, the core of a problem-solving ML model can be 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 [2]. To alleviate various critical issues caused by the poor memory specifications of IoT devices, before deployment the NNs are deeply optimized [3] using various methods such as pruning, quantization, sparsification, model architecture tuning, etc.

NNs can be viewed as 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 [4]. The operator nodes in the model graph can be 2D convolutions (Conv2D), depthwise separable 2D convolutions (DepthwiseConv2D), Maximum Pooling (MaxPool), etc. These operator nodes can take more than one input to produce an output. In such model computation graphs, buffers hold the input, intermediate, and output tensors before feeding them to the operators during the model execution. After execution, the items in the output buffer will be fed as input to the next operator, and the input buffers can be reclaimed by removing the stored data. In IoT devices, random access memory (RAM) or static RAM (SRAM) is the only available fast read-write space [5]. So the tensors generated during ML model execution are stored here, increasing the RAM/SRAM consumption, which should be aimed to reduce via applying optimization methods.

Based on recent empirical studies, many IoT devices, although running the deeply optimized version of 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 RAM/SRAM) [5]. Before deployment, the execution memory requirement of ML models is often unknown or *calculated with less accuracy*. i.e., there will exist a few MB deviations in the calculations. When the model is targeted to run on better-resourced devices like smartphones and SBCs (like Raspberry Pi, Google Coral Dev Board, BeagleBone AI, and Jetson Nano), these few MB deviations do not cause any issues. But when developers target the small CPU or MCU based IoT devices (with only a few MB of memory), the low-accuracy calculation can cause run-time memory overflows and/or restricts the flashing of models on IoT devices due to RAM/SRAM peaks. In such cases where the target hardware fails to accommodate the ML model, developers either have to alter the model architecture and retrain to produce a lower memory-consuming model (waste of GPU days and electricity) or upgrade the IoT device hardware (loss of money). Hence, there is a need for a method that developers can use during the ML model design phase to exactly know how much memory their model demands when executing on a device.

In this paper, we thereby present a Tensor Memory Mapping (TMM) method for ML and TinyML models, which can

ACM MobiCom '22, October 17–21, 2022, Sydney, NSW, Australia
2022. ACM ISBN 978-1-4503-9181-8/22/10…$15.00
https://doi.org/10.1145/3495243.3558265

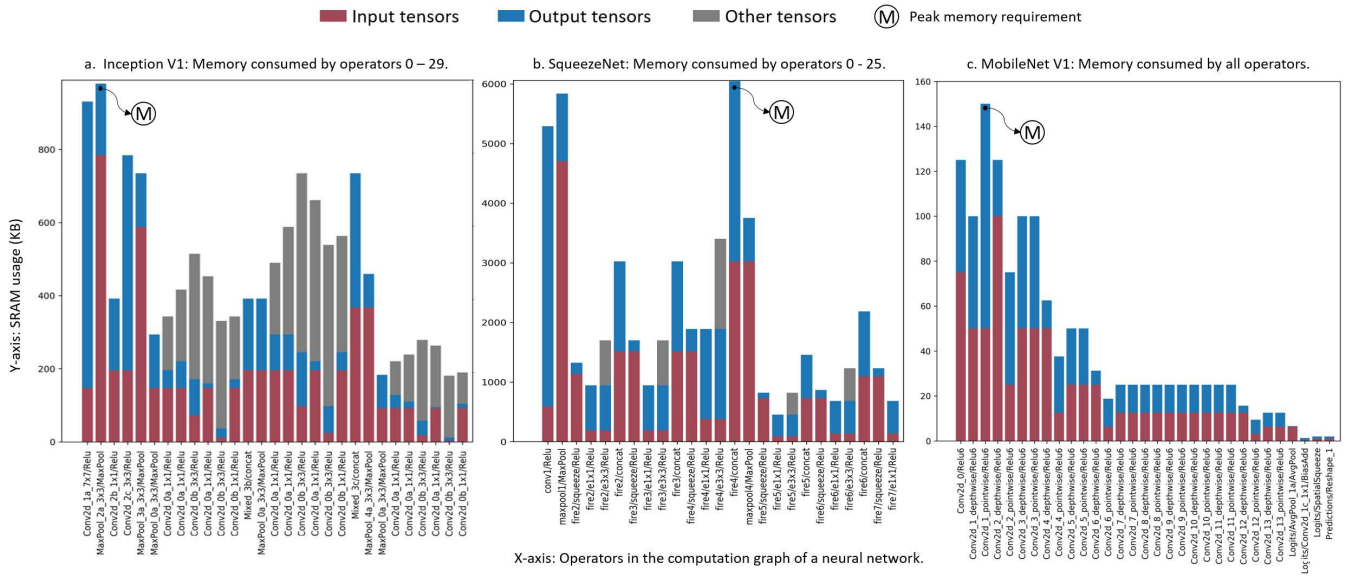

Figure 1: Accurate calculation and visualization of tensor memory requirement for each operator in neural network computation graphs - performed using the proposed TMM method.

be realized to accurately estimate the memory consumed by models during execution on IoT devices. The contributions of this paper can be summarised as follows:

- TMM is compatible with a broad range of pre-trained models like MicroNet, Wav2letter, MobileNet, ResNet, NASNet, Tiny-YOLO, etc., that are available for off-the-shelf usage in repositories such as ARM Model Zoo, TensorFlow Hub, etc.
- TMM can accurately calculate and visualize the tensor memory requirement of each operator in the computation graph of the given NN model.
- The results produced by TMM can enable researchers and engineers to analyze multiple memory aspects to obtain valuable insights that can guide them to customize their ML model for highly reduced memory.

2 TMM METHOD DESIGN

We treat the internal of NNs 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 the 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.

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

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 (developers can aim to find a memory friendly sequence). 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, and (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 unreleased tensors.

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

Here, $MF_{n_{init}} = \sum MT_{sr}(t)$ is the function to compute 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, the function MT_{sr} is used to find its allocated memory size.

When running a model, its graph nodes are topologically executed one by one. For example, the VGG and AlexNet iteratively apply a linear sequence of layers to transform the input data. But, the newer networks like Inception, NasNet, MobileNet are non-linear as they contain branches. For these networks, the input data transformation is performed in divergent paths because the same input tensors are accessible for processing by several layers and operators. Hence when executing such branched models, the execution method has

Table 1: Peak memory requirement of popular off-the-shelf models - computed using the TMM method.

Model Task/Category	Pre-trained Model Name	Peak RAM/SRAM Usage (KB)
Image Classification	MobileNetV1	98.304
	SqueezeNet	6195.200
	InceptionV1	1003.520
	MnasNet	1605.632
	NASNet Mobile	4511.660
	DenseNet	8429.568
Semantic Segmentation	DeepLabv3	5639.592
Pose Estimation	PoseNet	6575.904
Text Detection	EAST	5324.800

access to multiple operators. The Eqn 2 applies to models trained using any ML frameworks (like TensorFLow, PyTorch) to estimate the graph memory consumption. It can also calculate the memory requirements for branched NNs that have multiple operators execution sequences.

3 TMM METHOD TESTING

The TMM method implementation is suitable for pre-trained models (both quantized and unquantized versions). For each of the operators in any given model graph, TMM computes the total required RAM/SRAM. i.e., the space required to store the *input tensors + output tensors + other tensors*, then exports the detailed report in CSV format.

We start the testing by downloading popular pre-trained TensorFlow Lite models (.tflite format) from TensorFlow Hub. For comprehensiveness, the models selected belong to various problem domains ranging from image classification to text detection. Since the chosen models contain hundreds of operators, the complexity of the TMM method will be high. Hence, the testing is conducted on a standard Ubuntu laptop with Intel (R) Core (TM) i7-5500 CPU @ 2.40 GHz. After downloading, we pass the models to TMM and tabulate the corresponding peak memory usage in Table 1.

TMM can also produce high-resolution images to visualize the tensor memory requirement of each operator. For example, when we feed the InceptionV1 model that contains 84 graph nodes/operators to TMM, it produces Figure 1. a. (for brevity, out of 84, we show only 0 - 29 operators) along with the detailed CSV report. Similarly, we test TMM on SqueezeNet, MobileNetV1 and show the results in Figure 1. b-c. from which the following can be observed:

- The operators that consume the peak on-device memory during execution can be identified. Here, such operators are circled (M), which are maxPool_2a_3x3 /MaxPool in InceptionV1 (consume 1003.520 KB), fire4/concat in SqueezeNet (consume 6195.200 KB), and in MobileNetV1 it is the conv2d_1_pointwise/Relu6 operator (consume 98.304 KB).
- Most of the operators inside the InceptionV1 model consume high memory (RAM/SRAM) to accommodate *other tensors*, whereas MobileNetV1 operators do not contain *other tensors* at all.
- The three nodes inside SqueezeNet (conv1/Relu, maxpool1/MaxPool, fire4/concat) consume significantly high memory than others. Such nodes can be replaced with cheaper ones that perform the same tasks.

4 CONCLUSION

This paper presented TMM method for accurate calculation and visualization of the tensor memory requirement of each operator in the computation graph of any given neural network model. TMM was tested by using it to analyze the memory consumption of 9 popular models from the domain of image classification, semantic segmentation, pose estimation, and text detection.

The high-accuracy calculations produced by the TMM method can be used by researchers and engineers when developing novel approaches for (i) efficient execution of deeply compressed NNs on IoT devices/products, (ii) memory conservation by loading fewer tensors and tensors re-usage, (iii) finding the cheapest-memory NN graph operators execution sequence for networks with divergent data flow paths.

ACKNOWLEDGEMENTS

Bharath is the corresponding author of this article. This publication has emanated from research supported in part by a grant from Science Foundation Ireland under Grant Number SFI/16/RC/3918 (Confirm), and also by a grant from SFI under Grant Number SFI 12/RC/2289_P2 (Insight). For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission.

REFERENCES

[1] R. David, J. Duke, A. Jain, V. Janapa Reddi, et al. Tensorflow lite micro: embedded machine learning for tinyml systems. In *Proceedings of Machine Learning and Systems*, 2021.

[2] B. Sudharsan, J. G. Breslin, and M. I. Ali. Ml-mcu: a framework to train ml classifiers on mcu-based iot edge devices. *IEEE Internet of Things Journal*, 2021.

[3] T. Chen, T. Moreau, Z. Jiang, et al. Tvm: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[4] C. I. Kanatsoulis and A. Ribeiro. Graph neural networks are more powerful than we think. *arXiv preprint*, 2022.

[5] B. Sudharsan, P. Yadav, J. G. Breslin, and M. I. Ali. An sram optimized approach for constant memory consumption and ultra-fast execution of ml classifiers on tinyml hardware. In *IEEE International Conference on Services Computing (SCC)*, 2021.