MinUn: Accurate ML Inference on Microcontrollers

SHIKHAR JAISWAL, Microsoft Research, India
RAHUL KIRAN KRANTI GOLI, ETH Zurich, Switzerland
AAYAN KUMAR, UC Berkeley, USA
VIVEK SESHADRI, Microsoft Research, India
RAHUL SHARMA, Microsoft Research, India

Running machine learning inference on tiny devices, known as TinyML, is an emerging research area. This task requires generating inference code that uses memory frugally, a task that standard ML frameworks are ill-suited for. A deployment framework for TinyML must be a) parametric in the number representations to take advantage of the emerging representations like posits, b) carefully assign high-precision to a few tensors so that most tensors can be kept in low-precision while still maintaining model accuracy, and c) avoid memory fragmentation. We describe MinUn, the first TinyML framework that holistically addresses these issues and outperforms the prior TinyML frameworks.

CCS Concepts: • **Software and its engineering** → *Source code generation*; **Retargetable compilers**; • **Computer systems organization** → *Embedded hardware.*

Additional Key Words and Phrases: TinyML, Memory Management, Programming Languages, Compilers, Number Representations, Embedded Devices

ACM Reference Format:
Shikhar Jaiswal, Rahul Kiran Kranti Goli, Aayan Kumar, Vivek Seshadri, and Rahul Sharma. 2018. MinUn: Accurate ML Inference on Microcontrollers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 1 (January 2018), 34 pages.

1 INTRODUCTION

Even though memory is one of the most expensive resources, widely-used machine learning (ML) frameworks like TensorFlow and PyTorch assume the availability of plentiful memory during inference. Memory constraints are hard: exceeding the available memory by a single byte will cause a crash. Hence, these frameworks, which rely on interpreters, are unsuitable for running ML on memory-constrained devices. For example, an Arduino Uno [Banzi and Shiloh 2014] has 2KBs of RAM and no ML interpreter can fit in it. In this paper, we explore deployment frameworks for running ML inference on such devices.

Recently, there has been a flurry of work in the TinyML[1] space that aims to run ML on low-power embedded devices [Dennis et al. 2019, 2018; Ghanathe et al. 2021; Gopinath et al. 2019; Gupta et al. 2017; Kumar et al. 2017, 2020; Kusupati et al. 2018; Lin et al. 2021, 2020; Saha et al. 2020; Wang et al. 2019]. As a motivating example consider a low-power chip embedded in the brain that can detect the onset of seizures and warn the user [Kumar et al. 2017]. Animal testing of brain chips is already under way by NeuraLink [Musk 2019]. Such chips need to both perform inference locally on the device (user might be in a place with no internet connectivity) and be low-power (to avoid tissue damage caused by heat dissipation and brain surgeries for battery replacement). Since maintaining large RAMs drains batteries, these tiny devices need to have small RAMs, ranging from a few bytes

[1] https://www.tinyml.org/

Authors' addresses: Shikhar Jaiswal, Microsoft Research, India, t-sjaiswal@microsoft.com; Rahul Kiran Kranti Goli, ETH Zurich, Switzerland, rgoli@ethz.ch; Aayan Kumar, UC Berkeley, USA, aayan@berkeley.edu; Vivek Seshadri, Microsoft Research, India, visesha@microsoft.com; Rahul Sharma, Microsoft Research, India, rahsha@microsoft.com.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

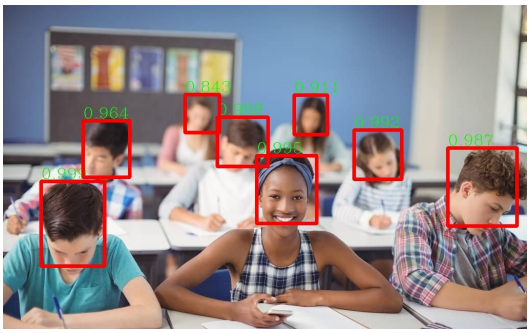

Fig. 1. Face detection example.

to a few kilobytes (KBs). Although, the technical specifications of NeuraLink are unavailable in public domain, for reference, the microcontrollers in pacemakers in hearts have memories between 16 bytes and 10KB [Jekova and Krasteva 2004; Sharma 2014]. Hence, there is a need for deployment frameworks that compile ML models to memory efficient code that can run within the specified RAM limits of the tiny devices.

Running ML inference on tiny devices is non-trivial. For example, Convolutional Neural Networks (CNNs) have millions of parameters, require MBs or GBs of memory, and are unsuitable to be run on tiny devices. Hence, we are interested in recent models targeted for tiny devices that have been specifically designed to provide good accuracy while requiring only thousands of parameters and KBs of memory [Gupta et al. 2017; Kumar et al. 2017; Kusupati et al. 2018; Saha et al. 2020], e.g., Recurrent Neural Networks (RNNs). However, even running these models on tiny devices is challenging. All such ML models use high-precision 32-bit floating-points and we want to squeeze them onto tiny devices by further reducing bitwidths. For example, suppose we want to run *face detection* [He et al. 2019; Yoo et al. 2019; Zhang et al. 2017; Zhao et al. 2019b] on a standard ARM Cortex-M class microcontroller with 256KBs of RAM [ARM 2021]. See **Figure 1** for an example where an ML model has marked the heads in the image. The RNNPool [Saha et al. 2020] model provides state-of-the-art accuracy while consuming only $\sim$ 600KBs of RAM, which is a huge improvement over prior models, for e.g., even size-optimized architectures like MobileNetV2-SSDLite need over 3 MBs of RAM. However, this RNNPool model still exceeds the available memory of the 256 KB device.

An approach that has been well-studied in the literature is to compile or retrain 32-bit floating-point ML models to low bitwidth 8-bit or 2-bit or 1-bit code [Chen et al. 2019; Gong et al. 2019; He and Fan 2019; Hou et al. 2019; Jacob et al. 2018; Krishnamoorthi 2018; Li et al. 2017; Louizos et al. 2019; Martinez et al. 2018; Meller et al. 2019; Nagel et al. 2019; Sakr and Shanbhag 2019; Zhao et al. 2019a; Zhou et al. 2018]. However, the bitwidth required to maintain accuracy depends on the number of model parameters. In particular, it is well-known that models with fewer parameters need larger bitwidths [Fromm et al. 2018] to maintain accuracy. The techniques that use bitwidths ≤ 8 have only been shown to succeed on large models with millions of parameters. For the models of our interest, e.g., the RNNPool-based face detection model, even 8-bits are insufficient to maintain model accuracy (**Section 7.3**).

In the TinyML space, Tensorflow-Lite (TFLite) [Jacob et al. 2018] converts floating-point CNNs to models that uses 8-bits for weights and 32-bits for biases; these models are then run in the TFLite interpreter that has its own memory overheads. SeeDot [Gopinath et al. 2019] is more expressive and compiles arbitrary models to 16-bit fixed-point C++ code that runs on bare metal.

Shiftry [Kumar et al. 2020] goes a step further and compiles to a mixture of 8-bit and 16-bit fixed-point C++ code. Our evaluation shows that these prior frameworks are unsatisfactory when addressing the three primary subproblems of TinyML.

1.1 The Three Subproblems of TinyML

First, we need to use number representations that can both approximate the 32-bit floating-point numbers well with a smaller number of bits, and operations on which are efficiently implementable in hardware. Recently there has been a wave of new number representations [Chen et al. 2017; Gudovskiy and Rigazio 2017; Johnson 2018; Köster et al. 2017; Miyashita et al. 2016; Zhou et al. 2017]: TFLite's *Zero-Skew*, Google's *BFloat16*, NVIDIA's *TensorFloat-32*, Microsoft's *MSFP*, Tesla's *CFP8* and the *posit* [Gustafson and Yonemoto 2017; Gustafson 2017] representation. Posits are attractive as they have better dynamic range and precision compared to floating-point. Prior frameworks for TinyML are tied to specific representations like zero-skew [Jacob et al. 2018] or fixed-point [Gopinath et al. 2019; Kumar et al. 2020]. A robust framework that remains relevant with rapidly evolving number representations must be parametric in the number representation.

Second, for each tensor[2], we need to decide whether to keep it in high precision or in low precision. Keeping all tensors in low precision leads to huge accuracy loss while keeping them all in high precision is wasteful memory-wise (**Section 7**). Hence, for N tensors, we have 2^N choices and need a heuristic to select a good *bitwidth assignment* that minimizes memory, while retaining model accuracy. Crucially, the decision of whether a tensor is assigned low-precision or high-precision must take into account *both* the size of the tensor, and the impact it has on the precision. No prior work provides such a technique to obtain a good bitwidth assignment; we show that baselines that either exclusively use size or exclusively use accuracy to assign the bitwidths are suboptimal and hence we need a novel mechanism that considers both. Note that TFLite and SeeDot use the same bitwidth assignment for all models, whereas, Shiftry exclusively uses accuracy to find a suitable bitwidth assignment.

Third, unlike modern CPUs, tiny devices have no hardware support for virtual memory. Therefore, *fragmentation* can quickly make a program go out of memory. Whether a program can fit in a given memory limit or not is an NP-complete bin packing problem. The solutions provided by prior works for this *memory management* problem, i.e, deciding the physical memory address at which a new tensor is allocated, are unsatisfactory: TFLite asks programmers to manually[3] manage RAM, SeeDot uses wasteful static allocation, and Shiftry uses heuristics to reduce fragmentation that have no optimality guarantees.

1.2 Our Contributions

We provide a framework, MinUn, that makes significant advances on all three subproblems. MinUn is the first TinyML framework which is parametric on the number representation and we evaluate MinUn with both fixed-point and posit representations. We have designed MinUn to have a clear separation between the representation-specific and the representation-independent parts (**Section 5.1**). In contrast, the prior TinyML frameworks are extremely tied to the number representation that they work with, e.g., fixed-point for SeeDot and Shiftry.

MinUn provides the first compiler from high level ML inference code to implementations over posits. Posits [Gustafson and Yonemoto 2017; Gustafson 2017] are rapidly gaining traction as an

[2]In TinyML, it is common for all elements of a tensor to have the same bitwidth to avoid the memory overheads of book-keeping the bitwidths associated with the individual elements.
[3]https://www.tensorflow.org/lite/microcontrollers#limitations

alternative to floating-point and a rich support system for posits is evolving which includes hardware [Podobas and Matsuoka 2018a], software simulators [Leong 2020], debugging tools [Chowdhary et al. 2020], and math libraries [Lim et al. 2021; Lim and Nagarakatte 2021]. We believe that a compiler like MinUn is the obvious next step in this progression and facilitates inference of ML models with posits.

For the bitwidth assignment problem, we propose a novel exploration algorithm, Haunter, that assigns a *promotability* score (**Section 5.1**) to each tensor. Tensors with fewer elements get a higher score and tensors with low impact on precision get a lower score. Initially, all tensors are kept in low-precision and tensors with high scores are promoted (assigned a higher bitwidth) with priority to high precision while ensuring that the memory constraints are met. This achieves two broad objectives: smaller tensors with considerable impact on precision are prioritized for early promotion, and larger tensors with low impact on precision are prioritized for late promotion. This heuristic produces better assignments than the accuracy-based heuristic of Shiftry, while being much more efficient and scalable (**Section 5.2**).

Finally, for RAM management, MinUn encodes the memory management problem to a bin-packing problem and solves it using Knuth's Algorithm X [Knuth 2000], which is guaranteed to return the optimum result albeit in exponential time. Here, our main contribution is to come up with an effective encoding and to adapt the general framework of Algorithm X to ensure a tractable runtime in practice. Our evaluation shows that this implementation terminates in reasonable time on our benchmarks and reduces RAM consumption by 0% to 24% over a greedy heuristic. This result has ramifications beyond machine learning and leads us to believe that optimum memory allocation can help address heuristic suboptimal memory management in the space of embedded devices which is currently performed via wasteful static allocation, fragmentation-prone dynamic allocation, memory pools with restricted applicability, and so on. MinUn is the *first* framework to provide optimum memory management: if there is a mapping from variables to physical addresses that keeps fragmentation low enough for a model to run in the given RAM limit then MinUn is guaranteed to find it. The prior memory management heuristics fail to provide this guarantee.

We evaluate MinUn on models that have been designed by ML researchers to be deployed on tiny devices and have been considered by prior TinyML work [Ghanathe et al. 2021; Gopinath et al. 2019; Kumar et al. 2020]. These include 1) ProtoNN [Gupta et al. 2017] (a variant of k-nearest neighbors), 2) Bonsai [Kumar et al. 2017] (a variant of decision trees), 3) FastGRNN [Kusupati et al. 2018] (a variant of recurrent neural networks (RNNs)) operating on speech and time-series data from sensors, and 4) RNNPool [Saha et al. 2020], which combines CNNs and RNNs to achieve state-of-the-art accuracy on computer vision tasks while keeping the memory consumption low. The RAM consumption of these models with MinUn varies between 20 bytes and 226 kilobytes, depending on the complexity of the model. In particular, for the model in **Figure 1**, MinUn reduces the RAM consumption from $\sim$ 600KBs to $\sim$ 180KBs, thus enabling this model to run on devices with 256KBs of RAM. Although MinUn was not designed for large models, even on an ImageNet-scale CNN [Iandola et al. 2016], MinUn outperforms the prior TinyML frameworks (**Section 7.6**).

The believe that MinUn has an important role in the repertoire of techniques to reduce memory usage of ML models. We anticipate that the ML researchers will apply various techniques like structural pruning, low-rank approximation of weight matrices, novel model architectures, etc., to create models with fewer parameters and MinUn will further reduce the memory usage by keeping most of these parameters in low bitwidths. For example, the models used in our evaluation are based on novel ML architectures that keep the number of model parameters small, and MinUn further reduces the peak RAM usage by keeping most of the model parameters in 8-bits. Hence, the "post training quantization" of MinUn is complementary to the quantization techniques used during training.

1.3 Organization

The rest of the paper is organized as follows. **Section 2** discusses the requisite background on different number representations, ML and Computer Vision terminology and microcontrollers. **Section 3** provides a working example for our compiler. After providing a high level overview of MinUn in **Section 4**, we provide the method and analysis for Haunter in **Section 5.1** and **Section 5.2** respectively, and the encoding and adaptation of Algorithm X to the *memory management* problem in **Section 5.3**. **Section 6** discusses the implementation details of MinUn and **Section 7** provides the evaluation to show that MinUn outperforms the state-of-the-art. Finally, **Section 8** discusses related work and **Section 9** concludes the paper.

2 PRELIMINARIES

In this section, we discuss the terminology from fixed-point representation, posit representation, machine learning and computer vision problems, and microcontrollers.

2.1 Fixed-Point Representation

In standard fixed-point arithmetic, a real number r is stored as the b-bit signed integer $\lfloor r \times 2^s \rfloor_b$, where s is a predetermined integer called the *scale* and b is termed as the *bitwidth* of the representation. As an example, consider the real number $r = 1.6181$:

$$r = 1.6181 = 1.6181 \times \frac{2^{14}}{2^{14}} \approx \frac{\lfloor 1.6181 \times 2^{14} \rfloor}{2^{14}} = 26510 \times 2^{-14}$$

To represent 1.6181 in fixed-point arithmetic, we store the 16-bit integer 26510, associated with the scale of 14. The interpreted real value would be $26510 \times 2^{-14} \approx 1.61804$, which is a close approximation of the original value.

For any real number and a given bitwidth, there is an optimum scale for the fixed-point representation; in the example above, given a 16-bit integer representation, 14 is the optimum scale for 1.6181, as using a scale of 15 or above causes integer overflow, and using a scale of 13 or below leads to more imprecise results.

A generalization of the above approach is the "zero-skew" quantization scheme by TFLite [Jacob et al. 2018]. Here, a real number r is stored in the following manner:

$$r = S(q_b - Z) \tag{1}$$

where S is an arbitrary positive real number, termed as the skew, and Z is the quantized value of 0, termed as the zero-point, expressed in the same type as the b-bit unsigned integer q_b. Let us assume that we wish to represent a uniformly sampled real from the range $[-2, 2]$ in 8-bits. Hence, by choosing $r = 1.6181$, $S = \frac{\text{Variable Range}}{\text{uint8 Range}} = \frac{4}{255} \approx 0.015686$ and $Z = -\frac{\text{Variable Range Infimum}}{S} \approx 128$, we obtain q as:

$$q_8 = \lfloor \frac{1.6181}{0.015686} \rfloor + 128 \approx 231 \tag{2}$$

Substituting these approximations back into equation (1) gives us the interpreted value 1.6157.

The skew and the zero-point are colloquially referred to as quantization parameters. All values in a tensor use a unique set of quantization parameters. A higher bitwidth preserves more precision, as long as the underlying integer does not overflow due to a high scale.

2.2 Posit Representation

Posit [Gustafson and Yonemoto 2017; Gustafson 2017] is a novel real-number representation that is designed as an alternative to the IEEE-754 floating-point [IEEE 2019].

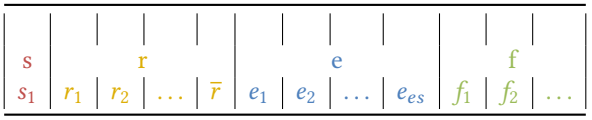

Fig. 2. Example posit format.

While an IEEE-754 float consists of a sign bit and a fixed number of bits for mantissa and exponent, the posit representation consists of 4 sections - the sign-bit (s), the regime bits (r), the exponent bits (e) and the fraction bits (f) - where the last three sections have a dynamic number of bits assigned to them, out of the overall fixed number of bits per posit.

The sign bit denotes the sign of the number, where a 0 denotes a positive number and a 1 denotes a negative number. The regime bits, which signify a super-exponent, define a regime value k, which can be positive, negative or zero:

- Positive regime values are represented by a stream of 1s delimited by a single 0. The value of the regime would be one less than the number of leading 1s. For example, if the regime bits are 1110, then $k = 2$; 11111 in a posit with only 6 bits would correspond to $k = 4$.
- Negative regimes are represented by a stream of 0s delimited by a single 1. The value of the regime in this would be $k = -1 \times p$ where p is the number of leading 0s. For example, if the regime bits are 00001, then $k = -4$; 000000 in a posit with only 7 bits would correspond to $k = -6$.
- The value zero is represented by the regime being 10.

The length of the regime is dynamic, and is determined by the number of bits observed till a delimiting bit is encountered. Hence for an n-bit posit, there can be a minimum of 2 and a maximum of $n - 1$ regime bits, (all bits excluding the sign bit). Therefore, the value of k can range from $-(n-1)$ to $(n-2)$ in an n-bit number. For a given bitwidth, the es value signifies the maximum number of bits available for the exponent section. Once the sign, regime and exponent bits have been considered, only then the remaining bits are interpreted as fraction bits.

To find the value of a number in posit representation, we define a parameter called $useed$ ($= 2^{2^{es}}$). The value of a posit with sign-bit s is

$$(-1)^s \times (useed)^k \times 2^E \times 1.F$$

where E is the value of the exponent represented by the exponent bits (e), k is the value denoted by the regime bits r, and F is the fraction represented by the fraction bits f. It is possible in the posit representation for a number to contain no exponent bits and/or no fraction bits. In those cases the corresponding values of exponent (E) and fraction (F) are 0.

For example, assuming $es = 2$ and a bitwidth of 8 bits, a bit string 01101101 will be interpreted as:

- Sign bit ($s = 0$).
- Regime bits ($r = \{110\}$). Hence $k = 1$.
- Exponent bits ($e = \{11\}$). Hence $E = 2^3 = 8$.
- Fraction bits ($f = \{01\}$). Hence $1.F = 1.25$.

Since $useed = 2^{2^{es}} = 16$, thus the real number represented by our bit string is $1 \times 16^1 \times 8 \times 1.25 = 160$.

2.3 ML and Computer Vision Preliminaries

In this paper, we focus on two broad paradigms of ML problem space: classification and localization (commonly known as detection).

Classification-based models aim to predict singular discrete labels. For example, one can design a classifier which takes as input an image, and predicts a label indicating whether a human is present in the picture or not. The performance of such a model can be objectively measured using classification accuracy, which is the percentage of times the label is correct.

Localization-based models aim to predict more continuous outputs which determine the location of an instance. For example, given the image of a human being, one might need to employ a detection model in order to obtain the bounding-box coordinates of the human being in the picture frame. However, this makes the performance of the model continuous in terms of output quality. We use *Mean Average Precision (MAP)* scores in such a scenario. MAP measures the mean area under the precision-recall curves, and is a standard metric for localization problems like face detection.

ML implementations typically choose the floating-point representation. However, different number representations offer salient advantages, in terms of computation, memory footprint and performance. As such, the effectiveness of a model expressed in a specific number representation is judged by how well it performs compared to its floating-point counterpart.

2.4 Microcontrollers

Microcontrollers, designed for embedded applications, are a class of small, low-power computers on a single integrated circuit (IC). They comprise one or more CPU cores, with some memory and I/O peripherals. Two different kinds of memory are commonly used with microcontrollers, a reprogrammable *read-only* memory for storing firmware called Flash (similar to ROM, EPROM or EEPROM for personal computers) and a read-write memory called RAM (employing SRAM technology). Microcontrollers allow the storage of read-only bits such as model weights and the compiled software binaries on the Flash, and use their RAM for storing the mutable temporary variables corresponding to the results of intermediate computations. On these devices, RAM is a much more scarce resource than the Flash memory, with the latter often being user-expandable.

3 WORKING EXAMPLE

To motivate the problem addressed in this paper, we provide an exemplar linear classifier model and generate the posit code for it. Consider **Procedure 1**.

Let us set a constraint for running this program in 3 bytes of peak RAM consumption. Here, we assume $W_1 \in \mathbb{R}^{1\times2}$, $X_1 \in \mathbb{R}^{2\times1}$ and $B_1 \in \mathbb{R}$ are read-only tensors. Hence, they are stored and fetched directly from the Flash, and do not count towards RAM usage. Additionally, the only two bitwidth options available to us are 8 bits and 16 bits. The corresponding 16-bit *homogeneous* (all tensors assigned the same bitwidth) posit code is depicted in **Procedure 2**. Here, only the temporary tensors t_1 and t_2 are stored in RAM.

We observe that the homogeneous 16-bit program uses 4 bytes of RAM (because of t_1 and t_2 being 2 bytes each), making it unsuitable for our target platform. While homogeneous 8-bit code would only take 2 bytes of RAM, and satisfy our memory constraint, it might lead to excessive losses in accuracy. Hence, we make use of *heterogeneous* bitwidth configurations for our tensors, which use 8-bit for some tensors and 16-bit for others, to reduce RAM consumption, while aiming to minimize the accuracy loss.

Haunter defines a promotability score for each tensor (**Table 1** and **Section 5.1.2**). Ranking tensors in the decreasing order of promotability scores gives us an order list of promotion priority for the different tensors in the program.

Based on this promotion order, Haunter considers a few broad cases suitable for exploration, while starting from different initial configurations, as depicted in **Table 2**, and explained in **Section 5.1.3**. At each step of an exploration instance, Haunter promotes a tensor cumulatively (i.e. in a successive operation on top of the previous bitwidth configuration), while strictly maintaining

Procedure 1: Linear Classifier

$W_1 := \begin{pmatrix} -2.139562 & 1.885351 \end{pmatrix}$

$X_1 := \begin{pmatrix} 1.185109 \\ -2.206466 \end{pmatrix}$

$B_1 := \begin{pmatrix} 0.146048 \end{pmatrix}$

return $(W_1 \times X_1) + B_1$

Table 1. Variable sizes, values and promotability scores.

Var	#	16-Bit Values	8-Bit Values	Promotability
W_1	2	(-2.13965, 1.88525)	(-2.25, 1.875)	$\frac{0.01025}{2} \approx 0.00513$
B_1	1	0.146057	0.140625	$\frac{0.00543}{1} \approx 0.00543$
X_1	2	(1.18506, -2.20605)	(1.125, -2.25)	$\frac{0.04395}{2} \approx 0.02198$
t_1	1	-6.69531	-7	$\frac{0.30469}{1} \approx 0.30469$
t_2	1	-6.54883	-7	$\frac{0.45117}{1} \approx 0.45117$

Table 2. Cumulative promotion within memory limit with different initial states chosen by Haunter. Repeated instances have been omitted for brevity.

Promoted Vars	RAM Usage	Error
None	2	0.45047
t_2	3	
t_2, t_1	4	
t_2, X_1	3	
t_2, X_1, B_1	3	
t_2, X_1, B_1, W_1	3	0.19601
t_1	3	
t_1, t_2	4	
t_1, X_1	3	
t_1, X_1, B_1	3	
t_1, X_1, B_1, W_1	3	0.04953
t_2, t_1, X_1, B_1, W_1	4	
t_1, X_1, B_1, W_1	3	0.04953
⋮		

Procedure 2: 16-bit Homogeneous Posit (*es*=2) Code

$\texttt{posit}_{16}[1][2]W_1 := \begin{pmatrix} -2.13965_{16} & 1.88525_{16} \end{pmatrix}$

$\texttt{posit}_{16}[2][1]X_1 := \begin{pmatrix} 1.18506_{16} \\ -2.20605_{16} \end{pmatrix}$

$\texttt{posit}_{16}[1][1]B_1 := \begin{pmatrix} 0.146057_{16} \end{pmatrix}$

$\texttt{posit}_{16}\ t_1, t_2;$

$t_1 = (W_1 \times_{\texttt{quire}_{128}} X_1)$

$t_2 = t_1 + B_1$

return t_2

Procedure 3: Heterogeneous Posit (*es*=2) Code; t_2 uses 8-bits, rest use 16-bits

$\texttt{posit}_{16}[1][2]W_1 := \begin{pmatrix} -2.13965_{16} & 1.88525_{16} \end{pmatrix}$

$\texttt{posit}_{16}[2][1]X_1 := \begin{pmatrix} 1.18506_{16} \\ -2.20605_{16} \end{pmatrix}$

$\texttt{posit}_{16}[1][1]B_1 := \begin{pmatrix} 0.146057_{16} \end{pmatrix}$

$\texttt{char}_8[3]\ scratch;$

$scratch[0:1] = \texttt{posit}_{16}\ (W_1 \times_{\texttt{quire}_{128}} X_1)$

$scratch[2] = \texttt{posit}_8\ (scratch[0:1] + B_1)$

return $scratch[2]$

the constraint on the RAM consumption, and reverting the promotion in cases where the constraint is violated. At the end of each exploration instance, Haunter executes code on the derived bitwidth configuration, and records the accuracy.

Taking the example of the second instance from **Table 2**, we initiate an exploration with all the tensors being assigned a low bitwidth. Then, based on the promotion order (i.e. $[t_2, t_1, X_1, B_1, W_1]$), we promote t_2 and generate the code on the new bitwidth configuration in order to estimate its RAM usage. We continue promoting other tensors in the promotion order, while keeping the previous tensors in the promoted state, till we end up promoting a tensor which leads to an overshoot of the memory limit (t_1). At this stage, we simply revert the promotion of the "overshooting" tensor, and continue to consider other tensors, till we have exhausted our promotion order. Once we have considered all tensors, we simply execute the generated code and record the accuracy.

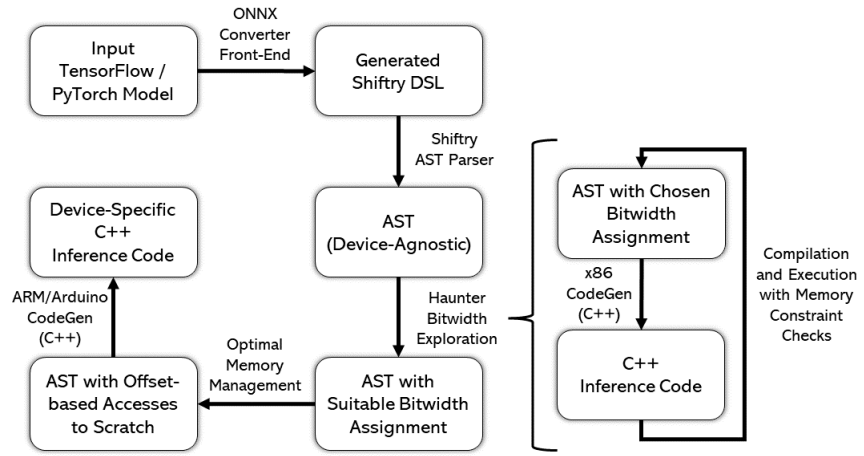

Fig. 3. MinUn Overview.

Finally, Haunter chooses the configuration incurring the minimum loss in accuracy, and generates the final code based on this bitwidth configuration which keeps t_2 as an 8-bit posit and the rest remain as 16-bit posits. During memory management, the RAM tensors are mapped to indices into a global array called *scratch* (**Procedure 3**) that resides in the RAM. This reduces the runtime overhead of allocating/deallocating tensors during program execution to zero. The RAM usage of the program at runtime is upper-bounded by the size of *scratch*. Here, t_1 occupies the first two bytes, t_2 occupies the third byte of *scratch*, and the RAM consumption of **Procedure 3** remains below 3-bytes, as desired.

4 COMPILER OVERVIEW

In this section, we provide a high-level overview of the MinUn compiler (**Figure 3**). MinUn expects as input a program written in the Shiftry-DSL [Kumar et al. 2020], a concise syntax for expressing operations and functions commonly used in ML models. The DSL code for models can either be written manually, or if the models are in TensorFlow/PyTorch then it can be obtained by exporting the models to ONNX and using MinUn's ONNX front-end (**Section 6**). MinUn then parses the input program from the DSL and generates an abstract syntax tree (AST) required for Haunter, our bitwidth exploration strategy. Haunter then returns a suitable bitwidth assignment for all the tensors in the program. This is done roughly by choosing an initial bitwidth assignment, generating the corresponding x86 C++ inference code for it, compiling and executing the code for logging accuracy and memory usage information, and using the logged information for making changes to the bitwidth assignment. Haunter ensures that the user-provided memory constraints are always met during bitwidth exploration, and returns a suitable assignment which maximises accuracy.

Then through our optimum memory management strategy, we obtain a memory utilization map from RAM tensors to scratch indices, corresponding to the given bitwidth configuration. Recall, that the aim of MinUn is to minimize the total size of the scratch while ensuring that the program executes correctly.

The final platform-specific code is generated using a C++ codegen pass over the AST, which also replaces all the intermediate RAM tensors, with offsets-based accesses to the scratch on the basis of our memory utilization map. The final output code is then cross-compiled and executed on the

microcontroller of choice. Further, we exposit the process of bitwidth exploration, and analyse the benefits of following our methodology.

5 TECHNICAL DETAILS

In this section, we explain Haunter, analyze its complexity, and describe our optimum memory management mechanism.

5.1 Bitwidth Assignment with Haunter

The Haunter algorithm works in a three-stage process, with the first stage being dependent on the number representation being considered, and the remaining two stages staying agnostic to the representation. Haunter uses three pieces of user-provided information - the amount of available RAM, a pair of bitwidths (*highBitwidth* and *lowBitwidth*) between which the exploration is to be commenced, and a soft limit factor that acts as a tuning parameter. The soft limit enables Haunter to explore bitwidth configurations that might potentially overshoot the given memory limit when used with the standard first-fit heuristic for memory management [Nutt 2002]. Additionally, it can also be used to make the exploration process more frugal, by choosing a soft limit < 1. We use ρ to denote the bitwidth assignment, that maps each tensor to two possible bitwidths {*lowBitwidth*, *highBitwidth*}. Note that Haunter generalizes in a straightforward manner to k possible bitwidths but we have not found this generalization to be useful in practice (**Section 7.7.2**).

5.1.1 Preprocessing (Stage I): This stage finds the values of data-dependent parameters that capture the runtime ranges of inputs and intermediate tensors. Examples of such parameters include the most suitable value of *es* for 8-bit and 16-bit posits, and scales for individual tensors for fixed-point. **Algorithm 4** succinctly describes the driver method used for obtaining these parameters for posits, where we simply pick an appropriate *es* value based on accuracy obtained with homogeneous bitwidth configurations. To determine scales of fixed-point, we use Shiftry's data-driven scale assignment procedure [Kumar et al. 2020], that profiles each tensor in the floating-point version of the code by running it on a given set of inputs. This profiling information includes the minimum and maximum values observed for each tensor, and is used for assigning a suitable scale to the tensor (**Section 2.1**).

5.1.2 Heat-Map Generation (Stage II): During inference on a particular data point, a "value map" stores key-value pairs, where the keys are the tensor names, and the values are the floating point values held by that tensor during execution on the provided data point. Haunter starts from an initial configuration where all the tensors are kept in *lowBitwidth*. In the first phase (**Algorithm 5**), we set each promotable tensor to *highBitwidth* simultaneously, and then compile and execute the code to obtain a "high-bitwidth value map" (lines 1-4). We repeat the same procedure once more, by simultaneously setting each tensor to *lowBitwidth*, compiling and executing to obtain a similar "low-bitwidth value map" (lines 5-8).

Based on these two maps, we attempt to create a "heat map" in the second phase (**Algorithm 6**), wherein, we simply calculate the absolute deviation in the representative floating-point values between the *highBitwidth* and *lowBitwidth* case for each tensor. Taking the example of tensor X_1 from **Procedure 1**, we calculate the representative floating-point values held in the tensor assuming both 16-bit and 8-bit posit structure (lines 3-4), and then calculate element-wise absolute deviations between the two tensors (lines 6-8). Since keeping track of large tensors is infeasible, we simply sort the element-wise deviations in increasing order and take the 95th percentile deviation

Algorithm 4: Determining *es* through accuracy.

```
Function Preprocess(lowBitwidth, highBitwidth):
    if lowBitwidth == 8 then
        es8 = 2
        ρ ← {var ↦ 8}∀var ∈ allVars
        accuracyES0 ← Compile&Execute(es = 0, ρ) # Generate, compile and execute model code
        accuracyES2 ← Compile&Execute(es = 2, ρ) # with the given set of parameters.
        if accuracyES0 > accuracyES2 then
            es8 = 0
    if highBitwidth == 16 then
        es16 = 2
        ρ ← {var ↦ 16}∀var ∈ allVars
        accuracyES1 ← Compile&Execute(es = 1, ρ)
        accuracyES2 ← Compile&Execute(es = 2, ρ)
        if accuracyES1 > accuracyES2 then
            es16 = 1
    return es8, es16
```

as the representative error deviation for each tensor (line 9). We define the promotability score as:

$$\text{Promotability} = \frac{\text{95th Percentile Error Deviation}}{\text{Tensor Cardinality}}$$

Here, tensor cardinality refers to the size of the tensor or the number of elements in a tensor. Sorting the tensors in decreasing order of the promotability scores provides an ordered ranking called *promotionOrder*, which prioritizes smaller-sized tensors with large error deviations to be promoted first (line 10). One could advocate the use of accuracy values as the metric to be used in place of error deviation for promotability scores. While it is a more direct metric for promoting tensors, this strategy, like Shiftry's bitwidth exploration, suffers from higher computational complexity (**Section 5.2**).

Algorithm 5: Generating value maps for each bitwidth.

```
Function CreateValueMaps(lowBitwidth, highBitwidth):
    ρ ← {var ↦ highBitwidth}∀var ∈ allVars
    ClearLogs() # Clear logger file used for storing tensor values.
    accHigh ← Compile&Execute(ρ)
    varsValueMapHigh ← Log&SaveValues() # Create value map from logged data.
    ρ ← {var ↦ lowBitwidth}∀var ∈ allVars
    ClearLogs()
    accLow ← Compile&Execute(ρ)
    varsValueMapLow ← Log&SaveValues()
    SaveAcc&Bitwidth(accLow, ρ) # Save accuracy and bitwidth configuration information.
    return varsValueMapLow, varsValueMapHigh
```

Algorithm 6: Generating heat map, given the value maps, and promotion order list.

```
Function CreateHeatMap(varsValueMapLow, varsValueMapHigh):
1    heatMap = {}
2    for var ∈ allVars do
3        lowValues ← varsValueMapLow[var]
4        highValues ← varsValueMapHigh[var]
5        errors ← []
6        for lowValue, highValue ∈ {lowValues, highValues} do
7            absError = |highValue − lowValue|
8            error.append(absError)
9        heatMap[var] ← Get95thPercentileValue(errors) ÷ varsToSizeMap[var]
10   promotionOrder ← DecreasingArgSort(heatMap).keys()
11   return heatMap, promotionOrder
```

5.1.3 Promotion Algorithm (Stage III): The algorithm itself runs in three stages as described in **Algorithm 8**. The first stage (lines 1-4) aims to cumulatively promote all the tensors while staying within the *memoryLimit*, and also determine which tensors can potentially overshoot the provided limit. The second stage (lines 6-14) individually considers these "overshooting" tensors, by promoting them first, and then commencing the *promotionOrder*-based exploration. The final stage (lines 16-21) commences the exploration after simultaneously promoting all of these "overshooting" tensors. The final bitwidth configuration is chosen as the one minimizing the disagreement counts against the floating point code outputs, and is returned as the output of the bitwidth exploration.

Algorithm 7: Helper function of **Algorithm 8**; promoting all variables to high bitwidth subject to memory limit and soft limit factor.

```
Function PromoteWithinMemoryLimit(promotionOrder, memoryLimit, softLimitFactor):
1    overshootingVars ← []
2    for var ∈ promotionOrder do
3        ρ ← {var ↦ highBitwidth}
4        memoryUsage ← Compile(ρ) # Generate and compile model code for the given configuration.
5        if memoryUsage > memoryLimit × softLimitFactor then
6            ρ ← {var ↦ lowBitwidth}
7        if memoryUsage ≥ memoryLimit × softLimitFactor then
8            overshootingVars.append(var)
9    return overshootingVars
```

5.2 Complexity Analysis

Bitwidth exploration, via HAUNTER, is the most computationally expensive block of MINUN, owing to repeated code compilation for varying bitwidth assignments, and execution calls for measuring the accuracy. While the former depends on the the size of the program text of the model, the latter is determined by both the runtime of the model and the size of the dataset being considered.

We denote the amount of time required per compilation call as $\mathcal{T}_{compilation}$, and treat it as the upper bound of the compilation time required for different bitwidth configurations. We follow the same terminology for execution call latency (per example) as well, terming it $\mathcal{T}_{execution}$.

Algorithm 8: Haunter Promotion Algorithm

```
Function PromotionAlgorithm(promotionOrder, memoryLimit, softLimitFactor):
    ρ ← {var ↦ lowBitwidth}∀var ∈ allVars
    overshootingVars ←
     PromoteWithinMemoryLimit(promotionOrder, memoryLimit, softLimitFactor)
    accuracy ← Compile&Execute(ρ)
    SaveAcc&Bitwidth(accuracy, ρ)

    for promotable ∈ overshootingVars do
        ρ ← {var ↦ lowBitwidth}∀var ∈ allVars
        ρ ← {promotable ↦ highBitwidth}
        memoryUsage ← GetMemoryUsage(ρ)
        if memoryUsage > memoryLimit × softLimitFactor then
            continue
        PromoteWithinMemoryLimit(promotionOrder, memoryLimit, softLimitFactor)
        accuracy ← Compile&Execute(ρ)
        SaveAcc&Bitwidth(accuracy, ρ)

    ρ ← {var ↦ lowBitwidth}∀var ∈ allVars
    ρ ← {var ↦ highBitwidth}∀var ∈ overshootingVars
    PromoteWithinMemoryLimit(promotionOrder, memoryLimit, softLimitFactor)
    accuracy ← Compile&Execute(ρ)
    SaveAcc&Bitwidth(accuracy, ρ)
    ρ ← FindBitwidthConfigWithBestAccuracy()
    return ρ
```

The bitwidth assignment mechanism of the prior work of Shiftry follows an exploration phase where code compilation and execution occurs twice per each tensor, once during individual tensor demotion, and once during cumulative tensor demotion. Hence for a model with N tensors and supplied with a dataset of D samples, Shiftry incurs a cost of $O(N)$ compilation and $O(N)$ execution calls, resulting in a bitwidth exploration latency of:

$$O(N \times \mathcal{T}_{compilation} + N \times D \times \mathcal{T}_{execution})$$

For large models, supplied with large datasets, the second term is dominant, since $\mathcal{T}_{execution}$ becomes more expensive than $\mathcal{T}_{compilation}$. Haunter focuses on reducing the number of execution calls on average to $O(\log N)$, thereby significantly reducing exploration latency.

The core observation that guides our average case analysis is that most ML models portray an asymmetric diversity in the tensor sizes. While classification or localization-based models try to "downsize" the input into smaller activations (by some multiplicative factor) at each step, generative models aim to "upsize" the input into larger activations. Empirically, for our benchmarks, the tensor sizes appear to follow an exponential distribution. With that assumption, a constant parameter $\alpha > 1$, and $memoryLimit = C$, for some positive scalar $C = \Theta(N^{\alpha}) << \alpha^{N}$, we observe that the maximum number of tensors being, safely and cumulatively, promotable to a higher bitwidth is:

$$\lfloor \log (memoryLimit) \rfloor = \lfloor \log C \rfloor = O(\log N)$$

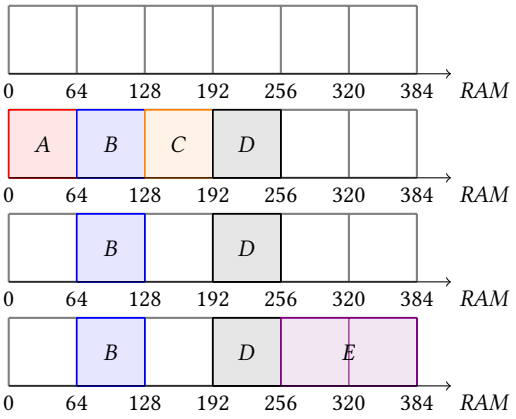

Fig. 4. Fragmentation example: 1) Start with an empty RAM. 2) Allocate tensors A, B, C, and D, each of 64-bytes. 3) Later, deallocate A and C, freeing 128 bytes. 4) Then allocate E of size 128 bytes. Fragmented RAM Usage: 384 bytes, Optimal Peak RAM Usage: 256 bytes.

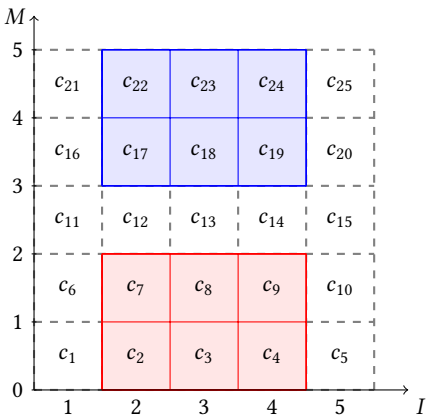

Fig. 5. Example canvas with potential memory maps (in color) for a 2-byte tensor active between instructions 2 and 4. Note that each such map would form a unique element in C_{v_i} for a given v_i.

Hence, irrespective of the promotion order and the size of the *overshootingVars* set in **Algorithm 8**, only $O(\log N)$ number of variables are able to proceed beyond line 10, leading to an average case of $O(N \log N)$ compilation calls and $O(\log N)$ execution calls in total. The total bitwidth exploration latency becomes:

$$O(N \times \log N \times \mathcal{T}_{compilation} + \log N \times D \times \mathcal{T}_{execution})$$

This reduction in execution calls becomes significant even for moderately-sized models such as the Face-B model, described in **Section 7.2**, having 135 tensors, running on a dataset of 405 QVGA images. In our experiments, we observed the exploration time get cut to about half, from 46 minutes using SHIFTRY to 25 minutes using MINUN.

5.3 Optimum Memory Management

Once the bitwidth assignment has been computed, MINUN must allocate tensors at physical memory addresses to keep the peak RAM consumption low by maximizing the reuse of RAM locations. There are several ways to do this memory management.

Static Assignment: Tensors are pushed onto the program stack when they come in scope and popped out when they go out of scope in Last-In-First-Out (LIFO) order. This approach results in high RAM consumption as a tensor X cannot be deallocated before tensors instantiated later are deallocated, even when X is no longer live.

Dynamic Assignment: Tensors are allocated memory on the program heap, and freed when no longer needed. This approach delegates the memory management to the runtime, which may not be preferable for resource-constrained devices owing to their runtime overheads. This approach is known to be prone to fragmentation [Kumar et al. 2020], as depicted in **Figure 4**. Here, deallocating variables A and C has fragmented the memory.

Delegating Memory Management to the Compiler: Given an ML model, tensor sizes and live ranges (range of instructions where a tensor is used and needs memory) are known at compile time. A mapping from tensors to memory locations (or indices) on a contiguous array (scratch) can be

computed during the compilation process itself. This reuses the memory for tensors that are no longer alive to allocate new tensors. For example, in **Figure 4**, one way to avoid fragmentation is for the compiler to map A and C to memory locations that are adjacent to each other. Computing such maps is an NP-Hard bin packing problem which can be addressed through suboptimal greedy heuristics. Although this problem has the flavor of *register allocation* [Aho et al. 2006], note that registers don't suffer from fragmentation, which is the primary problem here.

We take the third approach, but unlike prior work [Kumar et al. 2020] we do not use greedy heuristics. Instead, we approach the bin-packing problem as an exact cover problem.

Definition 1 (Exact Cover): Let $\mathcal{P}(X)$ denote the power set of a set X. Given a set X and a set $S \subset \mathcal{P}(X)$, an exact cover of X is a set $S^* \subset S$ such that:

- $s_i \cap s_j = \emptyset, \forall s_i, s_j \in S^*, i \neq j$
- $\bigcup_{s_i \in S^*} s_i = X$

Consider a 2-D canvas of size $M \times I$ (where M is the memory budget in bytes along y-axis and I is the number of instructions in the program along x-axis). The smallest individual element c_i of the canvas is a square of unit area, semantically denoting a byte-sized tensor which is active only for a single instruction. For a tensor v_i active between instructions i_{start} and i_{end}, and occupying b_i bytes in memory, we can map a rectangle made of $b_i \times (i_{end} - i_{start})$ contiguous units on this canvas, occurring between i_{start} and i_{start}.

Let $C = \{c_1, c_2, \ldots, c_{M \times I}\}$ denote the set of all unit elements of the canvas, and $V = \{v_1, v_2, \ldots, v_N\}$ denote the set of all tensors in the program. Additionally, for each tensor v_i, we define a new set $C_{v_i} \subset \mathcal{P}(C)$ containing all subsets of unit elements in C, which can be potential memory maps to the tensor. Hence, each individual set in C_{v_i} forms a contiguous block of memory spanning the live range of v_i, and has a height of at least b_i units, as depicted in **Figure 5**.

Let $X = C \cup V$ and $S \subset \mathcal{P}(X)$ with the following constraint:

- $\forall s_k \in S, v_i \in V [v_i \in s_k \implies s_k \setminus v_i \in C_{v_i}]$

Given these sets, we solve the exact cover problem using an exponential-time, backtracking search algorithm called Algorithm X [Knuth 2000], with the following optimizations to speed up the exploration:

- We use the Dancing Links Algorithm as proposed in [Knuth 2000] to efficiently assign and unassign a tensor rectangle to a particular memory location.
- We visit the tensors in decreasing order of the area of the rectangles, as a heuristic to quickly prune out assignments causing conflicts between tensors.
- To reduce the size of the problem, we make a coarsening assumption: all tensors are assumed to have a size as a multiple of some coarsening constant k. For example, if a program has 3 tensors of size 32, 56 and 64 bytes, then we can set $k = 32$, rounding up the sizes of the tensors as needed. Once all tensor sizes are a multiple of k, we can effectively create a canvas of size $\frac{M}{k} \times I$ instead of $M \times I$ by dividing all sizes by k, which reduces the problem size. Note that choosing $k = 1$ recovers the original formulation of the algorithm, and will always find the optimal solution, if it exists.

Our optimum memory management mechanism aims to find an exact cover for the minimum viable value of M. At each instruction, we can find minimum memory budget required for execution by tightly stacking the alive tensors at that instruction, with no gaps and no overlaps along the y-axis, and looking at height of the stack. Calculating this height for all I instructions, and picking the highest value out of them, gives us this minimum budget.

Once an exact cover is found for the minimum M, we can map a tensor from the canvas to the scratch, by taking its minimum y-coordinate, and using that as an offset from the starting location

of the scratch. Notice that since M denotes the minimum viable height of the canvas, it also denotes the total size of the scratch. Since exact cover finds a suitable allocation within the minimum viable budget, it effectively resolves the fragmentation problem.

The proof regarding the optimality of our memory management technique reduces to the proof that Algorithm X will always find an optimum assignment if it exists, which is well-known. If $P \neq NP$, then all polynomial-time memory management algorithms are suboptimal. Since all prior works on memory management use polynomial-time heuristics, they are suboptimal.

6 IMPLEMENTATION DETAILS

Our framework is built on top of Shiftry, and has been implemented in 19K lines of Python code and 12K lines of C/C++ code. Additionally, we make use of the following set of libraries in our experiments in order to obtain arithmetic routines for different number representations:

- BFloat [Abadi et al. 2015] is a standalone C++ header-only sub-library implemented in the TensorFlow library, allowing conversion between Float32 and BFloat16 types.
- SoftPosit [Leong 2020] provides posit arithmetic routines for simulating on x86-based platforms using C++. The library contains all functions expected in the Posit Standard [Group 2018]. For 8-bit posits, $es = 0$ and $es = 2$ support, and for 16-bit posits, $es = 1$ and $es = 2$ support is available. For 9-bit, 10-bit, 12-bit and 32-bit posits, only $es = 2$ support is available at the time of writing.
- Gemmlowp [Jacob et al. 2015] is a matrix multiplication library, designed for use with "low precision" 8-bit fixed-point types. It is the library used by TFLite's zero-skew number representation, written in C++.

For the evaluation on large models (**Section 7.6**), we downloaded the pre-trained ONNX model from the public ONNX Model Zoo [Team 2021], and implemented a Python front-end for automatically compiling ONNX models to the DSL (**Section 4**) programs that MinUn takes as input. In cases where certain arithmetic functions are not available for a given number representation (namely, BFloat16 arithmetic, and *exp(x)* function in SoftPosit), we simply convert the given representation to Float32, fall back on the internal Float32 arithmetic for computation, and convert the generated output to the original type. This ensures fairness of comparison against the Float32 gold standard. The timeout for memory management is set as 2 hours in our evaluation.

7 EVALUATION

We refer to MinUn parameterized with the posit representation and the fixed-point representation as MinUn-Posit and MinUn-Fixed, respectively. We will also refer to vanilla Shiftry as Shiftry-Fixed, and our adaption of Shiftry to posits as Shiftry-Posit. Here, we show evaluation to justify the following claims:

(1) Running ML models designed for tiny devices with low-precision 8-bit representations incurs unacceptable accuracy drops (**Figure 6**) and high-precision 16-bit representations preserve accuracy (**Table 3**).
(2) MinUn-Posit generated code that mixes 16-bit high-precision and 8-bit low-precision matches the accuracy of 32-bit floating-point models while consuming significantly less RAM (**Figure 7**).
(3) Haunter, the bitwidth assignment mechanism of MinUn, outperforms the mechanisms that are exclusively accuracy-based or exclusively size-based on both fixed-point (**Table 4**) and posit representations (**Table 5**).

(4) For some models, using optimum memory management significantly lowers the RAM consumption compared to a standard first-fit heuristic (**Section 7.6** and **Table 9** and **Table 10** in **Appendix A**).
(5) Even on a large ImageNet-scale model [Iandola et al. 2016], MinUn outperforms the prior TinyML frameworks (**Table 6**).

We run MinUn generated fixed-point code on two ARM microcontrollers, an Arduino Due with 96KB RAM (for classification models) and an STM32H747 board with 1MB RAM (for localization models). Note that the RAM usage (given by the size of *scratch* buffer) and model accuracy are independent of the underlying hardware and improvements in them are oblivious to the microcontroller used. What depends on the microcontroller is the latency of the program. As expected, MinUn's improvements in memory usage have negligible impact on the execution latency on microcontrollers. We relegate this evaluation to **Table 11** in **Appendix A**, and focus on memory and accuracy here.

We compare MinUn with Shiftry [Kumar et al. 2020] that uses both 8-bit and 16-bit fixed-point arithmetic and TFLite [Jacob et al. 2018] that uses 8-bit zero-skew representation. Since SeeDot [Gopinath et al. 2019] uses 16-bit fixed-point, it is guaranteed to have worse RAM consumption than MinUn and we omit comparisons with it.

7.1 Classification Models and Datasets

We evaluate on three state-of-the-art TinyML classification models developed for resource constrained devices, namely ProtoNN [Gupta et al. 2017], Bonsai [Kumar et al. 2017], and FastGRNN [Kusupati et al. 2018]. These are suitable for running on Arduino class devices, which support a minimum of 2KBs of SRAM and 40KBs of Flash. For fairness of evaluation, we cover the entire suite of datasets evaluated by Shiftry. These include cifar [Krizhevsky 2009], character recognition (cr) [de Campos et al. 2009], usps [Hull 1994], curet [Varma and Zisserman 2005], letter [Hsu and Lin 2002], ward [Yang et al. 2009] and mnist [LeCun et al. 1998] for ProtoNN and Bonsai, and dsa [Altun et al. 2010], google [Warden 2018], har [Anguita et al. 2012], mnist [LeCun et al. 1998], usps [Hull 1994], industrial [Kumar et al. 2020] and wakeword [Kusupati et al. 2018] for FastGRNN.

7.2 Localization Models and Datasets

RNNPool [Saha et al. 2020] is a new pooling layer developed for reducing the sizes of convolutional outputs, while retaining sufficient information for downstream tasks, thereby saving compute and reducing memory footprint. Using RNNPool, we design three face detection models (Face-A, Face-B and Face-C) for classroom/conference setting, which take as input $320 \times 240 \times 1$ monochromatic QVGA images. The Face-C model is a direct replication of the RNNPool-Face-M4 model introduced in [Saha et al. 2020]. The architectures of these models are summarised in **Table 7** in **Appendix A**. These RNNPool-based models are designed for ARM Cortex-M class devices [Jaiswal et al. 2021], which support a minimum of 256KBs of SRAM and 1MB of Flash. We train these models on the WIDER FACE [Yang et al. 2016] dataset and fine-tune them on SCUT-HEAD [Peng et al. 2018] dataset. For model accuracy, we evaluate the generated codes on 20% validation split (405 images) of SCUT-HEAD Part-B, and report the MAP scores.

7.3 Comparison With 8-Bit Baselines

We aim to justify our first claim by comparing the accuracy of models with different 8-bit number representations against the accuracy of the Float32 gold standard. **Figure 6** (data in **Table 8** of **Appendix A**) shows that 8-bit representations have poor accuracy. In the case of 8-bit posits

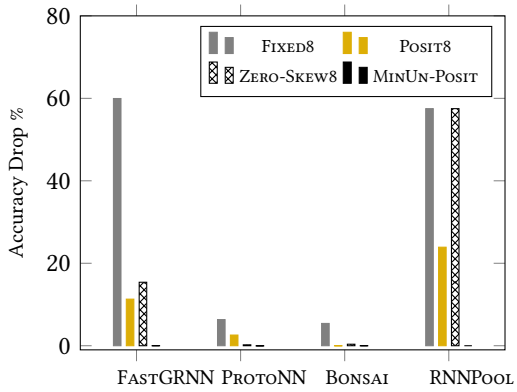

Fig. 6. Accuracy/MAP drop against Float32 gold standard for different models (lower is better).

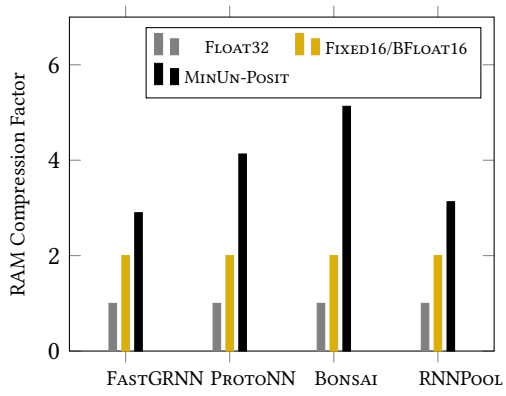

Fig. 7. RAM compression relative to Float32 gold standard for different models (higher is better).

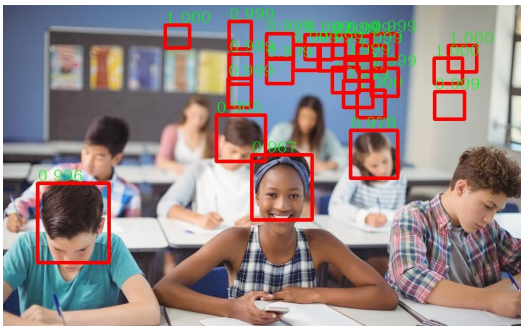

Fig. 8. Example using Face-C model on Shiftry-Fixed.

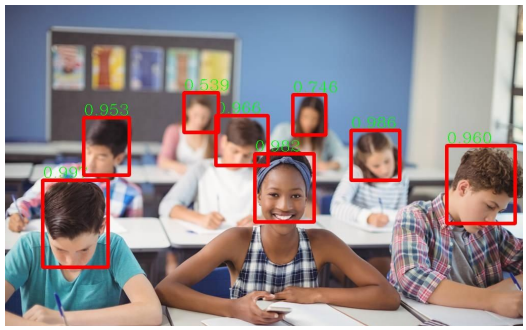

Fig. 9. Example using Face-C model on MinUn-Fixed.

(Posit8), we simply pick the *es* which empirically leads to better accuracy on a dataset. 8-bit fixed-point (Fixed8) gives the worst accuracy, 8-bit zero-skew representation of TFLite (Zero-Skew8) performs better, and Posit8 performs the best among the 8-bit representations. However, even the accuracy loss of Posit8 is unacceptable for FastGRNN and RNNPool. By using high bitwidth 16-bit posits sparingly, the accuracy drop of MinUn-Posit is low for all models.

7.4 Comparison With 16-Bit Baselines

We compare the accuracy and RAM consumption of MinUn-Posit against the gold-standard Float32 baseline. Since 16-bit representations also offer comparable accuracy with certain models at about 2× less RAM consumption, we compare with 16-bit fixed-point (Fixed16) and BFloat16 as well. We generate our MinUn-Posit results while ensuring that the accuracy obtained stays within 0.2% of the corresponding Float32 result, for each dataset. **Table 3** summarises our experiments, where we observe negligible differences in accuracy compared to Float32, while achieving significant reduction in peak RAM consumption, even compared to Fixed16 or BFloat16. The corresponding RAM usage reductions (2.9×–5.1×) are summarised in **Figure 7**.

Table 3. Performance of MinUn-Posit on FastGRNN, ProtoNN, Bonsai and RNNPool models against Float32, Fixed16 and BFloat16 baselines. Accuracy is in percent, and size is in bytes. For classification-based models, the number of classes follow the dataset name. Average accuracy/MAP drop is calculated with respect to the Float32 value.

Model & Dataset	Float32		Fixed16		BFloat16		MinUn-Posit		
	Accuracy	RAM	Accuracy	RAM	Accuracy	RAM	Accuracy	RAM (OPT)	Bitwidth
FastGRNN									
DSA-19	77.76	1280	77.70	640	77.59	640	78.00	448	{8, 12}
WAKEWORD-2	98.97	640	98.74	320	98.97	320	99.20	192	{8, 12}
GOOGLE-12	92.99	2000	92.99	1000	92.76	1000	92.83	700	{10, 12}
GOOGLE-30	78.74	2000	78.86	1000	78.76	1000	78.70	750	{8, 12}
HAR-2	91.65	1600	91.52	800	91.55	800	91.55	500	{8, 10}
HAR-6	92.03	1600	92.03	800	91.52	800	91.99	560	{8, 12}
USPS-10	94.12	1664	94.12	832	94.12	832	94.17	448	{8, 10}
MNIST-10	98.05	2560	85.74	1280	98.12	1280	98.10	960	{8, 12}
INDUSTRIAL-72	89.96	1280	89.88	640	89.53	640	89.80	480	{12, 16}
Average Drop	**Base Case**		**0.41**		**0.15**		**-0.01**		
ProtoNN									
CIFAR-2	76.45	312	76.49	156	76.47	156	76.59	70	{8, 12}
CR-2	72.91	412	73.07	206	73.12	206	73.17	94	{8, 12}
CR-62	32.74	552	32.74	276	32.79	276	32.69	171	{8, 12}
CURET-61	54.53	544	54.53	272	54.67	272	54.53	138	{8, 12}
LETTER-26	84.02	308	83.60	154	83.92	154	84.02	81	{8, 12}
MNIST-2	95.21	240	95.23	120	95.25	120	95.30	39	{8, 10}
MNIST-10	92.06	244	92.04	122	92.09	122	92.10	49	{8, 10}
USPS-2	94.27	240	94.27	120	94.32	120	94.47	44	{8, 12}
USPS-10	92.53	3240	92.58	1620	92.58	1620	92.48	800	{8, 9}
WARD-2	94.87	212	94.87	106	94.82	106	94.82	39	{8, 10}
Average Drop	**Base Case**		**0.02**		**-0.05**		**-0.06**		
Bonsai									
CIFAR-2	75.95	320	75.94	160	75.91	160	75.90	40	{8, 12}
CR-2	74.60	320	74.66	160	74.71	160	74.97	40	{8, 12}
CR-62	10.76	1072	10.81	536	10.76	536	10.76	258	{8, 12}
CURET-61	45.55	1072	45.55	536	45.62	536	45.55	255	{8, 12}
LETTER-26	65.04	496	65.10	248	64.98	248	64.84	114	{8, 12}
MNIST-2	95.96	160	95.96	80	95.96	80	95.98	20	{8, 12}
MNIST-10	93.56	240	93.47	120	93.51	120	93.44	50	{8, 12}
USPS-2	94.82	480	94.82	240	94.82	240	94.92	60	{8, 12}
USPS-10	91.63	304	91.63	152	91.53	152	91.63	45	{8, 12}
WARD-2	95.13	160	95.18	80	95.13	80	95.18	20	{8, 12}
Average Drop	**Base Case**		**-0.01**		**0.01**		**-0.02**		
RNNPool									
FACE-A	0.643	618K	0.628	309K	0.646	309K	0.647	197K	{10, 12}
FACE-B	0.336	614K	0.301	307K	0.335	307K	0.333	192K	{10, 12}
FACE-C	0.575	618K	0.562	309K	0.574	309K	0.573	203K	{10, 12}
Average Drop	**Base Case**		**0.021**		**0.000**		**0.000**		

7.5 Bitwidth Assignment

We compare MinUn's bitwidth assignment mechanism with the mechanism of Shiftry and other exclusively accuracy-based and exclusively size-based mechanisms in both posit and fixed-point. In particular, **Figure 8** and **Figure 9** show qualitatively the difference between the output of MinUn and Shiftry on the image in **Figure 1**.

In size-based baseline, we promote all tensors to higher bitwidth. Then, re-using Shiftry's cumulative demotion process, we demote individual tensors, by the order of their sizes, till we arrive at a bitwidth configuration which satisfies the memory constraint being considered. Since one can initiate cumulative demotion by either considering increasing or decreasing order of tensors sizes, we report the best out of those two possible results for each dataset.

Similarly, for accuracy-based baseline, we add the same memory constraint to Shiftry's demotion process. Here, we order the tensors by their individual demotion accuracies (i.e., the accuracy obtained by demoting a single tensor of interest to lower bitwidth, while keeping the remaining tensors in higher bitwidth) in decreasing order, and initiate the demotion process, and stop when we arrive at a bitwidth configuration which satisfies the limit. We also separately consider a tensor "promotion" experiment, where we initiate a cumulative promotion process after initializing all tensors to lower bitwidth, and promote tensors by the increasing order of their individual demotion accuracies, till we stay within the consumption limit. Here as well, we report the best out of these two results for each dataset.

In all of our experiments, we consider the RAM consumption observed using Shiftry as the memory limit to be adhered to, and use soft limit factors from $\{1.0, 1.1\}$ for fixed-point, and $\{0.8, 0.9, 1.0, 1.1\}$ for our posit experiments. In case of MinUn's result, we provide the RAM consumption values while employing the first-fit greedy heuristic, as well as the optimum memory management using Algorithm X. **Table 4** summarises our experiments for fixed-point representation, with average accuracy drops against MinUn-Fixed mentioned for each exploration strategy on a per model basis. We observe better average accuracy numbers across most models and baselines while strictly staying within the same RAM constraints. Similarly, **Table 5** summarises our experiments for posits, where we compare Shiftry-Posit against MinUn-Posit. Here as well, we observe better average accuracy numbers across all models and baselines while strictly staying within the same RAM constraints. Refer **Table 9** and **Table 10** in **Appendix A** for more details.

An acute observer might notice that the average accuracy drop for Shiftry-Fixed is worse than that of accuracy-based heuristic. It is primarily because Shiftry's use of accuracy information is slightly different. Shiftry sets every tensor to 16-bits, demotes a single tensor at a time to 8-bits, and measures accuracy of the new configuration. Based on this information, it orders the tensors from least accuracy drop to most accuracy drop, and then cumulatively demotes the tensors till the overall accuracy drop stays within 2% of the homogeneous 16-bit configuration accuracy. On the other hand, the accuracy-based heuristic defines the memory constraint as the peak RAM usage achieved by Shiftry's result, and aims to maximize the accuracy through cumulative promotion or cumulative demotion.

7.6 Evaluation on SqueezeNet

Table 6 evaluates MinUn-Posit on SqueezeNet [Iandola et al. 2016] - an ImageNet-scale CNN that matches the accuracy of AlexNet [Krizhevsky et al. 2012] while having 50× smaller size. While SqueezeNet wasn't designed for being deployed on low-RAM devices, we demonstrate the versatility of our framework in achieving significant RAM usage reductions even for large models through this experiment. Note that for even larger CNNs like ResNet50 and MobileNets, keeping all tensors in low bitwidth maintains accuracy [Jacob et al. 2018], which makes bitwidth assignment

Table 4. Average accuracy/MAP drop of computationally expensive exploration strategies and the previous state-of-the-art on FastGRNN, ProtoNN, Bonsai and RNNPool models against MinUn-Fixed value.

Model	Shiftry-Fixed Average Drop	Accuracy-Based Average Drop	Size-Based Average Drop
FastGRNN	0.5%	-0.2%	-0.27%
ProtoNN	0.74%	0.48%	0.71%
Bonsai	0.58%	0.30%	0.25%
RNNPool	0.233	-0.003	-0.003

Table 5. Average accuracy/MAP drop of computationally expensive exploration strategies and the previous state-of-the-art on FastGRNN, ProtoNN, Bonsai and RNNPool models against MinUn-Posit value.

Model	Shiftry-Posit Average Drop	Accuracy-Based Average Drop	Size-Based Average Drop
FastGRNN	0.58%	0.02%	0.04%
ProtoNN	0.47%	0.17%	1.73%
Bonsai	0.08%	0.14%	0.06%
RNNPool	0.030	0.012	0.012

Table 6. Evaluation of different frameworks on SqueezeNet.

Framework	Accuracy (%)	RAM (MBs)
Float32	55.02	4.42
Shiftry-Fixed (8-bit and 16-bit)	48.33	2.21
TFLite-Zero-Skew8	45.73	1.11
MinUn-Posit (8-bit and 16-bit)	54.81	1.16

trivial. To measure accuracy, we evaluate the generated code on 48,000 RGB images spread across 1000 classes. We observe a peak RAM consumption of 1.44 MBs on the code generated by the standard first-fit [Nutt 2002] heuristic for memory management on MinUn-Posit, which was further reduced to 1.16 MBs on using our optimum memory management, offering a improvement of almost 20%. As evident from **Table 6**, we are able to achieve upto 3.81x reduction in peak RAM consumption with minimal loss in accuracy, relative to the Float32 gold standard, while existing TinyML frameworks (TFLite and Shiftry) fail to offer comparable reductions without leading to significant accuracy losses.

7.7 Design Decisions

We discuss two approaches to improve MinUn that we evaluated but did not pursue because the gains they provided were small in practice.

7.7.1 Permitting Overflows: For fixed-point, Haunter assigns bitwidths in a manner in which integer overflows are ruled out. However, this strategy is not always the best for achieving high classification accuracy. In particular, Shiftry permits values to overflow on some inputs to achieve higher precision on other inputs, so that the overall classification accuracy is high. While this optimization works out well in the case of fixed-point representation (where overflows are much

better understood), it's effect is hard to predict in case of other arbitrary number representations. Since Haunter doesn't permit overflows, it misses out on this trade-off. This limitation is best captured through FastGRNN model on HAR-6 dataset (details in **Table 9** of **Appendix A**), where the generated model takes a hit of almost 2% compared to other strategies, and is the only cause of the average accuracy drops being negative for FastGRNN models in **Table 4**.

7.7.2 Generalization to More Bitwidth Options: Haunter decides between two bitwidths, *lowBitwidth* and *highBitwidth*. It is plausible that if there are more bitwidth options then Haunter can generate even better code. Having k bitwidth options leads to a total of k^N possible bitwidth assignments, where N is the number of tensors in the program. In general, adding more bitwidth options directly into the algorithm blows up the search space and leads to many code executions, making MinUn painfully slow. We experimented with three bitwidths (8, 12 and 16) for posits, but did not find any tangible benefit in terms of accuracy in our experiments compared to two bitwidth options of (8 and 16) or (8 and 12) or (12 and 16) cases, depending on specific models.

As such, we chose a much simpler strategy in **Table 3**. Given $B = \{8, 9, 10, 12, 16\}$ options for bitwidths, we generated $|B|$ homogeneous bitwidth codes (one for each option) and ordered the bitwidths by the accuracy/MAP obtained on their corresponding codes. Out of these sorted bitwidths, we simply chose the two options between which the 32-bit floating point accuracy lied, as the *lowBitwidth* and *highBitwidth* respectively. In situations where there was a tie in accuracy between multiple options for *lowBitwidth* or *highBitwidth*, we simply picked the lowest of the tied bitwidth options. In total, these only require $|B|$ additional code generations and executions, which is affordable since $|B| = 5$ is small.

8 RELATED WORK

Haunter is related to floating-point tuning algorithms like Precimonious [Rubio-González et al. 2013]. However, their objective is completely orthogonal to the problem considered by MinUn. Haunter aims to maximize the accuracy of a program while strictly respecting the memory usage constraints, while the Precimonious algorithm aims to maximize performance (in terms of latency of execution) while strictly respecting the precision constraints. We are not aware of any floating-point heuristic tuning algorithm which has the same objectives as Haunter.

Running ML models with minimum resources is a vast subject and we do not attempt to survey it here. The design of new CNN architectures manually [Huang et al. 2017; Sandler et al. 2018] or automatically with Network Architecture Search (NAS) [Cai et al. 2018; Shaw et al. 2019; Tan and Le 2019] focuses on reducing model size and compute requirements but not the peak RAM usage, which is our focus here; if the peak RAM usage exceeds the available RAM then the model fails. Techniques like pruning channels/filters [He et al. 2017] and spatial downsampling [Saha et al. 2020] reduce peak memory usage and are complementary to MinUn.

There are various approaches for quantization in the ML literature. In *hybrid-quantization* [Iandola et al. 2016; Jacob et al. 2018] on smartphones, each low bitwidth tensor is converted to 32-bit floating-point at runtime, computed upon in floating-point, and then converted back to low bitwidth. This approach is untenable for the models and devices that we consider because we have observed that even converting a single quantized tensor to 32-bit overflows the RAM. Apart from Shiftry [Kumar et al. 2020], prior papers on quantization use the trivial bitwidth assignment of mapping all tensors in large CNNs to low bitwidths [Chen et al. 2019; Courbariaux and Bengio 2016; Gong et al. 2019; He and Fan 2019; Hou et al. 2019; Hubara et al. 2016; Jacob et al. 2018; Krishnamoorthi 2018; Li et al. 2017; Lin et al. 2015; Louizos et al. 2019; Martinez et al. 2018; Meller et al. 2019; Nagel et al. 2019; Rastegari et al. 2016; Sakr and Shanbhag 2019; Zhao et al. 2019a; Zhou et al. 2018]. Most of these approaches rely on retraining a low-bitwidth model, which is a

computationally very expensive process. Thus, "post-training quantization" frameworks [Gopinath et al. 2019; Jacob et al. 2018; Kumar et al. 2020] like MinUn offer a distinct advantage.

MinUn can be thought of as an approximation framework for ML models. Most existing approximation frameworks map floating-point programs to other floating-point programs [Baek and Chilimbi 2010; Panchekha et al. 2015; Rubio-González et al. 2013; Schkufza et al. 2014; Sidiroglou-Douskos et al. 2011; Zhu et al. 2012]. Converters from floating-point to fixed-point that have not been designed for ML include [Babb et al. 1999; Banerjee et al. 2003; Bečvář and Štukjunger 2005; Brooks and Martonosi 1999; Darulova and Kuncak 2014, 2017; Darulova et al. 2013; Menard et al. 2002; Nayak et al. 2001; Willems 1997].

9 CONCLUSION

MinUn is the first TinyML framework which is parametric on number representation, generates a bitwidth assignment automatically taking into account both the size of the tensors and their impact on accuracy, and performs optimum memory management. Our evaluation shows that MinUn significantly outperforms state-of-the-art TinyML frameworks in accuracy and RAM consumption.

Prior works are almost unilaterally fixated on a single number representation, and make use of a number of optimizations specific to their use-case. MinUn's contribution is unique, in the sense that it generalizes the number representation to be an adjustable feature. Hence, each and every number representation uses the same set of optimizations and exploration strategies, which doesn't favor one over the other. While one might expect lower accuracy because of this generalization, our evaluation surprisingly shows that except for a few outliers, our work still outperforms individually tuned approaches. This makes MinUn a versatile framework for trying out arbitrary number representations and comparing them in a fair manner.

MinUn-Posit compiles ML models to posits and hence is related to the rapidly evolving developer ecosystem for posits. Once hardware for posits [Podobas and Matsuoka 2018b] becomes commercially available, we will be able to run MinUn-Posit generated code natively instead of simulating it in software.

There are two future directions in which MinUn can be extended but are beyond the scope of this work. First, we would like to generate code that mixes different number representations to further decrease the memory footprint. Such implementations can then be burned on an FPGA for low-power inference. Unlike microcontrollers, FPGAs can be reconfigured to provide native hardware support for arbitrary number representations. Second, we would like to automatically synthesize model-specific number representations that can generate the best implementation for a given ML model. A number representation is a function from reals to bitvectors and synthesizing the best number representation for a given model would be a more ambitious goal that would bring us closer to implementations with optimum memory consumption.

ACKNOWLEDGMENTS

We would like to thank Oindrila Saha, Shubham Ugare, Prateek Jain and Harsha Vardhan Simhadri for their useful insights and discussions on this work.

REFERENCES

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/bfloat16.cc. https://doi.org/10.5281/zenodo.4724125

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

Kerem Altun, Billur Barshan, and Orkun Tunçel. 2010. Comparative study on classifying human activities with miniature inertial and magnetic sensors. *Pattern Recognition* 43, 10 (2010), 3605–3620. https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities

Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge L. Reyes-Ortiz. 2012. Human Activity Recognition on Smartphones Using a Multiclass Hardware-Friendly Support Vector Machine. In *Proceedings of the 4th International Conference on Ambient Assisted Living and Home Care* (Vitoria-Gasteiz, Spain) *(IWAAL'12)*. Springer-Verlag, Berlin, Heidelberg, 216–223. https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones

ARM. 2021. ARM Cortex-M official website. https://developer.arm.com/ip-products/processors/cortex-m.

Jonathan Babb, Martin C. Rinard, Csaba Andras Moritz, Walter Lee, Matthew I. Frank, Rajeev Barua, and Saman P. Amarasinghe. 1999. Parallelizing Applications into Silicon. In *7th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '99), 21-23 April 1999, Napa, CA, USA*. IEEE, Napa, CA, USA, 70. https://doi.org/10.1109/FPGA.1999.803669

Woongki Baek and Trishul M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Association for Computing Machinery, New York, NY, USA, 198–209. https://doi.org/10.1145/1809028.1806620

P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe. 2003. Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware design. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.* IEEE Computer Society, Los Alamitos, CA, USA, 263–264. https://doi.org/10.1109/FPGA.2003.1227262

Massimo Banzi and Michael Shiloh. 2014. *Getting started with Arduino: the open source electronics prototyping platform.* Maker Media, Inc.

M Bečvář and P Štukjunger. 2005. Fixed-point arithmetic in FPGA. *Acta Polytechnica* 45, 2 (2005).

David M. Brooks and Margaret Martonosi. 1999. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture, Orlando, FL, USA, January 9-12, 1999*. Association for Computing Machinery, New York, NY, USA, 13–22.

Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).

Shangyu Chen, Wenya Wang, and Sinno Jialin Pan. 2019. MetaQuant: Learning to Quantize by Learning to Penetrate Non-differentiable Quantization. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 3916–3926. http://papers.nips.cc/paper/8647-metaquant-learning-to-quantize-by-learning-to-penetrate-non-differentiable-quantization.pdf

Xi Chen, Xiaolin Hu, Hucheng Zhou, and Ningyi Xu. 2017. FxpNet: Training a deep convolutional neural network in fixed-point representation. In *2017 International Joint Conference on Neural Networks, IJCNN 2017, Anchorage, AK, USA, May 14-19, 2017*. IEEE, 2494–2501. https://doi.org/10.1109/IJCNN.2017.7966159

Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and detecting numerical errors in computation with posits. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 731–746.

Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR* abs/1602.02830 (2016). arXiv:1602.02830 http://arxiv.org/abs/1602.02830

Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/2535838.2535874

Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (March 2017), 28 pages. https://doi.org/10.1145/3014426

Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. 2013. Synthesis of Fixed-point Programs. In *Proceedings of the Eleventh ACM International Conference on Embedded Software* (Montreal, Quebec, Canada) *(EMSOFT '13)*. IEEE Press, Piscataway, NJ, USA, Article 22, 10 pages. http://dl.acm.org/citation.cfm?id=2555754.2555776

Teófilo Emídio de Campos, Bodla Rakesh Babu, and Manik Varma. 2009. Character Recognition in Natural Images. In *VISAPP 2009 - Proceedings of the Fourth International Conference on Computer Vision Theory and Applications, Lisboa, Portugal, February 5-8, 2009 - Volume 2*. INSTICC Press, Portugal, 273–280. https://www.researchgate.net/publication/221416071_Character_Recognition_in_Natural_Images

Don Dennis, Durmus Alp Emre Acar, Vikram Mandikal, Vinu Sankar Sadasivan, Harsha Vardhan Simhadri, Venkatesh Saligrama, and Prateek Jain. 2019. Shallow RNN: Accurate Time-series Classification on Resource Constrained Devices. In *Proceedings of the Thirty-second Annual Conference on Neural Information Processing Systems (NeurIPS)*.

Don Dennis, Chirag Pabbaraju, Harsha Vardhan Simhadri, and Prateek Jain. 2018. Multiple Instance Learning for Efficient Sequential Data Classification on Resource-constrained Devices. In *Proceedings of the Thirty-first Annual Conference on Neural Information Processing Systems (NeurIPS)*. 10976–10987. all_papers/DennisPSJ18.pdf slides/DennisPSJ18.pdf.

Josh Fromm, Shwetak Patel, and Matthai Philipose. 2018. Heterogeneous Bitwidth Binarization in Convolutional Neural Networks. *CoRR* abs/1805.10368 (2018). arXiv:1805.10368 http://arxiv.org/abs/1805.10368

Nikhil Pratap Ghanathe, Vivek Seshadri, Rahul Sharma, Steve Wilton, and Aayan Kumar. 2021. MAFIA: Machine Learning Acceleration on FPGAs for IoT Applications. In *31st International Conference on Field-Programmable Logic and Applications, FPL 2021, Dresden, Germany, August 30 - Sept. 3, 2021*. 347–354.

Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. 2019. Differentiable Soft Quantization: Bridging Full-Precision and Low-Bit Neural Networks. In *The IEEE International Conference on Computer Vision (ICCV)*. https://doi.org/10.1109/ICCV.2019.00495

Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019. Compiling KB-Sized Machine Learning Models to Tiny IoT Devices. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, 79–95. https://www.microsoft.com/en-us/research/uploads/prod/2018/10/pldi19-SeeDot.pdf

Posit Working Group. 2018. Posit Standard Documentation. https://positbub.org/docs/posit_standard.pdf.

Denis A. Gudovskiy and Luca Rigazio. 2017. ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks. *CoRR* abs/1706.02393 (2017). https://www.researchgate.net/publication/317419072_ShiftCNN_Generalized_Low-Precision_Architecture_for_Inference_of_Convolutional_Neural_Networks

Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. ProtoNN: compressed and accurate kNN for resource-scarce devices. In *International Conference on Machine Learning*. PMLR, International Convention Centre, Sydney, Australia, 1331–1340. https://dl.acm.org/doi/10.5555/3305381.3305519

Gustafson and Yonemoto. 2017. Beating Floating Point at Its Own Game: Posit Arithmetic. *Supercomput. Front. Innov.: Int. J.* 4, 2 (June 2017), 71–86. https://doi.org/10.14529/jsfi170206

John Gustafson. 2017. Posit Arithmetic. (2017). https://positbub.org/docs/Posits4.pdf

Yonghao He, Dezhong Xu, Lifang Wu, Meng Jian, Shiming Xiang, and Chunhong Pan. 2019. LFFD: A Light and Fast Face Detector for Edge Devices. In *arXiv:1904.10633*.

Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. 1398–1406.

Zhezhi He and Deliang Fan. 2019. Simultaneously Optimizing Weight and Quantizer of Ternary Neural Network Using Truncated Gaussian Approximation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://arxiv.org/abs/1810.01018

Lu Hou, Jinhua Zhu, James Kwok, Fei Gao, Tao Qin, and Tie-Yan Liu. 2019. Normalization Helps Training of Quantized LSTM. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 7346–7356. http://papers.nips.cc/paper/8954-normalization-helps-training-of-quantized-lstm.pdf

Chih-Wei Hsu and Chih-Jen Lin. 2002. A comparison of methods for multiclass support vector machines. *IEEE transactions on Neural Networks* 13, 2 (2002), 415–425. https://doi.org/10.1109/72.991427

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 4107–4115. http://papers.nips.cc/paper/6573-binarized-neural-networks.pdf

Jonathan J. Hull. 1994. A database for handwritten text recognition research. *IEEE Transactions on pattern analysis and machine intelligence* 16, 5 (1994), 550–554. https://doi.org/10.1109/34.291440

Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). https://arxiv.org/abs/1602.07360v1

IEEE. 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Benoit Jacob, Pete Warden, Miao Wang, David Andersen, Maciek Chociej, Justine Tunney, Mark J. Matthews, Marie White, Suharsh Sivakumar, Sagi Marcovich, Murat Efe Guney, Sarah Knepper, Mourad Gouicem, Richard Winterton, David Mansell, Andreas Gal, Alexey Frunze, and Alexey Frunze. 2015. gemmlowp: a small self-contained low-precision gemm library. https://github.com/google/gemmlowp.

Shikhar Jaiswal, Oindrila Saha, Aayan Kumar, Harsha Vardhan Simhadri, Prateek Jain, and Rahul Sharma. 2021. Enabling Accurate Computer Vision on Tiny Microcontrollers with RNNPool Operator and SeeDot Compiler. https://towardsdatascience.com/enabling-accurate-computer-vision-on-tiny-microcontrollers-with-rnnpool-operator-and-seedot-d6944930dcf9

Irena Jekova and Vessela Krasteva. 2004. Real Time detection of ventricular fibrillation and tachycardia. *Physiological measurement* 25 (11 2004), 1167–78. https://doi.org/10.1088/0967-3334/25/5/007

Jeff Johnson. 2018. Rethinking floating point for deep learning. *CoRR* abs/1811.01721 (2018). https://arxiv.org/abs/1811.01721

Don E. Knuth. 2000. Dancing Links. *CoRR* abs/cs/0011047 (2000). https://arxiv.org/abs/cs/0011047

Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. *CoRR* abs/1711.02213 (2017). https://arxiv.org/abs/1711.02213

Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR* abs/1806.08342 (2018). https://arxiv.org/abs/1806.08342

Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images.* Technical Report. Citeseer.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.* 1106–1114. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In *International Conference on Machine Learning*. PMLR, International Convention Centre, Sydney, Australia, 1935–1944. https://dl.acm.org/doi/10.5555/3305381.3305581

Aayan Kumar, Vivek Seshadri, and Rahul Sharma. 2020. Shiftry: RNN Inference in 2KB of RAM. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 182 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428250

Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. 2018. FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network. In *Proceedings of the Thirty-first Annual Conference on Neural Information Processing Systems (NeurIPS)*. 9031–9042. https://dl.acm.org/doi/10.5555/3327546.3327577

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

Siew Hoon Leong. 2020. SoftPosit. https://gitlab.com/cerlane/SoftPosit. https://doi.org/10.5281/zenodo.3709035

Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. 2017. Training Quantized Nets: A Deeper Understanding. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5811–5821. http://papers.nips.cc/paper/7163-training-quantized-nets-a-deeper-understanding.pdf

Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2021. An approach to generate correctly rounded math libraries for new floating point variants. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30.

Jay P. Lim and Santosh Nagarakatte. 2021. High performance correctly rounded math libraries for 32-bit floating point representations. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 359–374.

Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. 2021. MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.

Ji Lin, Wei-Ming Chen, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.

Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural Networks with Few Multiplications. *CoRR* abs/1510.03009 (2015). arXiv:1510.03009 http://arxiv.org/abs/1510.03009

Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. 2019. Relaxed Quantization for Discretized Neural Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HkxjYoCqKX

Julieta Martinez, Shobhit Zakhmi, Holger H. Hoos, and James J. Little. 2018. LSQ++: Lower running time and higher recall in multi-codebook quantization. In *The European Conference on Computer Vision (ECCV)*. https://openaccess.thecvf.com/content_ECCV_2018/papers/Julieta_Martinez_LSQ_lower_runtime_ECCV_2018_paper.pdf

Eldad Meller, Alexander Finkelstein, Uri Almog, and Mark Grobman. 2019. Same, Same But Different - Recovering Neural Network Quantization Error Through Weight Factorization. *CoRR* abs/1902.01917 (2019). https://arxiv.org/abs/1902.01917

Daniel Menard, Daniel Chillet, François Charot, and Olivier Sentieys. 2002. Automatic Floating-point to Fixed-point Conversion for DSP Code Generation. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (Grenoble, France) *(CASES '02)*. Association for Computing Machinery, New York, NY, USA, 270–276. https://doi.org/10.1145/581630.581674

Daisuke Miyashita, Edward H. Lee, and Boris Murmann. 2016. Convolutional Neural Networks using Logarithmic Data Representation. *CoRR* abs/1603.01025 (2016). https://arxiv.org/abs/1603.01025

Elon Musk. 2019. An Integrated Brain-Machine Interface Platform With Thousands of Channels. *J Med Internet Res* 21, 10 (31 Oct 2019), e16194. https://doi.org/10.2196/16194

Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. 2019. Data-Free Quantization Through Weight Equalization and Bias Correction. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 1325–1334. https://openaccess.thecvf.com/content_ICCV_2019/papers/Nagel_Data-Free_Quantization_Through_Weight_Equalization_and_Bias_Correction_ICCV_2019_paper.pdf

Anshuman Nayak, Malay Haldar, Alok N. Choudhary, and Prithviraj Banerjee. 2001. Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001, Munich, Germany, March 12-16, 2001*. 722–728. https://doi.org/10.1109/DATE.2001.915108

Gary J. Nutt. 2002. *Operating Systems: A Modern Perspective*. Addison Wesley. https://books.google.co.in/books?id=AHBGAAAAYAAJ

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 1–11.

Dezhi Peng, Zikai Sun, Zirong Chen, Zirui Cai, Lele Xie, and Lianwen Jin. 2018. Detecting Heads using Feature Refine Net and Cascaded Multi-scale Architecture. *CoRR* abs/1803.09256 (2018). arXiv:1803.09256 http://arxiv.org/abs/1803.09256

Artur Podobas and Satoshi Matsuoka. 2018a. Hardware Implementation of POSITs and Their Application in FPGAs. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 138–145.

Artur Podobas and Satoshi Matsuoka. 2018b. Hardware Implementation of POSITs and Their Application in FPGAs. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 138–145. https://doi.org/10.1109/IPDPSW.2018.00029

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR* abs/1603.05279 (2016). arXiv:1603.05279 http://arxiv.org/abs/1603.05279

Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 27, 12 pages. https://doi.org/10.1145/2503210.2503296

Oindrila Saha, Aditya Kusupati, Harsha Vardhan Simhadri, Manik Varma, and Prateek Jain. 2020. RNNPool: Efficient Non-linear Pooling for RAM Constrained Inference. In *Advances in Neural Information Processing Systems*.

Charbel Sakr and Naresh Shanbhag. 2019. Per-Tensor Fixed-Point Quantization of the Back-Propagation Algorithm. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rkxaNjA9Ym

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 53–64. https://doi.org/10.1145/2666356.2594302

Abhishek A. Sharma. 2014. A Consolidated Review on Embedded MicroControllers for Pace Maker Applications. In *The 2014 International Conference on Embedded Systems and Applications*.

Albert Shaw, Daniel Hunter, Forrest Landola, and Sammy Sidhu. 2019. SqueezeNAS: Fast neural architecture search for faster semantic segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*. 0–0.

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 124–134. https://doi.org/10.1145/2025113.2025133

Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*. PMLR, 6105–6114.

ONNX Development Team. 2021. ONNX Model Zoo. https://github.com/onnx/models/

Manik Varma and Andrew Zisserman. 2005. A statistical approach to texture classification from single images. *International journal of computer vision* 62, 1-2 (2005), 61–81. https://doi.org/10.1023/B:VISI.0000046589.39864.ee

Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8612–8620.

Pete Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. arXiv:1804.03209 [cs.CL]

M. Willems. 1997. FRIDGE : Floating-point programming of fixed-point digital signal processors. *Proc. International Conference on Signal Processing Applications and Technology 1997 (ICSPAT-97), Sept.* (1997). https://ci.nii.ac.jp/naid/10018558547/en/

Jingjing Yang, Yuanning Li, Yonghong Tian, Lingyu Duan, and Wen Gao. 2009. Group-sensitive multiple kernel learning for object categorization. In *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, Kyoto, Japan, 436–443. https://doi.org/10.1109/ICCV.2009.5459172

Shuo Yang, Ping Luo, Chen Change Loy, and Xiaoou Tang. 2016. WIDER FACE: A Face Detection Benchmark. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Young Joon Yoo, Dongyoon Han, and Sangdoo Yun. 2019. EXTD: Extremely Tiny Face Detector via Iterative Filter Reuse. *CoRR* abs/1906.06579 (2019). arXiv:1906.06579 http://arxiv.org/abs/1906.06579

Shifeng Zhang, Xiangyu Zhu, Zhen Lei, Hailin Shi, Xiaobo Wang, and Stan Z Li. 2017. Faceboxes: A CPU real-time face detector with high accuracy. In *2017 IEEE International Joint Conference on Biometrics (IJCB)*. IEEE, 1–9.

Xu Zhao, Xiaoqing Liang, Chaoyang Zhao, Ming Tang, and Jinqiao Wang. 2019b. Real-Time Multi-Scale Face Detector on Embedded Devices. *Sensors* 19, 9 (2019). https://doi.org/10.3390/s19092158

Yiren Zhao, Xitong Gao, Daniel Bates, Robert Mullins, and Cheng-Zhong Xu. 2019a. Focused Quantization for Sparse CNNs. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 5584–5593. http://papers.nips.cc/paper/8796-focused-quantization-for-sparse-cnns.pdf

Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental Network Quantization: Towards Lossless CNNs with Low-precision Weights. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://arxiv.org/abs/1702.03044v2

Aojun Zhou, Anbang Yao, Kuan Wang, and Yurong Chen. 2018. Explicit Loss-Error-Aware Quantization for Low-Bit Deep Neural Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://openaccess.thecvf.com/content_cvpr_2018/papers/Zhou_Explicit_Loss-Error-Aware_Quantization_CVPR_2018_paper.pdf

Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. 2012. Randomized Accuracy-aware Program Transformations for Efficient Approximate Computations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 441–454. https://doi.org/10.1145/2103621.2103710

A APPENDIX

In this section, we provide detailed results from all of our experiments.

Table 7 summarises the architecture of the face detection models evaluated in our experiments. All of our face detection models comprise more than one inverted bottleneck residual blocks, commonly called *MBConv* (see Fig. 3b of [Sandler et al. 2018]). Each MBConv layer consists of three convolution operations executed sequentially:

- C1: A point-wise convolution which expands the number of channels from c_{in} to $c_{in} \times t$, where t is the expansion factor.
- C2: A depth-wise separable convolution with a kernel size of 3×3 and a stride of either 1 or 2.
- C3: A point-wise convolution which reduces the number of channels in the intermediate tensor from $c_{in} \times t$ to c_{out}.

Table 8 provides accuracy values on different datasets, while comparing different 8-bit representations against the Float32 gold standard. For certain cases, the data-driven scale assignment mechanism of Shiftry is unable to determine the appropriate scales for 8-bit fixed-point numbers, and such instances are removed while calculating average accuracy drops. These results show that 8-bit representations suffer from large losses in accuracy.

Table 9 and **Table 10** provide accuracy and RAM consumption values of different bitwidth assignment mechanisms for fixed-point and posit representation respectively. In both of these experiments, we set the RAM consumption of Shiftry's result as the memory limit for the other strategies. In case of MinUn's result, we provide the RAM consumption values while employing the first-fit greedy heuristic, as well as the optimum memory management using Algorithm X. This evaluation shows that MinUn has better accuracy while consuming less memory than the baselines. Of particular interest is the FACE-B model in **Table 9** where optimum RAM management reduces the RAM usage from 218KB to 165KB.

Table 11 provides the latency values of different models, on codes generated by Shiftry-Fixed and MinUn-Fixed corresponding to **Table 9**. Latency numbers for classification and localization models have been generated on Arduino Due (84 MHz, 96KB SRAM, 512KB Flash) and STM32H747 (240 MHz, 1MB SRAM, 2MB Flash) boards respectively. This evaluation shows that the reduction in RAM usage by MinUn does not degrade latency on microcontrollers.

Table 7. Architecture schematics of face detection models. Each line denotes a sequence of layers, repeated n times. The first layer of each MBConv sequence has stride s and rest use stride 1. Expansion factor t is multiplied to the input channels to change the width. The number of output channels is c.

Input	Layer	t	c	n	s
$320 \times 240 \times 1$	Conv2D 3×3	1	4	1	2
$160 \times 120 \times 4$	RNNPool	1	64	1	4
$40 \times 30 \times 64$	MBConv	2	32	1	1
$40 \times 30 \times 32$	MBConv	2	32	7	1
$40 \times 30 \times 32$	MBConv	2	96	1	2
$20 \times 15 \times 96$	MBConv	2	96	2	1
$20 \times 15 \times 96$	MBConv	2	128	1	1
$20 \times 15 \times 128$	MBConv	2	128	2	1

The architecture of Face-A.

Input	Layer	t	c	n	s
$320 \times 240 \times 1$	Conv2D 3×3	1	4	1	2
$160 \times 120 \times 4$	RNNPool	1	64	1	8
$20 \times 15 \times 64$	MBConv	2	32	1	1
$20 \times 15 \times 32$	MBConv	2	96	1	1
$20 \times 15 \times 96$	MBConv	2	96	1	1
$20 \times 15 \times 96$	MBConv	2	128	1	1

The architecture of Face-B.

Input	Layer	t	c	n	s
$320 \times 240 \times 1$	Conv2D 3×3	1	4	1	2
$160 \times 120 \times 4$	RNNPool	1	64	1	4
$40 \times 30 \times 64$	MBConv	2	32	1	1
$40 \times 30 \times 32$	MBConv	2	32	1	1
$40 \times 30 \times 32$	MBConv	2	64	1	2
$20 \times 15 \times 64$	MBConv	2	64	1	1

The architecture of Face-C.

Table 8. Performance of homogeneous 8-bit codes of different representations on FastGRNN, ProtoNN, Bonsai and RNNPool models against the Float32 gold standard. Accuracy is in percent. For classification-based models, the number of classes follow the dataset name. An × in the column indicates that data-driven compilation fails to determine a suitable scale. Average accuracy/MAP drop is calculated with respect to Float32 value.

Model & Dataset	Float32 Accuracy	Fixed8 Accuracy	Posit8 (Best ES) Accuracy	TFLite-Zero-Skew8 Accuracy
FastGRNN				
DSA-19	77.76	6.56	69.87	76.56
WAKEWORD-2	98.97	95.66	98.52	84.13
GOOGLE-12	92.99	8.08	44.63	84.42
GOOGLE-30	78.74	4.21	51.57	58.92
HAR-2	91.65	50.87	84.83	88.12
HAR-6	92.03	16.36	91.31	91.89
USPS-10	94.12	52.12	93.92	93.87
MNIST-10	98.05	×	96.45	12.04
INDUSTRIAL-72	89.96	2.73	81.50	85.69
Average Drop	**Base Case**	**59.96**	**11.30**	**15.40**
ProtoNN				
CIFAR-2	76.45	71.47	66.18	76.64
CR-2	72.91	69.72	66.33	72.85
CR-62	32.74	30.76	31.33	31.75
CURET-61	54.53	39.42	53.10	54.17
LETTER-26	84.02	80.84	82.80	83.50
MNIST-2	95.21	87.40	94.11	95.23
MNIST-10	92.06	85.98	91.23	91.78
USPS-2	94.27	92.58	93.32	94.27
USPS-10	92.53	90.78	91.83	92.73
WARD-2	94.87	77.21	93.42	94.51
Average Drop	**Base Case**	**6.34**	**2.59**	**0.22**
Bonsai				
CIFAR-2	75.95	73.14	75.90	75.70
CR-2	74.60	72.22	74.97	74.34
CR-62	10.76	9.03	10.76	10.03
CURET-61	45.55	34.50	45.12	44.05
LETTER-26	65.04	60.96	65.28	64.62
MNIST-2	95.96	93.51	95.88	96.03
MNIST-10	93.56	76.32	93.49	93.28
USPS-2	94.82	91.08	94.92	94.67
USPS-10	91.63	89.89	91.83	91.68
WARD-2	95.13	88.30	95.18	94.93
Average Drop	**Base Case**	**5.41**	**-0.03**	**0.37**
RNNPool				
FACE-A	0.643	x	0.560	x
FACE-B	0.336	x	0.209	x
FACE-C	0.575	5e-5	0.069	3e-5
Average Drop	**Base Case**	**0.575**	**0.239**	**0.575**

Table 9. Performance of MinUn-Fixed on FastGRNN, ProtoNN, Bonsai and RNNPool models against computationally expensive bitwidth assignment mechanisms and the previous state-of-the-art. Accuracy is in percent, and size is in bytes. For classification-based models, the number of classes follow the dataset name. Average accuracy/MAP drop is calculated with respect to MinUn-Fixed value.

Model & Dataset	Shiftry-Fixed		Accuracy-Based		Size-Based		MinUn-Fixed		
	Accuracy	RAM	Accuracy	RAM	Accuracy	RAM	Accuracy	RAM	RAM (OPT)
FastGRNN									
DSA-19	74.39	740	**77.70**	640	**77.70**	640	**77.70**	640	640
WAKEWORD-2	98.63	377	**98.74**	320	**98.74**	320	**98.74**	320	320
GOOGLE-12	92.24	1141	**92.99**	1000	**92.99**	1000	**92.99**	1000	1000
GOOGLE-30	78.42	1000	**78.86**	1000	**78.86**	1000	**78.86**	1000	1000
HAR-2	90.09	845	**91.52**	800	**91.52**	800	**91.52**	800	800
HAR-6	90.23	720	90.50	720	**90.84**	720	88.26	730	720
USPS-10	92.98	512	92.98	512	92.88	512	**93.02**	512	512
MNIST-10	**85.74**	1280	**85.74**	1280	**85.74**	1280	**85.74**	1280	1280
INDUSTRIAL-72	87.31	520	87.31	520	**87.71**	520	**87.71**	520	520
Average Drop	**0.5**		**-0.2**		**-0.27**		**Base Case**		
ProtoNN									
CIFAR-2	75.18	93	75.65	93	75.43	81	**76.16**	93	93
CR-2	**72.85**	124	72.53	124	71.32	106	72.64	124	124
CR-62	31.70	272	31.70	272	31.75	226	**32.27**	226	226
CURET-61	53.10	146	**54.10**	145	51.60	138	53.89	145	145
LETTER-26	81.78	77	81.78	77	82.42	77	**83.30**	77	77
MNIST-2	94.80	60	95.07	60	95.12	60	**95.26**	60	56
MNIST-10	91.78	61	**91.95**	61	91.93	61	91.94	61	61
USPS-2	92.83	60	93.92	60	94.02	60	**94.07**	60	56
USPS-10	91.63	810	91.63	810	92.03	810	**92.73**	820	800
WARD-2	93.68	63	93.68	63	94.05	63	**94.51**	63	63
Average Drop	**0.74**		**0.48**		**0.71**		**Base Case**		
Bonsai									
CIFAR-2	74.60	80	74.60	80	74.89	80	**75.08**	80	66
CR-2	74.34	100	74.34	100	**74.87**	100	74.44	100	66
CR-62	9.40	268	9.40	268	9.40	268	**9.56**	288	268
CURET-61	42.77	524	43.41	524	43.98	524	**45.55**	536	488
LETTER-26	63.26	124	63.48	108	63.52	108	**63.52**	122	108
MNIST-2	95.85	50	**96.03**	50	95.99	50	95.92	50	36
MNIST-10	92.70	60	**93.12**	60	92.59	60	93.05	60	60
USPS-2	94.07	120	94.07	120	94.12	120	**94.37**	120	96
USPS-10	91.48	76	91.48	76	91.53	76	**91.63**	80	72
WARD-2	93.94	50	**95.29**	50	94.77	50	95.08	50	36
Average Drop	**0.58**		**0.30**		**0.25**		**Base Case**		
RNNPool									
FACE-A	0.338	235K	**0.635**	213K	**0.635**	213K	0.626	232K	232K
FACE-B	0.314	218K	**0.320**	218K	**0.320**	218K	**0.320**	218K	165K
FACE-C	0.139	204K	**0.548**	188K	**0.548**	188K	**0.548**	188K	185K
Average Drop	**0.233**		**-0.003**		**-0.003**		**Base Case**		

Table 10. Performance of MinUn-Posit on FastGRNN, ProtoNN, Bonsai and RNNPool models against computationally expensive bitwidth assignment mechanisms and the previous state-of-the-art. Accuracy is in percent, and size is in bytes. For classification-based models, the number of classes follow the dataset name. Average accuracy/MAP drop is calculated with respect to MinUn-Posit value.

Model & Dataset	Shiftry-Posit		Accuracy-Based		Size-Based		MinUn-Posit		
	Accuracy	RAM	Accuracy	RAM	Accuracy	RAM	Accuracy	RAM	RAM (OPT)
FastGRNN									
DSA-19	77.00	448	77.00	448	76.91	448	**77.13**	456	448
WAKEWORD-2	**98.40**	160	**98.40**	160	**98.40**	160	**98.40**	160	160
GOOGLE-12	92.50	941	92.57	941	92.54	941	**92.63**	900	900
GOOGLE-30	77.31	951	78.64	951	78.35	800	**78.65**	902	900
HAR-2	90.80	720	**91.18**	720	91.14	720	**91.18**	720	720
HAR-6	**91.35**	400	**91.35**	400	91.25	400	**91.35**	400	400
USPS-10	94.02	416	94.02	416	**94.17**	416	93.82	416	416
MNIST-10	96.68	768	96.68	768	**96.86**	768	96.83	768	768
INDUSTRIAL-72	86.56	640	**89.84**	640	**89.84**	640	**89.84**	640	640
Average Drop	**0.58**		**0.02**		**0.04**		**Base Case**		
ProtoNN									
CIFAR-2	76.30	79	**76.52**	79	68.82	51	76.47	79	79
CR-2	**73.07**	104	72.85	104	66.97	66	72.80	104	104
CR-62	32.12	136	**32.53**	136	31.44	136	32.43	136	135
CURET-61	53.10	134	**54.17**	134	53.10	134	**54.17**	134	133
LETTER-26	82.62	64	82.62	64	82.62	64	**82.70**	64	63
MNIST-2	93.93	40	93.93	40	94.67	40	**94.86**	40	33
MNIST-10	91.31	41	91.31	41	91.37	41	**91.62**	41	41
USPS-2	93.17	40	93.17	40	94.02	40	**94.07**	40	36
USPS-10	91.83	810	**92.83**	810	92.43	810	92.53	820	800
WARD-2	94.41	54	**94.98**	54	93.79	36	94.93	54	54
Average Drop	**0.47**		**0.17**		**1.73**		**Base Case**		
Bonsai									
CIFAR-2	75.90	40	75.67	40	**75.96**	40	75.90	40	40
CR-2	**74.97**	40	74.66	40	74.76	40	**74.97**	40	40
CR-62	**10.76**	268	10.50	268	**10.76**	268	**10.76**	268	248
CURET-61	45.12	268	45.69	268	45.47	268	**45.76**	268	256
LETTER-26	**65.28**	124	65.26	118	65.08	118	**65.28**	124	104
MNIST-2	95.88	20	95.88	20	**96.06**	20	96.03	20	20
MNIST-10	**93.49**	50	93.43	50	**93.49**	50	**93.49**	50	50
USPS-2	**94.92**	60	94.77	60	**94.92**	60	**94.92**	60	60
USPS-10	**91.83**	48	91.73	48	**91.83**	48	**91.83**	48	44
WARD-2	**95.18**	20	**95.18**	20	**95.18**	20	**95.18**	20	20
Average Drop	**0.08**		**0.14**		**0.06**		**Base Case**		
RNNPool									
FACE-A	0.583	235K	0.638	213K	0.638	213K	**0.640**	230K	230K
FACE-B	0.297	230K	0.297	230K	0.297	230K	**0.331**	230K	230K
FACE-C	**0.464**	230K	**0.464**	230K	**0.464**	230K	**0.464**	230K	230K
Average Drop	**0.030**		**0.012**		**0.012**		**Base Case**		

Table 11. Latency values of FastGRNN, ProtoNN, Bonsai and RNNPool models for Shiftry-Fixed and MinUn-Fixed generated codes. Latency values are in seconds. For classification-based models, the number of classes follow the dataset name.

Model & Dataset	Shiftry-Fixed Latency	MinUn-Fixed Latency
FastGRNN		
DSA-19	1.964	2.047
WAKEWORD-2	1.167	1.124
GOOGLE-12	2.645	2.718
GOOGLE-30	3.043	3.060
HAR-2	2.690	2.633
HAR-6	3.201	2.990
USPS-10	1.094	1.172
MNIST-10	0.901	0.903
INDUSTRIAL-72	0.092	0.092
ProtoNN		
CIFAR-2	0.003	0.004
CR-2	0.006	0.007
CR-62	0.008	0.007
CURET-61	0.007	0.008
LETTER-26	0.010	0.010
MNIST-2	0.004	0.004
MNIST-10	0.004	0.004
USPS-2	0.003	0.003
USPS-10	0.004	0.005
WARD-2	0.004	0.005
Bonsai		
CIFAR-2	0.002	0.002
CR-2	0.002	0.002
CR-62	0.010	0.010
CURET-61	0.013	0.015
LETTER-26	0.004	0.005
MNIST-2	0.002	0.003
MNIST-10	0.004	0.005
USPS-2	0.002	0.002
USPS-10	0.003	0.003
WARD-2	0.003	0.003
RNNPool		
FACE-A	16.661	16.864
FACE-B	7.880	7.942
FACE-C	9.440	9.140