MinUN: Accurate ML Inference on Microcontrollers

Shikhar Jaiswal†
Microsoft Research
India

Rahul Kranti Kiran Goli‡
Microsoft Research
India

Aayan Kumar§
Microsoft Research
India

Vivek Seshadri
Microsoft Research
India

Rahul Sharma
Microsoft Research
India

Abstract
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 a) be parametric in the number representation 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 to generate efficient code for ARM microcontrollers (e.g., Arduino Uno, Due and STM32H747) that outperforms the prior TinyML frameworks.

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

Keywords: TinyML, Memory Management, Programming Languages, Compilers, Number Representations, Embedded Devices

ACM Reference Format:
Shikhar Jaiswal, Rahul Kranti Kiran Goli, Aayan Kumar, Vivek Seshadri, and Rahul Sharma. 2023. MinUN: Accurate ML Inference on Microcontrollers. In Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES ’23), June 18, 2023, Orlando, FL, USA. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3589610.3596278

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. These frameworks, which rely on interpreters, are unsuitable for running ML on memory-constrained devices. For example, an Arduino Uno [10] has 2 KBs 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 TinyML1 space that aims to run ML on low-power embedded devices [21, 22, 24, 26, 29, 52–54, 58, 59, 74, 84]. As a motivating example, consider a low-power chip embedded in the brain that can detect the onset of seizures and warn the user [52]. Animal testing of brain chips is already under way by NeURALINK [66]. 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 to a few kilobytes (KBs). Although, the technical specifications of NeURALINK are unavailable in public domain, for reference, the microcontrollers in pacemakers for hearts have memories between 16 bytes and 10 KBs [45, 78]. 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 billions 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

---

1 https://www.tinyml.org/
have been specifically designed to provide good accuracy while requiring only thousands of parameters and KBs of memory [29, 52, 54, 74], 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 [32, 89–91] on an ARM Cortex-M class microcontroller with 256 KBs of RAM [5]. See Figure 1 for an example where an ML model has marked the heads in the image. The RNNPool [74] model provides state-of-the-art accuracy while consuming only 600 KBs 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 [14, 25, 34, 35, 41, 49, 57, 61–63, 67, 75, 92, 94]. 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 [23] 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 6.3).

In the TinyML space, Tensorflow-Lite (TFLite) [41] 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 [26] is more expressive and compiles arbitrary models to 16-bit fixed-point C++ code that runs on bare metal. Shiftry [53] 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 [15, 28, 46, 48, 65, 93]: TFLite’s Zero-Skew, Google’s BFloat16, NVIDIA’s TensorFloat-32, Microsoft’s MSFP, Tesla’s CFP8 and the posit [30, 31]. 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 [41] or fixed-point [26, 53]. 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 6). 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.

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. Although reusing memory for different variables has the flavor of register allocation [2], note that registers don't suffer from fragmentation, which is the primary problem here.

1.2 Our Contributions

We provide a framework, MinUn\(^4\), that makes significant advances on all three subproblems. MinUn is the first TinyML framework which is parametric over any arbitrary 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

\(^{2}\)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

\(^{4}\)Instructional repository for generating datasets and reproducing experiments is hosted at: https://github.com/ShikharJ/MinUn

Figure 1. Face detection example.
the representation-independent parts (Section 4.1). In contrast, the prior TinyML frameworks are extremely tied to the number representation that they work with, e.g., fixed-point for SeeDot and ShiftIt.

For the bitwidth assignment problem, we propose a novel exploration algorithm, HAUNTER (Section 4.1), which uses both model accuracy and tensor size to produce better assignments than the accuracy-based heuristic of ShiftIt, while being much more efficient and scalable (Section 4.2).

Finally, for RAM management, MinUn encodes the memory management problem to a bin-packing problem and solves it using Knuth’s Algorithm X [47] (Section 4.3), 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.

In particular, for the model in Figure 1, MinUn reduces the RAM consumption from 600 KBs to 180 KBs, thus enabling this model to run on devices with 256 KBs of RAM. Although MinUn was not designed for large models, even on an ImageNet-scale CNN [40], MinUn outperforms the prior TinyML frameworks (Section 6.7).

We 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 lower bitwidth. Hence, the “post training quantization” of MinUn is complementary to the quantization techniques used during training.

Next, we discuss related work (Section 2) followed by an overview of MinUn (Section 3). Section 4 contains technical details, Section 5 contains implementation details and Section 6 presents our empirical results. Additional background and a worked out example is provided in [43].

2 Related Work

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 [37, 76] or automatically with Network Architecture Search (NAS) [13, 79, 81] 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 [33] and spatial down-sampling [74] reduce peak memory usage and are complementary to MinUn.

There are various approaches for quantization in the ML literature. In hybrid-quantization [40, 41] 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 ShiftIt [53], prior papers on quantization use the trivial bitwidth assignment of mapping all tensors in large CNNs to low bitwidths [14, 16, 25, 34, 35, 38, 41, 49, 57, 60–63, 67, 72, 75, 92, 94]. Most of these approaches rely on re-training a low-bitwidth model, which is a computationally very expensive process. Thus, “post-training quantization” frameworks [26, 41, 53] like MinUn offer a distinct advantage.

MinUn is related to floating-point tuning algorithms like Precimonious [73]. However, their objective is completely orthogonal to the problem considered by MinUn. MinUn 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 MinUn.

Memory management in the space of embedded devices is currently performed via wasteful static allocation, fragmentation - prone dynamic allocation, memory pools with restricted applicability, and so on. Avisar et al. [6] use Integer Linear Programming (ILP) to generate optimal allocation schemes for a memory management problem that is different from ours and their allocations are suboptimal for the problem we consider.

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 [8, 70, 73, 77, 80, 95]. Converters from floating-point to fixed-point that have not been designed for ML include [7, 9, 11, 12, 17–19, 64, 68, 86].

3 Compiler Overview

We now provide a high-level overview of the MinUn compiler (Figure 2). MinUn expects as input a program written in the ShiftIt-DSL [53], a concise syntax for expressing operations and functions commonly used in ML models.

The DSL code for models can be written manually, or automatically obtained by exporting TensorFlow/PyTorch models to ONNX and using MinUn’s ONNX front-end (Section 5). 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 achieved by choosing an initial bitwidth assignment, generating the corresponding x86 C++ inference code.
for it, compiling it using GCC and executing the code for logging accuracy and memory usage, and using the logged information for making changes to the bitwidth assignment. HAUNTER ensures that user-provided memory constraints are always met during bitwidth exploration, and returns a suitable assignment which maximizes accuracy.

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

The final device-specific C++ code is generated using a codegen pass over the AST, which also replaces all the intermediate RAM tensors, with offsets-based access to scratch on the basis of our memory map. This final C++ code is then cross-compiled (using either ARM GNU Toolchain or Arduino IDE) and executed on the microcontroller of choice. Further, we exposit the technical details, and analyze the empirical benefits of MtnUn.

4 Technical Details
In this section, we explain HAUNTER, analyze its complexity, and describe our optimum memory management mechanism. A working example and detailed algorithms used by HAUNTER are provided in [43].

4.1 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 which can be used to make the exploration process more frugal, by choosing a soft limit < 1. We use $\rho$ to denote the bitwidth assignment, which maps each tensor to two possible bitwidths \{lowBitwidth, highBitwidth\}. Note that while we limit most discussions to 8-bit / 16-bit to simplify exposition and allow comparison, HAUNTER generalizes in a straightforward manner to exploration across various bitwidths (Section 6.8.2).

4.1.1 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 (since multiple $es$ options are supported for these bitwidths), and scales for individual tensors for fixed-point. For obtaining these parameters for posits, 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 [53], 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 fixed-point tensor.

4.1.2 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, we set each promotable tensor to highBitwidth simultaneously, and then compile and execute the code to obtain a “high-bitwidth value map”. 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”.

Based on these two maps, we attempt to create a “heat map” in the second phase, wherein, we calculate the absolute deviation in the representative floating-point values between the highBitwidth and lowBitwidth case for each tensor. Since keeping track of large tensors is infeasible, we simply sort the element-wise deviations in increasing order and take the 95th percentile deviation as the representative error deviation for each tensor. We define the promotability score as:

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

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. 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 4.2).

4.1.3 Promotion Algorithm (Stage III): The promotion algorithm (Algorithm 1) itself runs in three stages. The first

![Diagram](image)
We denote the amount of time required per code generation call as $T_{\text{codegen}}$, and treat it as the upper bound of the code generation and compilation time required for different bitwidth configurations. We follow this terminology for execution call latency (per example) as well, terming it $T_{\text{execution}}$.

For a model of $N$ tensors, supplied with a dataset of $D$ samples, SHIFTRY incurs a bitwidth exploration latency of:

$$O(N \times D \times T_{\text{execution}} + N \times T_{\text{codegen}})$$

which HAUNTER improves to (see [43] for details):

$$O(\log N \times D \times T_{\text{execution}} + N \times \log N \times T_{\text{codegen}})$$

Table 4 reports the empirical reduction in code generation and compilation time.

### 4.3 Optimum Memory Management

Once the bitwidth assignment has been made, MinUn must allocate tensors at physical memory addresses to keep the peak RAM usage low by maximizing RAM location reuse. 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 prone to fragmentation [53], as depicted in Figure 3. Here, deallocating variables $A$ and $C$ has fragmented the memory.

#### Delegating Memory Management to the Compiler:

Given an ML model with a fixed-sized input, tensor sizes and live ranges (range of instructions where a tensor is used and needs memory) can be known at compile time. A mapping from tensors to memory locations (or indices) on a contiguous array (scratch) can be computed during the code generation process itself. This reuses the memory for tensors that are no longer alive to allocate new tensors. For example, in Figure 3, one way to avoid fragmentation is for the compiler to map $A$ and $C$ to memory locations that are adjacent.

Computing such maps is an NP-Hard bin packing problem which can be addressed through suboptimal greedy heuristics.

- **Static and Dynamic Assignment methods** are unable to fit FastGRNN [54] and RNNPool models for the typical RAM budgets that are available with microcontrollers. Hence, we take the third approach, but unlike prior work [53] we do not use greedy heuristics. Instead, we approach the bin-packing problem as an exact cover problem.

#### Definition 1 (Exact Cover)

Let $P(X)$ denote the power set of a set $X$. Given a set $X$ and a set $S \subseteq P(X)$, an exact cover of $X$ is a set $S' \subseteq S$ such that:

```plaintext
Algorithm 1: HAUNTER Promotion Algorithm

Function PromotionAlgorithm(promotionOrder, memoryLimit, softLimitFactor):
    # Stage I: Explore from the lowest memory config.
    ρ ← {var → lowBitwidth}|var ∈ allVars
    overshootingVars ← PromoteWithinMemoryLimit(promotionOrder, memoryLimit, softLimitFactor)
    accuracy ← Compile&Execute(ρ)
    SaveAcc&Bitwidth(accuracy, ρ)

    # Stage II: Promote each overshooting tensor & explore.
    for promotable ∈ overshootingVars do
        ρ ← {var → lowBitwidth}|var ∈ allVars
        ρ ← {promotable → highBitwidth}
        memoryUsage ← GetMemoryUsage(ρ)
        if memoryUsage > memoryLimit × softLimitFactor
            continue
        PromoteWithinMemoryLimit(promotionOrder, memoryLimit, softLimitFactor)
        accuracy ← Compile&Execute(ρ)
        SaveAcc&Bitwidth(accuracy, ρ)

    # Stage III: Promote all overshooting tensors & explore.
    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
```
we can map a rectangle made of Algorithm X [47], with the following optimizations to speed up the exploration:

- We use the Dancing Links Algorithm as proposed in [47] 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.
- 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 down to the proof that Algorithm X will always find an optimum assignment if it exists, which is well-known. An acute reader might argue that the size of the generated C++ code itself might be another point of concern. However, microcontrollers store compiled binaries on Flash, which is typically plentiful.

5 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

Figure 3. 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.

Figure 4. 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 $C_{v_l}$ for a given $v_l$. 1)

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

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 $v_l$ active between instructions $i_{\text{start}}$ and $i_{\text{end}}$, and occupying $b_{l}$ bytes in memory, we can map a rectangle made of $b_{l} \times (i_{\text{end}} - i_{\text{start}})$ contiguous units on this canvas, occurring between $i_{\text{start}}$ and $i_{\text{start}}$.

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 $v_l$, we define a new set $C_{v_l} \subseteq \mathcal{P}(C)$ containing all subsets of unit elements in $C$, which can be potential memory maps to the tensor. Hence, each individual set in $C_{v_l}$ forms a contiguous block of memory spanning the live range of $v_l$, and has a height of at least $b_{l}$ units, as depicted in Figure 4.

Let $X = C \cup V$ and $S \subseteq \mathcal{P}(X)$ with the following constraint: \[\forall s_k \in S, \forall v_l \in V[\forall v_l \in s_k \Rightarrow s_k \setminus v_l \in C_{v_l}]\]

Given these sets, we solve the exact cover problem using an exponential-time, backtracking search algorithm called Algorithm X [47], with the following optimizations to speed up the exploration:

- We use the Dancing Links Algorithm as proposed in [47] 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.
- 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 $k = 32$,
of libraries in our experiments in order to obtain arithmetic routines for different number representations:

- **BFLOAT** [1] is a standalone C++ header-only sub-library implemented in the TensorFlow library, allowing conversion between **FLOAT32** and **BFLOAT16** types.
- **SOFTPOSIT** [56] provides posit arithmetic routines for simulating on x86-based platforms using C++. The library contains all functions expected in the Posit Standard [27]. 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** [42] 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 6.7), we downloaded the pre-trained ONNX model from the public ONNX Model Zoo [82], and implemented a Python front-end for automatically compiling ONNX models to the DSL (Section 3) 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. For **RNNPool** models, we use a coarsening parameter $k = 64$, and for remaining models, we use a coarsening parameter $k = 1$.

6 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 with low-precision 8-bit representations incurs unacceptable accuracy drops (Figure 5).
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 6).
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 1) and posit representations (Table 2).
4. **MinUn**-generated fixed-point code runs on real microcontrollers (Section 6.6).
5. For some models, using optimum memory management significantly lowers the RAM consumption compared to a standard first-fit heuristic [43].

6. Even on an ImageNet-scale model [40], **MinUn** outperforms the prior TinyML frameworks (Table 5).

6.1 Classification Models and Datasets

We evaluate on three state-of-the-art TinyML classification models developed for resource constrained devices, namely **ProtoNN** [29], **BONSAI** [52], and **FASTGRNN** [54]. These are suitable for running on Arduino class devices, which support a minimum of 2 KBS of SRAM and 40 KBS of Flash. For fairness of evaluation, we cover the entire suite of datasets evaluated by **SHIFTRY**. These include CIFAR [50], Character Recognition (CR) [20], USPS [39], Curet [83], Letter [36], Ward [87] and MNIST [55] for **ProtoNN** and **BONSAI**, and DSA [3], Google [85], HAR [4], MNIST [55], USPS [39], Industrial [53] and Wakeword [54] for **FASTGRNN**.

6.2 Localization Models and Datasets

**RNNPool** [74] 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 × 240 × 1 monochromatic QVGA images. The Face-C model is a direct replication of the **RNNPool-Face-M4** model introduced in [74]. These **RNNPool**-based models are designed for ARM Cortex-M class devices [44], which support a minimum of 256 KBS of SRAM and 1 MB of Flash. We train these models on the WIDER FACE [88] dataset and fine-tune them on SCUT-HEAD [71] dataset. For model accuracy, we evaluate the generated codes on 20% validation split (405 images) of SCUT-HEAD Part-B, and report the **Mean Average Precision (MAP)** scores.

6.3 Comparison With 8-Bit Baselines

We 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 5 shows that 8-bit representations have poor accuracy. In the case of 8-bit posits (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.
We compare the accuracy and RAM consumption of MinUn-Fixed against the golden 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. Figure 6 summarizes our experiments, where we observe significant reduction of $2.9 \times -5.1 \times$ in peak RAM consumption even compared to Fixed16 or BFloat16, while observing only negligible differences in accuracy compared to Float32.

6.4 Comparison With 16-Bit Baselines

We compare MinUn’s bitwidth assignment mechanism with the mechanism of Shiftry and other exclusively accuracy-based and exclusively size-based mechanisms in both posits and fixed-point.

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, while keeping the remaining tensors in higher 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 $\{1, 0.8, 0.9, 1.0, 1.1\}$ for fixed-point, and $\{0.8, 0.9, 1.0, 1.1\}$ for our posits experiments. In case of MinUn’s result, we report the RAM consumption values while employing Shiftry’s first-fit greedy heuristic, as well as the optimum memory management using Algorithm X. Table 1 summarizes 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. Section 6.6 shows that MinUn achieves these improvements over Shiftry with negligible difference in execution latency, and a significant improvement in compilation latency. Similarly, Table 2 summarizes our experiments for posits, where we compare Shiftry-Postit against MinUn-Postit. Here as well, we observe better
6.6 Inference Latency and Compilation Time

Note that the RAM usage and model accuracy are independent of the underlying hardware, and improvements in them average accuracy numbers across all models and baselines while strictly staying within the same RAM constraints. Of particular interest is fixed-point RNNPool where MinUn shows a 23.3% improvement in MAP score over Shiftry resulting in significant improvement in inference quality (see Figure 7 and Figure 8 for an example).

Table 3. Inference time of MinUn-Fixed on Arduino Uno board for FastGRNN, ProtoNN and Bonsai models. Latency values are in seconds, and are obtained by averaging over the test set. The number of classes follow the dataset name.

| Dataset        | Latency | Dataset        | ProtoNN Latency | Bonsai Latency |
|----------------|---------|----------------|-----------------|----------------|
| FastGRNN       | CIFAR-2 | 0.07           | 0.08            |
| DSA-19         | CR-2    | 0.13           | 0.08            |
| WAKEWORD-2     | CR-62   | 0.27           | 0.31            |
| GOOGLE-12      | CURET-61| 0.07           | 0.36            |
| GOOGLE-30      | LETTER-26| 0.20          | 0.07            |
| HAR-2          | MNIST-2 | 0.10           | 0.08            |
| HAR-6          | MNIST-10| 0.11           | 0.09            |
| USPS-10        | USPS-2  | 0.05           | 0.08            |
| MNIST-10       | USPS-10 | 0.11           | 0.04            |
| INDUSTRIAL-72  | WARD-2  | 0.11           | 0.10            |

6.7 Evaluation on ImageNet

Table 5 evaluates MinUn-Posit on SqueezeNet [40] - an ImageNet-scale CNN that matches the accuracy of AlexNet.
we evaluated but did not pursue because the gains they provided were small in practice.

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 [69] heuristic for memory management on MinUn-Posit, which was further reduced to 1.16 MBs on using our optimum memory management, offering an improvement of almost 20%. As evident from Table 5, we achieve up to 3.81x reduction in peak RAM consumption with minimal loss in accuracy, relative to the Float32 gold standard, while existing TinyML frameworks (TF Lite and Shiftry) fail to offer comparable reductions without leading to significant accuracy losses. In particular, MinUn has 1.9x reduction in RAM consumption over Shiftry while obtaining 6.5% higher accuracy.

6.8 Discussion

Our evaluation shows that MinUn significantly improves the inference quality and RAM usage over Shiftry without degrading inference latency or compilation time. In particular, Figure 7 and Figure 8 show qualitatively the difference between the output of MinUn and Shiftry on the image in Figure 1. These images show that improvements in MAP scores by MinUn translates to significantly better inference output. We discuss two approaches to improve MinUn that we evaluated but did not pursue because the gains they provided were small in practice.

6.8.1 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 (full details in [43]), 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 1.

6.8.2 Generalizing to More Bitwidth Options: Our proposed algorithm 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 $k^N$ 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. 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.

7 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.

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

[1] 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

[2] 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.

[3] Kerem Altun, Bilhur 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
[62] Julieta Martinez, Shobhit Zakhmi, Holger H. Hoos, and James J. Little. 2018. LSQ++: Lower runtime 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

[63] 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

[64] 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

[65] 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

[66] Elon Musk. 2019. An Integrated Brain-Machine Interface Platform With Thousands of Channels. J. Med. Internet Res. 21, 10 (30 Oct 2019), e16194. https://doi.org/10.2196/16194

[67] 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

[68] Anshuman Nayak, Malay Haldar, Alok N. Choudhary, and Prithviraj Varma, and Prateek Jain. 2020. RNNPool: Efficient Non-linear Pooling with Loop Perforation. In Proceedings of the 19th ACM SIGSOFT Symposium and the 15th 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

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

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

[71] Manik Varma and Andrew Zisserman. 2005. A statistical approach to texture classification from single images. International journal of computer vision 61, 2-5 (2005), 61–81. https://doi.org/10.1023/B:VISI.0000046589.39864.ee

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

[73] Cindy Rubio-Gonzalez, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonomous: 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/10.1145/2503212.2503296

[74] Oindrila Saha, Aditya Kasupati, 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.

[75] 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=rkxNAj9Ym

[76] 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.

[77] Eric Schikufza, 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/10.1145/2666356.2594392

[78] Abhishek A. Sharma. 2014. A Consolidated Review on Embedded MicroController for Face Maker Applications. In The 2014 International Conference on Embedded Systems and Applications.

[79] Albert Shaw, Daniel Hunter, Forrest Landallo, 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.

[80] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In Proceedings of the 13th ACM SIGSOFT Symposium and the 12th 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

[81] Xu Zhao, Xiaoqing Liang, Chaoyang Zhao, Ming Tang, and Jinqiao Tian. 2019. Focused Quantization for Sparse CNNs. In Proceedings of the International Conference on Biometrics (IJCB). IEEE, 1–9.

[82] Shou Yang, Ping Luo, Chen Changye Loy, and Xiaoou Tang. 2016. WIDER FACE: A Face Detection Benchmark. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[83] Young Joon Yoo, Dongyoon Han, and Sangdoo Yun. 2019. EXTD: Accurate ML Inference on Microcontrollers LCTES ’23, June 18, 2023, Orlando, FL, USA.
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

[93] 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

[94] 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

[95] 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

Received 2023-03-16; accepted 2023-04-21