Shiftry: RNN Inference in 2KB of RAM

AAYAN KUMAR, Microsoft Research, India
VIVEK SESHADRI, Microsoft Research, India
RAHUL SHARMA, Microsoft Research, India

Traditionally, IoT devices send collected sensor data to an intelligent cloud where machine learning (ML) inference happens. However, this course is rapidly changing and there is a recent trend to run ML on the edge IoT devices themselves. An intelligent edge is attractive because it saves network round trip (efficiency) and keeps user data at the source (privacy). However, the IoT devices are much more resource constrained than the cloud, which makes running ML on them challenging. Specifically, consider Arduino Uno, a commonly used board, that has 2KB of RAM and 32KB of read-only Flash memory. Although recent breakthroughs in ML have created novel recurrent neural network (RNN) models that provide good accuracy with KB-sized models, deploying them on tiny devices with such hard memory requirements has remained elusive.

We provide, Shiftry, an automatic compiler from high-level floating-point ML models to fixed-point C-programs with 8-bit and 16-bit integers, which have significantly lower memory requirements. For this conversion, Shiftry uses a data-driven float-to-fixed procedure and a RAM management mechanism. These techniques enable us to provide first empirical evaluation of RNNs running on tiny edge devices. On simpler ML models that prior work could handle, Shiftry-generated code has lower latency and higher accuracy.

CCS Concepts: • **Software and its engineering → Compilers**; *Domain specific languages*; • **Hardware → On-chip resource management**.

1 INTRODUCTION

Machine learning (ML) algorithms are increasingly being deployed to build smart systems that deploy sensor devices (IoT devices) to collect data from the environment and process the data using powerful ML algorithms. Recently, there is a growing number of applications that require the ML inference to be run directly on the IoT device for reasons including energy efficiency and privacy. Examples of such applications are anomaly detection [Chakraborty et al. 2018], accessibility devices [Patil et al. 2018], sports training [Wang et al. 2018], etc. However, today, there is a mismatch between IoT devices and ML algorithms. On one hand, IoT devices have very low compute and memory resources. For example, Arduino Uno, a widely-used board by makers, has a 16 MHz processor with no hardware support for floating-point, 2 KB of read/write RAM, and 32 KB of read-only Flash memory. On the other hand, ML practitioners typically generate models in floating-point arithmetic with the goal of maximizing accuracy, often with no regard for the amount of memory available in the target device.

While there are libraries that can emulate floating-point in software, prior works (e.g., SeeDot [Gopinath et al. 2019a], TensorFlow-Lite [Jacob et al. 2017], etc.) have also proposed tools that can automatically convert floating-point models to integer models. These tools eliminate the overhead of software emulation of floating-point, thereby significantly reducing the latency of executing the prediction algorithms. However, these tools assume that the generated integer models will fit in the memory of the target device. Unfortunately, unlike compute constraints wherein a slow micro-controller will just take a long time to run a program, memory constraints are hard. A model that does not fit in the memory resources of the target device cannot be run on the device.

Our goal in this work is to build a compiler that compiles floating-point ML models (targeted for IoT devices) to code that can actually *run on the target device* with as high performance and

Authors' addresses: Aayan Kumar, Microsoft Research, India, t-aak@microsoft.com; Vivek Seshadri, Microsoft Research, India, visesha@microsoft.com; Rahul Sharma, Microsoft Research, India, rahsha@microsoft.com.

as little loss in accuracy as possible. IoT devices typically have two types of memory: 1) a read-only Flash that contains static data like the ML model parameters, and 2) a read/write RAM that contains all mutable states during program execution. Each of these two memories pose a different challenge. To successfully execute an ML inference algorithm on the target device, first, the ML model parameters should fit in the Flash memory. While ideally, we would like all variables to be 8-bit integers (the smallest unit of data supported by most IoT devices), we have observed that this choice is disastrous for accuracy in practice. Second, most IoT devices have limited or no support for dynamic memory allocation. ML inference algorithms typically maintain many intermediate variables with overlapping scopes throughout the program. Therefore, when the RAM available in the target device is not big enough to store all the intermediate variables, the compiler must be aware of the available RAM and manage memory in target code intelligently.

We propose Shiftry[1], a compiler that takes a floating-point ML model as input and generates fixed-point code for a target device with given memory constraints. Along with the model, Shiftry assumes that a small amount of validation set for the model is available to compare the accuracy of different programs. It handles the Flash constraint and RAM constraint using two techniques. First, generating fixed-point code requires a compiler to identify the *bitwidth* and *scale* for each variable in the program (Section 2.2). Shiftry starts by assigning 16-bits to each variable. It uses *data-driven scaling* (Section 6.1) to determine the scale for each variable. Then, Shiftry iteratively *demotes* different 16-bit variables to 8-bits by using the validation set to estimate the loss in accuracy for such demotions. At each iteration, Shiftry reassigns the scale of the demoted variable based on the new bitwidth and the profiled data.

Second, today, there are two possible ways to handle the RAM constraint. One is to allocate all temporary variables on the stack (supported by embedded C compilers). The other approach is to use dynamic memory allocation. However, both these solutions are insufficient in our scenario. On one hand, allocating variables on the stack can result in many variables that are no longer in use consuming unnecessary memory. On the other hand, dynamic memory allocation can result in severe fragmentation and may run out of contiguous free space to assign to new variables. In particular, we have observed that allocation based on malloc/free fails to run any of our benchmarks. Shiftry works around this problem by exploiting our observation that all the variable sizes and shapes are known at compile time. Based on this, Shiftry *statically* simulates dynamic memory allocation of variables and allocates them in blocks to provide contiguous free space for future variables. Even in the worst case, when free space may get fragmented, Shiftry inserts code for appropriately migrating variables, which will allow it to allocate memory for new variables.

With these techniques, the only programs that Shiftry cannot compile to a target device are those in which the model would not fit in the Flash even when using 8-bits for all variables or programs in which there is at least one program point where the live variables require more memory than the available RAM. Typical CNNs for computer vision, like AlexNet, ResNet, LeNet, VGGNet, etc., fall in this category of programs when considering IoT devices and are beyond the scope of this paper.

For our evaluations, we study two types of models. The first is the class of powerful models called the *Recurrent Neural Network* (RNNs). RNNs are sequence-to-sequence learning models and are a natural fit for IoT-based applications where the input data is often a time-series. We show the first evaluation of running state-of-the-art RNNs on an Arduino Uno, a task that has been out of reach for prior work. Second, we compare the performance of Shiftry for state-of-the-art variants of simpler models like decision trees [Kumar et al. 2017] and nearest neighbors [Gupta

[1]Implementation hosted at https://github.com/aayan636/shiftry

et al. 2017]. For these models, while prior work can generate fixed-point code that can run on Uno, Shiftry-generated code is *both faster and has better accuracy*.

Because of our focus on tiny IoT devices like the Uno, one might wonder, if using IoT devices with more memory makes Shiftry moot. Although IoT devices with more memory are becoming cheaper, the ML models are becoming larger as well. As a result, the problem of compressing ML models to fit into the memory constraints of the target device, the problem which Shiftry addresses, is here to stay. We demonstrate that Shiftry is applicable in settings that employ more resourceful devices as well by squeezing a complex face detection model on an ARM cortex M4 class device. Nonetheless, we mainly focus on the Uno, as more RAM generally implies a higher power consumption. Hence, applications where minimizing power consumption is the top priority prefer devices with lesser RAM.

The rest of the paper is organized as follows: After discussing some preliminaries in Section 2, we show the execution of Shiftry on a simple example in Section 3. We provide a brief description of the architecture of Shiftry compiler in Section 4. Next, we show the input language of Shiftry (Section 5), the use of data-driven scaling to generate precise 16-bit code (Section 6.1), and demotion of variables to 8-bit (Section 6.2). We discuss our memory management scheme to improve RAM usage in Section 7. Our evaluation in Section 8 shows that Shiftry generated code has better accuracy and latency compared to the code generated by prior work. We also show that Shiftry enables the first evaluation of RNNs executing on tiny microcontrollers in Section 8. Finally, Section 9 discusses related work and Section 10 concludes.

2 PRELIMINARIES

In this section, we discuss the required terminology from machine learning and standard fixed-point arithmetic operations.

2.1 ML Preliminaries

An ML classifier takes a vector of Real-valued features (X) as input and returns a class label (l). For example, we can design a classifier to take an image as an input and return a label that says if the image contains a cat. To perform the classification task, the ML model consists of a set of parameters (W). The classifier is associated with a training algorithm, a training dataset of inputs, and labels that are used to learn the parameters W using supervised learning. ML models also typically use a validation dataset for hyperparameter tuning. A simple linear classifier is of the form $l = W \times X > 0$. In this paper, we focus our attention on ML models that are specifically targeted to run on IoT devices with small amounts of memory.

The standard way to measure the performance of an ML model is its *classification accuracy* on a testing dataset (that is separate from the training dataset). As ML algorithms are expressed over Real numbers, a floating-point implementation of the model is typically considered as the benchmark for accuracy. The effectiveness of any approximation of the ML model (e.g., using fixed-point values instead of floating-point values) is judged by how well it performs compared to the floating-point model. For instance, a fixed-point model that achieves classification accuracy within 1% of the best performing floating-point model may be deemed good enough.

2.2 Fixed-Point Preliminaries

In fixed-point arithmetic, a real number r is stored as a b-bit integer $\lfloor r \times 2^s \rfloor_b$. This representation is parameterized by two values, s and b. The *bitwidth* b denotes that this value occupies b bits in memory. The parameter s is called the *scale* of the number, and determines the number of mantissa bits. For example, if $r = 5.697$, $b = 16$, and $s = 12$ then

$$5.697 = 5.697 \times 2^{12}/2^{12} \approx \lfloor 5.697 \times 2^{12} \rfloor_{16}/2^{12} = \lfloor 23334.912 \rfloor_{16}/2^{12} = 23334_{16}/2^{12}$$

Here, 23334 is the integer stored in a 16 bits wide block of memory with the scale 12. This integer value is interpreted as $23334/2^{12} \approx 5.6968$ which is a close approximation of the actual value. The same value when represented using a lower scale, say 6, results in the integer 364.

$$5.697 = 5.697 \times 2^{6}/2^{6} \approx \lfloor 5.697 \times 2^{6} \rfloor_{16}/2^{6} = \lfloor 364.608 \rfloor_{16}/2^{6} = 364_{16}/2^{6}$$

For a given bitwidth and a real number, a higher scale results in a more precise value as long as there is no overflow. In fact, in the example above, 12 is the best scale for the given value and bitwidth. Using a scale higher than 12 (say 13) will result in an overflow, as shown below.

$$5.697 = 5.697 \times 2^{13}/2^{13} \approx \lfloor 5.697 \times 2^{13} \rfloor_{16}/2^{13} = \lfloor 46669.824 \rfloor_{16}/2^{13} = -18867_{16}/2^{13}$$

Here, due to limited range of 16-bit integers, there is an overflow, due to which the end result, if parsed in fixed-point arithmetic, gives -2.303, which is garbage. Hence, we need to determine the optimum scale for each variable in the program so that we have the best precision and avoid overflows.

In Shiftry, we often *demote* variables to reduce the memory footprint, i.e., reduce the bitwidth of numbers, e.g, from 16-bit to 8-bit. Similarly, increasing the number of bits assigned to a variable is called *promotion*. Note that if a variable is demoted, it's scale would need to be altered too. Consider the same number as above: 5.697. If we use the same scale as in the 16-bit case, 12, the resulting integer, 23334, would overflow an 8-bit integer. It turns out the best scale for an 8-bit integer, so that the resulting integer doesn't cross 127 (*INT_MAX* for 8-bit integers) is 4:

$$5.697 = 5.697 \times 2^{4}/2^{4} \approx \lfloor 5.697 \times 2^{4} \rfloor_{8}/2^{4} = \lfloor 91.152 \rfloor_{8}/2^{4} = 91_{8}/2^{4}$$

The resulting integer representation, 91 with scale 4, evaluates in fixed-point arithmetic to 5.6875, a slightly worse approximation than the 16-bit representation.

3 WORKING EXAMPLE

In this section, we use a toy example of a linear model to both motivate the problem we address in the paper and the end-to-end working of our proposed solution, Shiftry. Non-linear activation functions are also supported by Shiftry and will be described in Section 5. Pseudocode 1 shows the example program in Shiftry DSL. The program consists of 5 read-only parameters $W_1 \in \mathbb{R}^{2\times 2}$, $B_1 \in \mathbb{R}^{2\times 1}$, $W_2 \in \mathbb{R}^{2\times 1}$, $B_2 \in \mathbb{R}$, and $X \in \mathbb{R}^{2\times 1}$ and returns a real number. For this example, our goal is to run this program on a target device with 14 bytes of Flash memory available for parameters and 8 bytes of RAM for intermediate computations.

Pseudocode 1: Example in Shiftry DSL

$$W_1 := \begin{pmatrix} 0.0421 & 0.1948 \\ 1.021 & -0.827 \end{pmatrix} \; B_1 := \begin{pmatrix} -0.032 \\ 0.619 \end{pmatrix} \qquad X := \begin{pmatrix} 2.391 \\ -3.583 \end{pmatrix}$$

$$W_2 := \begin{pmatrix} -0.402 & -1.013 \end{pmatrix} \; B_2 := \begin{pmatrix} 0.737 \end{pmatrix}$$

return $W_2 \times (W_1 \times X + B_1) + B_2$

In the following discussion, we will use numerical accuracy as a metric to compare different programs. We will measure numerical accuracy of a program as the difference in output of the program and the output of the floating-point implementation of the model. We refer to this difference as *precision loss*.

Pseudocode 2: Homogenous fixed-point code generated by Shiftry.

$$\texttt{int}_{16}[2][2]W_1 := \begin{pmatrix} 689_{16} & 3191_{16} \\ 16728_{16} & -13549_{16} \end{pmatrix}$$
$$\texttt{int}_{16}[2][1]B_1 := \begin{pmatrix} -1048_{16} \\ 20283_{16} \end{pmatrix}$$
$$\texttt{int}_{16}[2][1]X := \begin{pmatrix} 19587_{16} \\ -29351_{16} \end{pmatrix}$$
$$\texttt{int}_{16}[1][2]W_2 := \begin{pmatrix} -6586_{16} & -16596_{16} \end{pmatrix}$$
$$\texttt{int}_{16}[1][1]B_2 := \begin{pmatrix} 24150_{16} \end{pmatrix}$$

$\texttt{int}_{16}[2][1]\ t_1;\ \texttt{int}_{16}[2][1]\ t_2;$
$\texttt{int}_{16}[1][1]\ t_3;\ \texttt{int}_{16}[1][1]\ t_4;$

$t_1 = (W_1 \times_{\texttt{int}_{32}} X)/2^{15}$
$t_2 = t_1 +_{\texttt{int}_{16}} (B_1/2^3)$
$t_3 = (W_2 \times_{\texttt{int}_{32}} t_2)/2^{14}$
$t_4 = t_3 +_{\texttt{int}_{16}} (B_2/2^3)$
return t_4

Pseudocode 3: Heterogenous Fixed-point code: B_1 uses 8 bits, rest use 16 bits

$$\texttt{int}_{16}[2][2]W_1 := \begin{pmatrix} 689_{16} & 3191_{16} \\ 16728_{16} & -13549_{16} \end{pmatrix}$$
$$\texttt{int}_{8}[2][1]B_1 := \begin{pmatrix} -4_{8} \\ 79_{8} \end{pmatrix}$$
$$\texttt{int}_{16}[2][1]X := \begin{pmatrix} 19587_{16} \\ -29351_{16} \end{pmatrix}$$
$$\texttt{int}_{16}[1][2]W_2 := \begin{pmatrix} -6586_{16} & -16596_{16} \end{pmatrix}$$
$$\texttt{int}_{16}[1][1]B_2 := \begin{pmatrix} 24150_{16} \end{pmatrix}$$

$\texttt{int}_{16}[2][1]\ t_1;\ \texttt{int}_{16}[2][1]\ t_2;$
$\texttt{int}_{16}[1][1]\ t_3;\ \texttt{int}_{16}[1][1]\ t_4;$

$t_1 = (W_1 \times_{\texttt{int}_{32}} X)/2^{15}$
$t_2 = (t_1/2^5) +_{\texttt{int}_{16}} B_1$
$t_3 = (W_2 \times_{\texttt{int}_{32}} t_2)/2^{9}$
$t_4 = t_3 +_{\texttt{int}_{16}} (B_2/2^3)$
return t_4

In our example, the result of the floating-point code is −5.11167404. The floating-point model consumes 44 bytes of Flash. A naive program implementing the model in floating-point requires 24 bytes of working memory. Both of these requirements exceed the constraints of our target device.

Prior work [Gopinath et al. 2019a] has proposed a compiler that can automatically convert floating-point code to fixed-point code with a given bitwidth. Even this solution does not work for this example. On one hand, 16-bit fixed-point code (shown in Pseudocode 2) has a low precision loss (When run, it outputs -20935, which when translated to a floating-point number gives −5.11108398, an error of 0.0006). However, it still consumes 22 bytes of Flash and 12 bytes of RAM which does not fit our target device. On the other hand, although the 8-bit fixed point code meets the Flash constraint, it has high precision loss (0.2366, refer to Table 2). Our goal is to generate the code that has the least precision loss while meeting the memory constraints.

To reduce the memory usage, Shiftry demotes a subset of variables to use 8-bit integers instead of 16-bits. Shiftry supports two strategies. In the first strategy, it demotes the minimum number of variables that are required to fit the model on the device, thus maximizing accuracy. In the second strategy, Shiftry demotes the maximum number of variables while ensuring that the precision loss stays below a user-provided threshold (say 0.1). The latter usually leads to better latency as operations on demoted variables are cheaper. Shiftry identifies these demote-able variables as follows. First, for each variable, Shiftry generates a program with that variable demoted. For example, for the variable B_1, Shiftry generates the code in Pseudocode 3 which uses 8-bits for B_1. When demoting a variable, Shiftry automatically identifies both the initialization for the variable and the scale for the variable under the new bitwidth.

Shiftry records the precision loss of demoting each variable in the program. Table 1 shows this data for our example program. Shiftry then orders the variables in the increasing order of the corresponding precision loss. In our example, the order is B_2, B_1, X, W_1, W_2, t_3, t_4, t_1, t_2.

Var	t_4	Precision Loss
W_1	-5.0456	0.0660
W_2	-5.0402	0.0713
B_1	-5.1022	0.0093
B_2	-5.1171	0.0055
X	-5.0788	0.0328
t_1	-5.0012	0.1104
t_2	-5.0012	0.1104
t_3	-5.1875	0.0758
t_4	-5.1875	0.0758

Table 1. Partial Demote

Var	t_4	Precision Loss
B_2	-5.1171	0.0055
B_2, B_1	-5.1093	0.0022
B_2, B_1, X	-5.0781	0.0335
B_2, B_1, X, W_1	-5.0156	0.0960
B_2, B_1, X, W_1, W_2	-4.9453	0.1663
$B_2, B_1, X, W_1, W_2, t_3$	-5.0000	0.1116
$B_2, B_1, X, W_1, W_2, t_3, t_4$	-5.0000	0.1116
$B_2, B_1, X, W_1, W_2, t_3, t_4, t_1$	-4.8750	0.2366
$B_2, B_1, X, W_1, W_2, t_3, t_4, t_1, t_2$	-4.8750	0.2366

Table 2. Cumulative Demote

Finally, Shiftry demotes variables cumulatively in the order computed above. Specifically, Shiftry generates a program where B_2 is demoted, then a program where both B_2 and B_1 are demoted, and so on. Shiftry stops demoting variables when the precision loss exceeds the user-specified limit. Table 2 shows the precision loss of cumulatively demoting variables in our example. In this example, for the user-specified loss of 0.1, Shiftry chooses the program that demotes the variables B_2, B_1, X, and W_1. Pseudocode 4 shows the corresponding program. It meets the Flash constraint as the read-only parameters W_1, W_2, B_1, B_2, and X fit within 14 bytes.

A naive implementation of the chosen program consumes 12 bytes of working memory (for t_1, t_2, t_3, t_4), which still does not fit in the RAM of the target device. Unfortunately, currently available

Pseudocode 4: Heterogenous Fixed-point code: only B_2, B_1, X, W_1 use 8 bits, rest use 16 bits

$$
\begin{aligned}
&\mathtt{int}_8[2][2]W_1 := \begin{pmatrix} 2_8 & 12_8 \\ 65_8 & -52_8 \end{pmatrix}\\
&\mathtt{int}_8[2][1]B_1 := \begin{pmatrix} -4_8 \\ 79_8 \end{pmatrix}\\
&\mathtt{int}_8[2][1]X := \begin{pmatrix} 76_8 \\ -114_8 \end{pmatrix}\\
&\mathtt{int}_{16}[1][2]W_2 := \begin{pmatrix} -6586_{16} & -16596_{16} \end{pmatrix}\\
&\mathtt{int}_8[1][1]B_2 := \begin{pmatrix} 94_8 \end{pmatrix}\\
\\
&\mathtt{int}_{16}[2][1]\ t_1;\ \mathtt{int}_{16}[2][1]\ t_2;\\
&\mathtt{int}_{16}[1][1]\ t_3;\ \mathtt{int}_{16}[1][1]\ t_4;\\
\\
&t_1 = (W_1 \times_{\mathtt{int}_{16}} X)\\
&t_2 = (t_1/2^4) +_{\mathtt{int}_{16}} B_1\\
&t_3 = (W_2 \times_{\mathtt{int}_{32}} t_2)/2^9\\
&t_4 = (t_3/2^5) +_{\mathtt{int}_{16}} B_2\\
&\textbf{return}\ t_4
\end{aligned}
$$

Pseudocode 5: Heterogenous fixed-point code generated by Shiftry

$$
\begin{aligned}
&\mathtt{int}_8[2][2]W_1 := \begin{pmatrix} 2_8 & 12_8 \\ 65_8 & -52_8 \end{pmatrix}\\
&\mathtt{int}_8[2][1]B_1 := \begin{pmatrix} -4_8 \\ 79_8 \end{pmatrix}\\
&\mathtt{int}_8[2][1]X := \begin{pmatrix} 76_8 \\ -114_8 \end{pmatrix}\\
&\mathtt{int}_{16}[1][2]W_2 := \begin{pmatrix} -6586_{16} & -16596_{16} \end{pmatrix}\\
&\mathtt{int}_8[1][1]B_2 := \begin{pmatrix} 94_8 \end{pmatrix}\\
&\mathtt{int}_8\ mem_{0:8};\\
\\
&\begin{pmatrix} mem_{0:2} \\ mem_{2:4} \end{pmatrix} = (W_1 \times_{\mathtt{int}_{16}} X)\\
&\begin{pmatrix} mem_{4:6} \\ mem_{6:8} \end{pmatrix} = \left(\begin{pmatrix} mem_{0:2} \\ mem_{2:4} \end{pmatrix}/2^4\right) +_{\mathtt{int}_{16}} B_1\\
&\begin{pmatrix} mem_{0:2} \end{pmatrix} = \left(W_2 \times_{\mathtt{int}_{32}} \begin{pmatrix} mem_{4:6} \\ mem_{6:8} \end{pmatrix}\right)/2^9\\
&\begin{pmatrix} mem_{2:4} \end{pmatrix} = \left(\begin{pmatrix} mem_{0:2} \end{pmatrix}/2^5\right) +_{\mathtt{int}_{16}} B_2\\
&\textbf{return}\ \begin{pmatrix} mem_{1:2} \end{pmatrix}
\end{aligned}
$$

embedded compilers fall in this category. As mentioned in the introduction, dynamic memory management is both costly and results in fragmentation of free space.

Shiftry exploits two observations to mitigate this problem. First, as is the case with many programs, variables in the program are live only for a subset of instructions in the program with some variables having overlapping lifetimes. Second, for a program in Shiftry DSL, both the live range *and* the size of each variable is statically known at compile time. Table 3 shows the size and live range of each of the variables in working memory.

Var	Size	Live Range	Var	Size	Live Range
t_1	$(2 \times 1) \times 16$ bits = 4 bytes	1-2	t_3	$(1 \times 1) \times 16$ bits = 2 bytes	3-4
t_2	$(2 \times 1) \times 16$ bits = 4 bytes	2-3	t_4	$(1 \times 1) \times 16$ bits = 2 bytes	4-5

Table 3. Temporary Variable Details

From the live ranges in Table 3, Shiftry recognizes that variables t_3 and t_4 can fit into the memory block originally reserved for variable t_1. Shiftry views available memory as an array of values and determines the appropriate offsets into the array for each variable such that no two variables with overlapping live ranges conflict in memory. Pseudocode 5 shows this memory-optimized code for our example. This program consumes 8 bytes of RAM that fits in the target device.

4 OVERVIEW

In this section, we provide an overview of Shiftry and provide details in the subsequent sections. The input code to Shiftry is a program written using the source language, Shiftry-DSL (Figure 1), a high level language that provides compact syntax for operations that are commonly used in ML models (Appendix B). These include arithmetic operations over matrices of Reals. The Shiftry compiler first typechecks this program (Figure 3) and bugs like multiplying or adding matrices with incompatible dimensions are caught at compile time. The Shiftry compiler then compiles the input program, using the rules described in Figure 5, to a program in the target language (Figure 2). As opposed to the source language, the target language of Shiftry only supports integers and arrays over integers. A program in the target language is essentially a `main` procedure that makes calls to Shiftry's library functions (Library 7, 8 and 9) with the appropriate arguments. To reduce the RAM usage, Shiftry employs a memory management mechanism (Section 7) that replaces all intermediate variables with accesses to a single global array. This AST is then converted to C++ using a codegen pass [Aho et al. 2006]. Finally, the C++ program is compiled by the Arduino IDE [Banzi and Shiloh 2014] to assembly that can be run on an Arduino Uno for latency measurements.

For compiling a floating-point source program to a fixed-point target program, Shiftry crucially relies on two environments: σ, a map from variables to their scales, and β, a map from variables to their bitwidths. Shiftry determines these maps via exploration (Section 6). Shiftry assigns scales using runtime data (Section 6.1) and these scales are fine tuned to accommodate demotions in bitwidth (Section 6.2). In this process, Shiftry generates many fixed-point programs, evaluates their accuracy, and outputs (if possible) a program that meets the user-provided memory constraints. For measuring accuracy, Shiftry uses an x86-codegen and runs the fixed-point programs on commodity hardware. Although the exploration is embarrassingly parallel, it still constitutes the bulk of the compilation time. Moreover, the compilation time grows with the size of datasets and for very large datasets subsampling might be needed to keep the compilation times tractable.

For a simple example, consider the following source program and environments:

$$x := 2.25; y := 1.50; \mathbb{R}\ z; z = x \times y, \sigma = [x \mapsto 2, y \mapsto 1, z \mapsto 3], \beta = [x \mapsto 8, y \mapsto 8, z \mapsto 16]$$

Here, Shiftry outputs the following (simplified) C++ fragment as fixed-point code:

```
int8_t x = 9;                  // 9 = 2.25 * 4
int8_t y = 3;                  // 3 = 1.50 * 2
int16_t z = int16_t(x)*int16_t(y);
```

In the subsequent sections, we describe this compilation process formally (Section 5), inference of σ and β (Section 6), and our memory management mechanism (Section 7).

5 FORMAL DEVELOPMENT

The Shiftry compiler takes an ML model expressed in the Shiftry DSL as input. To keep the presentation simple, we focus only on the core constructs of the Shiftry DSL in Figure 1. We provide a complete list of operators supported by Shiftry in Appendix B. The Shiftry DSL, $\mathcal{L}$, is a high level imperative language that helps represent ML models compactly by providing arithmetic operators over matrices. See Pseudocode 19 for the implementation of an RNN in only 12 lines of Shiftry DSL code. The target language of the Shiftry compiler, $\mathcal{T}$ in Figure 2, has been designed to simplify code-generation for embedded devices. We present the type system for $\mathcal{L}$ in Figure 4 and the rules to compile programs in $\mathcal{L}$ to $\mathcal{T}$ in Figure 5. While a program in $\mathcal{L}$ expresses a mathematical computation over Reals, a program in $\mathcal{T}$ is a computation over fixed-point integers with finite bitwidths. We also describe our approaches to compute the transcendental functions occurring in ML models by fixed-point integers in Section 5.4.

5.1 Syntax

Figure 1 describes the syntax of the source language $\mathcal{L}$ of Shiftry. The input program is a sequence of declarations ($\tau\ x$) and initializations with values v ($x := v$), followed by a sequence of statements s, and ending with a return of a binary classification label. A statement can be either an assignment with a computational expression e (e.g., matrix multiplication, scalar exponentiation, etc.) or a for-loop. We disallow compound expressions for brevity of presentation.

$$
\begin{array}{rcl}
P & ::= & x\ := v;\ P \mid \tau\ x;\ P \mid s;\ \textbf{return}\ x\ >\ 0 \\
s & ::= & s_1; s_2 \mid x\ =\ e \mid \textbf{for}\ i = [0:n]\ \textbf{do}\ s \\
e & ::= & v \mid x \mid y[z] \mid y \times z \mid y + z \mid f(y) \\
v & ::= & n \mid r \mid [v_1, v_2, ..., v_n] \\
f & ::= & \mathtt{exp} \mid \mathtt{tanh} \mid \mathtt{sigmoid}
\end{array}
$$

Fig. 1. Syntax of the core source language $\mathcal{L}$

$$
\begin{array}{rcl}
P' & ::= & \tau' x := v';\ P' \mid \tau' x;\ P' \mid S';\ \textbf{return}\ x\ >\ 0 \\
s' & ::= & s'_1; s'_2 \mid x\ =\ e' \mid \textbf{for}\ i = [0:n]\ \textbf{do}\ s' \\
e' & ::= & v' \mid y[z] \mid y \times_{\tau'} z \mid \Psi_{\tau'}(e', n) \mid \\
 & & e'_1 +_{\tau'} e'_2 \mid f'(e') \\
v' & ::= & n_b \mid [v'_1, v'_2, ..., v'_n] \\
f' & ::= & \mathtt{Exp}^{\mathtt{Q}}{}_b \mid \mathtt{Tanh}^{\mathtt{Q}}{}_b \mid \mathtt{Sigmoid}^{\mathtt{Q}}{}_b \\
\tau' & ::= & \mathtt{int}_b \mid \mathtt{int}_b\mathtt{[n]} \mid \mathtt{int}_b\mathtt{[n_1][n_2]}
\end{array}
$$

Fig. 2. Syntax of the target language $\mathcal{T}$

5.2 Type system

$$
\tau \quad ::= \quad \mathbb{R} \mid \mathbb{Z} \mid \mathbb{R}[n] \mid \mathbb{R}[n_1][n_2]
$$

Fig. 3. Possible types in the source language of Shiftry

Figure 3 describes the possible types in $\mathcal{L}$. A variable can be a mathematical Integer ($\mathbb{Z}$), or a Real ($\mathbb{R}$), or a 1- or 2- dimensional matrix of Reals ($\mathbb{R}[n_1]$ or $\mathbb{R}[n_1][n_2]$). The type system is described in Figure 4. Here, we have two types of judgments, for statements and for expressions. The statement judgment $\Gamma_1 \vdash_s s : \tau,\ \Gamma_2$ is read as: "under the typing environment Γ_1, the statement s is well typed

$$\frac{x \notin domain(\Gamma)}{\Gamma \vdash_s \tau\ x : \tau,\ \Gamma[x \mapsto \tau]}\ T-Decl \qquad \frac{\Gamma \vdash_e x : \mathbb{R}}{\Gamma \vdash_s \textbf{return}\ x > 0 : \mathbb{Z},\ \Gamma}\ T-Return$$

$$\frac{\Gamma \vdash_e v : \tau \quad x \notin domain(\Gamma)}{\Gamma \vdash_s x := v : \tau,\ \Gamma[x \mapsto \tau]}\ T-Init \qquad \frac{\Gamma \vdash_e e : \tau \quad \Gamma \vdash_e x : \tau}{\Gamma \vdash_s x = e : \tau,\ \Gamma}\ T-Assn$$

$$\frac{i \notin \Gamma_1 \quad \Gamma_1[i \mapsto \mathbb{Z}] \vdash_s s : \tau,\ \Gamma_2}{\Gamma_1 \vdash_s \textbf{for}\ i\ =\ [0, n]\ \textbf{do}\ s : \tau,\ \Gamma_2 \setminus \{i\}}\ T-Loop \qquad \frac{\Gamma \vdash_s s_1 : \tau_1,\ \Gamma_1 \quad \Gamma_1 \vdash_s s_2 : \tau_2,\ \Gamma_2}{\Gamma \vdash_s s_1; s_2 : \tau_2,\ \Gamma_2}\ T-Seq$$

$$\frac{}{\vdash_e r : \mathbb{R}}\ T-Real \qquad \frac{}{\vdash_e n : \mathbb{Z}}\ T-Int \qquad \frac{x \in \Gamma}{\Gamma \vdash_e x : \Gamma(x)}\ T-Var \qquad \frac{\Gamma \vdash_e x : \mathbb{R}}{\Gamma \vdash_e f(x) : \mathbb{R}}\ T-Exp$$

$$\frac{\Gamma \vdash_e v_1 : \mathbb{R} \ldots \Gamma \vdash_e v_{n_1} : \mathbb{R}}{\Gamma \vdash_e [v_1, \ldots v_{n_1}] : \mathbb{R}[n_1]}\ T-Arr1D \qquad \frac{\Gamma \vdash_e v_1 : \mathbb{R}[n_2]\ \ldots\ \Gamma \vdash_e v_{n_1} : \mathbb{R}[n_2]}{\Gamma \vdash_e [v_1, \ldots v_{n_1}] : \mathbb{R}[n_1][n_2]}\ T-Arr2D$$

$$\frac{\Gamma \vdash_e x : \mathbb{R}[n_1][n_2] \quad \Gamma \vdash_e y : \mathbb{R}[n_1][n_2]}{\Gamma \vdash_e x + y : \mathbb{R}[n_1][n_2]}\ T-Add$$

$$\frac{\Gamma \vdash_e x : \mathbb{R}[n_1][n_2] \quad \Gamma \vdash_e y : \mathbb{R}[n_2][n_3]}{\Gamma \vdash_e x \times y : \mathbb{R}[n_1][n_3]}\ T-Mult$$

$$\frac{\Gamma \vdash_e x : \mathbb{R}[n] \quad \Gamma \vdash_e y : \mathbb{Z}}{\Gamma \vdash_e x[y] : \mathbb{R}}\ T-Idx1D \qquad \frac{\Gamma \vdash_e x : \mathbb{R}[n_1][n_2] \quad \Gamma \vdash_e y : \mathbb{Z}}{\Gamma \vdash_e x[y] : \mathbb{R}[n_2]}\ T-Idx2D$$

$$\frac{\Gamma \vdash_e e : \tau[1]}{\Gamma \vdash_e e : \tau}\ T-Squeeze \qquad \frac{\Gamma \vdash_e e : \tau}{\Gamma \vdash_e e : \tau[1]}\ T-Unsqueeze$$

Fig. 4. Type system

and has a type τ, and results in a new environment Γ_2". The expression judgment $\Gamma \vdash_e e : \tau$ is read as "under the typing environment Γ, the expression e is well typed and has a type τ". $\mathcal{L}$ is statically typed and the compiler checks that the arithmetic operators are applied to matrices of compatible dimensions. For example, when adding matrices, we check that the both the matrices have the same dimensions. The static information about dimensions is used to generate accurate fixed-point code expressed in the syntax of the target language $\mathcal{T}$ (Figure 2).

5.3 Compilation

Figure 5 describes the rules used by Shiftry to compile input code in $\mathcal{L}$ (Figure 1) to the target language $\mathcal{T}$ (Figure 2). We omit the operational semantics of $\mathcal{T}$ as they are standard and only present the semantics of the operators of $\mathcal{T}$ in Library 7, 8 and 9, and helper methods (used during the compilation process) in Library 6.

The main difference between $\mathcal{L}$ and $\mathcal{T}$ is that of explicit type-based parametrization of operators. For example, $\times_{\tau'}$ multiplies two matrices and generates an output matrix whose entries have a type τ' (Library 7). In the types τ' of $\mathcal{T}$ (Figure 2), $\texttt{int}_b$ denotes a b-bit integer. Since different variables in $\mathcal{T}$ can have different bitwidths, these annotations are required to ensure that the computations are performed with the right bitwidths. The function `dim` returns the dimensions of the matrices; this information is used in compiling arithmetic operators in Figure 5.

The compilation process requires two environments σ and β. The environment β maps variables to their bitwidths and σ maps variables to scales. The judgement $\sigma, \beta \vdash s \to s'$ is read as: "Under scales σ and bitwidths β, the statement $s \in \mathcal{L}$ is compiled to $s' \in \mathcal{T}$".

Apart from σ and β, the compilation process requires the following parameters: $\sigma_{e_8^{in}}$, $\sigma_{e_8^{out}}$, $\sigma_{e_{16}^{in}}$, $\sigma_{e_{16}^{out}}$, T_8, T_{16}^1, T_{16}^2 and ψ. These parameters are used to evaluate transcendental functions and we discuss them in Section 5.4. We show how σ and β are set in Section 6. Library 7's method `ShiftVars` is required for memory management and we discuss it in Section 7.1.

$$\frac{\tau'_y = \mathtt{int}_{\beta(y)} \quad \sigma_{xy} = \sigma(x) - \sigma(y)}{\sigma, \beta \vdash y = x \longrightarrow y = \Psi_{\tau'_y}(x, \sigma_{xy})}\ C-Var$$

$$\frac{v^Q = \lfloor v \times 2^{\sigma(x)} \rfloor_{\beta(x)}}{\sigma, \beta \vdash x = v \longrightarrow x = v^Q}\ C-Assn2D \qquad \frac{}{\sigma, \beta \vdash \mathbf{return}\ x\ >\ 0 \longrightarrow \mathbf{return}\ x\ >\ 0}\ C-Ret$$

$$\frac{\begin{array}{c}(n_1, n_2) = \mathtt{dim}(x) \\ \tau' = \mathtt{int}_{\beta(x)}[\mathtt{n_1}][\mathtt{n_2}] \\ v^Q = \lfloor v \times 2^{\sigma(x)} \rfloor_{\beta(x)}\end{array}}{\sigma, \beta \vdash x := v \longrightarrow \tau' x := v^Q}\ C-Init2D \qquad \frac{\begin{array}{c}(n_1, n_2) = \mathtt{dim}(x) \\ \tau' = \mathtt{int}_{\beta(x)}[\mathtt{n_1}][\mathtt{n_2}]\end{array}}{\sigma, \beta \vdash \tau\ x \longrightarrow \tau' x}\ C-Decl2D$$

$$\frac{\sigma, \beta \vdash s \longrightarrow s'}{\sigma, \beta \vdash \mathbf{for}\ i\ = [0:n]\ \mathbf{do}\ s \longrightarrow \mathbf{for}\ i\ = [0:n]\ \mathbf{do}\ s'}\ C-Loop$$

$$\frac{}{\sigma, \beta \vdash x = y[z] \longrightarrow x = y[z]}\ C-Index \qquad \frac{\sigma, \beta \vdash s_1 \longrightarrow s'_1 \quad \sigma, \beta \vdash s_2 \longrightarrow s'_2}{\sigma, \beta \vdash s_1; s_2 \longrightarrow s'_1; s'_2}\ C-Seq$$

$$\frac{\begin{array}{c}\tau'_x = \mathtt{int}_{\beta(x)} \quad \tau'_y = \mathtt{int}_{\beta(y)} \quad \tau'_z = \mathtt{int}_{\beta(z)} \quad \tau'_{temp} = \mathtt{int}_{\mathtt{max}(\beta(x),\beta(y))} \\ \mathtt{dim}(x) = \mathtt{dim}(y) = \mathtt{dim}(z) = (n_1, n_2) \quad \sigma_{min} = \mathtt{min}(\sigma(x), \sigma(y)) \\ \sigma'_x = \sigma(x) - \sigma_{min} \quad \sigma'_y = \sigma(y) - \sigma_{min} \quad \sigma'_z = \sigma_{min} - \sigma(z)\end{array}}{\sigma, \beta \vdash z = x + y \longrightarrow z = \Psi_{\tau'_z}(\Psi_{\tau'_{temp}}(x, \sigma'_x) +_{\tau'_{temp}} \Psi_{\tau'_{temp}}(y, \sigma'_y), \sigma'_z)}\ C-MatAdd$$

$$\frac{\begin{array}{c}\tau'_x = \mathtt{int}_{\beta(x)} \quad \tau'_y = \mathtt{int}_{\beta(y)} \quad \tau'_z = \mathtt{int}_{\beta(z)} \quad \mathtt{dim}(x) = (n_1, n_2) \\ \mathtt{dim}(y) = (n_2, n_3) \quad \mathtt{dim}(z) = (n_1, n_3) \quad \tau'_{temp} = int_{2^{\lceil log_2(\beta(x)+\beta(y)+\lceil log_2(n_2)\rceil - 1)\rceil}}\end{array}}{\sigma, \beta \vdash z = x \times y \longrightarrow z = \Psi_{\tau'_z}(x \times_{\tau'_{temp}} y, \sigma(x) + \sigma(y) - \sigma(z) - \lceil log_2(n_2) \rceil)}\ C-MatMul$$

$$\frac{\beta(y) = \beta(x) = 8 \quad \sigma(y) = \sigma_{e_8^{out}} \quad \sigma'_x = \sigma(x) - \sigma_{e_8^{in}} \quad T_8 = \mathtt{getTable}_8(\sigma_{\mathrm{e}_8^{in}}, \sigma_{\mathrm{e}_8^{out}})}{\sigma, \beta \vdash y = \mathtt{exp}(x) \longrightarrow y = \mathtt{Exp^Q}_8(T_8, \Psi_{int_8}(x, \sigma'_x))}\ C-Exp8$$

$$\frac{\begin{array}{c}\beta(y) = \beta(x) = 16 \quad \sigma(y) = \sigma_{e_{16}^{out}} \quad \sigma'_x = \sigma(x) - \sigma_{e_{16}^{in}} \\ (T_{16}^1, T_{16}^2) = \mathtt{getTables}_{16}(\sigma_{e_{16}^{in}}, \sigma_{e_{16}^{out}}, \psi)\end{array}}{\sigma, \beta \vdash y = \mathtt{exp}(x) \longrightarrow y = \mathtt{Exp^Q}_{16}(T_{16}^1, T_{16}^2, \Psi_{int_{16}}(x, \sigma'_x), \psi, \sigma_{e_{16}^{out}})}\ C-Exp16$$

$$\frac{\beta(y) = \beta(x) = 8 \quad \sigma(y) = \sigma_{e_8^{out}} \quad \sigma'_x = \sigma(x) - \sigma_{e_8^{in}} \quad T_8 = \mathtt{getTable}_8(\sigma_{\mathrm{e}_8^{in}}, \sigma_{\mathrm{e}_8^{out}})}{\sigma, \beta \vdash y = \mathtt{sigmoid}(x) \longrightarrow y = \mathtt{Sigmoid^Q}_8(T_8, \Psi_{int_8}(x, \sigma'_x), \sigma_{e_8^{out}})}\ C-Sgmd8$$

Fig. 5. Compilation rules

Consider the compilation rules for matrix multiplication (operator $\times$) and matrix addition (operator $+$) in Figure 5. Here, the scales of the arguments are first adjusted using the *scale shifting* function Ψ and then the relevant operator of Library 7, 8 or 9 is called with these adjusted arguments. These operators first convert arguments to a common bitwidth, say b, and then perform a standard matrix addition (`MatAdd`) or matrix multiplication (`MatMul`) over b-bit integers. Entire copies of typecasted input matrices are not made in the actual implementation, the type conversions are done on the fly while carrying out the operation. In the actual implementation, calls to Ψ are inlined with other operators like `MatAdd`, `MatMul`, etc., and Ψ is shown as a separate function call for ease of presentation.

The scale shifting function, Ψ divides an integral fixed-point value by a power-of-two to alter its scale. In $\Psi_{\mathtt{int}_b}$, b denotes the bitwidth of the result. For example, consider the 16-bit fixed-point representation of 5.697, as discussed in Section 2. In 16-bit fixed-point arithmetic, for a scale of 12, 5.697 is represented by the integer 23334. The transformation: $\Psi(23334, 8) = 23334/2^8 = 91$ produces 91, which is 5.697 in 16-bit fixed-point arithmetic with a scale of 4. Thus, applying the $\Psi(v, n)$ function to a value v of scale s reduces its scale to $s - n$. The Ψ operations incur a runtime overhead proportional to the complexity of the primary operation, e.g., while multiplying an $i \times j$ matrix with a $j \times k$ matrix, the Ψ operations incur $O(ijk)$ shift operations.

5.4 Computing Exponentials

The source language $\mathcal{L}$ provides three transcendental functions f that depend on the irrational e. We show how to compute `tanh` and `sigmoid` using a procedure to compute e^x in Section 5.4.1. We start by describing the techniques used by Shiftry for computing $f(x) = \mathtt{exp}(x)$ when the input x is a 16-bit/8-bit fixed-point number.

For 16-bit integers, Shiftry uses the exponentiation method of SeeDot [Gopinath et al. 2019a] that approximates exponentiation as a product of two values looked up from two different tables:

$$e^x = e^{2^\psi a + b} = e^{2^\psi a} \times e^b \approx T^1_{16}[a] \times T^2_{16}[b]$$

Here, a is a $15-\psi$-bit number and b is a ψ bit number. Our procedure differs from SeeDot [Gopinath et al. 2019a] in the choice of ψ. We set $\psi = 7$ in our evaluation which leads to a slightly higher Flash usage but better precision. Specifically, we need to store the table T^1_{16} with 2^8 entries and the table T^2_{16} with 2^7 entries of 16-bits each which brings the total Flash usage to 0.75KB for positive x. For negative x, we need another 0.75KB.

For 8-bit integers, instead of breaking x into 2 parts, we simply perform a single table lookup from a table T_8, which occupies only 128 bytes (a table with 2^7 entries, each occupying 8 bits).

Library 6: Auxillary functions

Function $\mathtt{getTable}_8(\sigma_{in}, \sigma_{out})$:
- $Table : \mathtt{int}_8[\,]$
- **for** $i \in [0 : 2^7]$ **do**
  - $Table[i] \longleftarrow \lfloor e^{\frac{i}{2^{\sigma_{in}}}} \times 2^{\sigma_{out}} \rfloor$
- **return** $Table$

Function $\mathtt{getTables}_{16}(\sigma_{in}, \sigma_{out}, \psi)$:
- $Table_1, Table_2 : \mathtt{int}_{16}[\,]$
- **for** $i \in [0 : 2^{15-\psi}]$ **do**
  - $Table_1[i] \longleftarrow \lfloor e^{\frac{i}{2^{(\sigma_{in}-\psi)}}} \times 2^{\sigma_{out}} \rfloor$
- **for** $i \in [0 : 2^{\psi}]$ **do**
  - $Table_2[i] \longleftarrow \lfloor e^{\frac{i}{2^{\sigma_{in}}}} \times 2^{\sigma_{out}} \rfloor$
- **return** $(Table_1, Table_2)$

Library 7: Functions for codegen

Operator $+_{\tau'}(A, B)$:
- **return** $\mathtt{MatAdd}((\tau')A, (\tau')B)$

Operator $-_{\tau'}(A, B)$:
- **return** $\mathtt{MatSub}((\tau')A, (\tau')B)$

Operator $\times_{\tau'}(A, B)$:
- **return** $\mathtt{MatMul}((\tau')A, (\tau')B)$

Function $\Psi_{\tau'}(A, n)$:
- **return** $(\tau')(\frac{A}{2^n})$

Function $\mathtt{ShiftVars}(migrateList)$:
- **for** $(a, b, c) \in migrateList$ **do**
  - $mem_{[b:b+c]} \longleftarrow mem_{[a:a+c]}$
- **return**

Although the table-based approach suffices to compute one exponentiation, if there are multiple calls to `exp` with arguments of distinct scales then we need different tables for each such call. To save memory, we use the following observation that enable us to compute all calls to `exp` in under 1KB of Flash.

The ML algorithms in our benchmarks can be rewritten to ensure that we need to compute e^x only for negative x (Section 5.4.1). Hence, we need the table(s) only for negative values of x. For $x \leq 0$, e^x lies in the range $(0, 1]$. For 8-bit integers, to avoid overflows, the maximum possible scale of the output is 6 (the scale of 7 would overflow for e^0). Recall, that higher scales lead to more precise results and we set the output scale of 8-bit exponentiation, $\sigma_{e_8^{out}}$, as 6. With an output scale of 6, the smallest non-zero output of fixed-point exponentiation is $2^{-6} \approx e^{-4.15}$. Hence, for any input below -4.15, the fixed-point output of exponentiation must be zero. Therefore, we can set the input scale $\sigma_{e_8^{in}}$ to 4 and map the output of all negative numbers with magnitude more than 4.15 to zero. Similarly for 16 bit exponentiation, $\sigma_{e_{16}^{in}}$ is set to 11, and $\sigma_{e_{16}^{out}}$ is set to 14. By fixing these values, we only need one instance each of T_8, T_{16}^1, and T_{16}^2, and the scales of arguments are adjusted to match the input scales $\sigma_{e^{in}}$ using Ψ.

Library 8: Functions for codegen

Function $\mathtt{Exp^Q}_8(x, T)$:
- **return** $T[x]$

Function $\mathtt{Sigmoid^Q}_8(x, T, n)$:
- **if** $x \leq 0$ **then**
  - $a \longleftarrow \mathtt{Exp^Q}_8(x, T)$
  - **return** $(2^n \times a)/(2^n + a)$
- **else**
  - $a \longleftarrow \mathtt{Exp^Q}_8(-x, T)$
  - **return** $(2^n \times 2^n)/(2^n + a)$

Library 9: Functions for codegen

Function $\mathtt{Exp^Q}_{16}(x, T_1, T_2, \psi, n)$:
- **return** $\Psi_{\mathtt{int}_{16}}(T_1[x/2^{\psi}] \times T_2[x\%2^{\psi}], n)$

Function $\mathtt{TanH^Q}_{16}(x, T_1, T_2, \psi, n_1, n_2)$:
- **if** $x \leq 0$ **then**
  - $a \longleftarrow \mathtt{Exp^Q}_{16}(2x, T_1, T_2, \psi, n_1)$
  - **return** $(2^{n_2} \times (a - 2^{n_2}))/(a + 2^{n_2})$
- **else**
  - $a \longleftarrow \mathtt{Exp^Q}_{16}(-2x, T_1, T_2, \psi, n_1)$
  - **return** $(2^{n_2} \times (2^{n_2} - a))/(a + 2^{n_2})$

5.4.1 *Computing sigmoid and tanh.*

We use $e_Q(x)$ to denote e^x with $x < 0$. Here, we show how to express `sigmoid` and `tanh` using e_Q. Consider the sigmoid function $\mathtt{sigmoid}(x) = \frac{1}{1+e^{-x}}$. For $x \geq 0$, $\mathtt{sigmoid}(x) = \frac{1}{1+e_Q(-x)}$. For $x < 0$, $\mathtt{sigmoid}(x) = \frac{e_Q(x)}{1+e_Q(x)}$. Similarly, for $x < 0$, $\mathtt{tanh}(x) = \frac{e_Q(2x)-1}{e_Q(2x)+1}$ and for $x \geq 0$, $\mathtt{tanh}(x) = \frac{1-e_Q(-2x)}{1+e_Q(-2x)}$. The 8-bit fixed-point implementation for `sigmoid` is provided in Library 8. It performs an integer division between a 16-bit number and an 8-bit number to output an 8-bit result. Similarly the 16-bit fixed-point implementation for `tanH` is provided in Library 9, which divides a 32-bit number and a 16-bit number to output a 16-bit result.

6 SCALE AND BITWIDTH ASSIGNMENT

The compilation process described in the previous section outputs a fixed-point code given mappings from variables to their bitwidths and their scales. We discuss how Shiftry infers scales assuming a bitwidth assignment (Section 6.1) and then Shiftry's mechanism to assign bitwidths (Section 6.2).

In this section, we use the reciprocal of *disagreement ratio* as our precision metric to measure the deviation between the floating-point model and fixed-point code. The disagreement ratio between two models A and B is a measure of the fraction of points in the validation set where the predictions of A and B do not match. In particular, disagreement ratio between model A and A is 0. Although classification accuracy appears to be a reasonable candidate for a precision metric, empirically, we have observed that the best code (good classification accuracy, better speed, smaller model size) is obtained when we used disagreement ratio, rather than classification accuracy, as the precision metric.

6.1 Data-Driven Scaling

Algorithm 10: Data-Driven scale computation

Function `GetScale(`$value$: `float`, $bitwidth$: `Int`)**:**
  return $(bitwidth - 1) - \lfloor log_2(value) + 1 \rfloor$

Function `Profile(`var : `Var`, $value$: `float`, $varToMinMax$: `Var` $\mapsto$ (`Float`, `Float`))**:**
  $(m, M) \longleftarrow varToMinMax[var]$
  $varToMinMax[var] \longleftarrow$ (`Min(`m, $value$), `Max(`M, $value$))
  return

Function `ComputeVarScales(`$varToMinMax$: `Var` $\mapsto$ (`Float`, `Float`), $varToBitwidth$: `Var` $\mapsto$ `Int`)**:**
  $varToScale$: `Var` $\mapsto$ `Int`
  for $var \mapsto (m, M) \in varToMinMax$ **do**
    $varToScale[var] \longleftarrow$ `GetScale(Max(`$|m|$, $|M|$), $varToBitwidth[var]$)
  return $varToScale$

Shiftry computes the scale of all the variables in the program by profiling the floating-point version of the code on the given validation set of inputs. It runs the floating-point code for available inputs, and records the maximum and minimum values taken by the variables, using the `Profile` procedure of Algorithm 10. Once these extrema are stored in *varToMinMax*, Algorithm 10's method `ComputeVarScales` computes the scales for the variables using their bitwidths *varToBitwidth*.

This technique results in a good scale assignment for most variables. However, it produces unsatisfactory results in two cases:

- For the input X to the classifier, outliers result in a poor scale assignment. For example, consider a bitwidth of 16 and 100,000 samples, where 99,998 samples lie in the range (-2, 2), but the remaining two are 9 and 17. Thus, for most inputs, a scale of 14 ensures that there are no overflows. However, to ensure that the outliers do not overflow, the scale would have to be reduced from 14 to 10, resulting in a loss of 4 bits of precision, which degrades precision for most inputs.
- Similarly, for 8-bit integers, the scale computed by this method can be too coarse; To fit the extrema within an 8-bit integer, we end up losing too much precision.

We discuss our techniques to address these challenges in Section 6.2. Finally, the scales of inputs to exponentiation are set using $\sigma_{e^{in}}$ (Section 5.4).

6.2 Setting Bitwidths

Algorithm 11 is the driver method that returns *varToBitwidth* (β) and *varToScale* (σ). The function `Evaluate` takes β and σ as inputs, generates a fixed-point code using Figure 5, runs this code on the validation set, and returns the precision (Algorithm 12 and 13) or classification accuracy (Algorithm 14). Algorithm 11 uses a 4-stage process and we describe these stages next:

Algorithm 11: Lowering bitwidths of variables

```
1  Function PerformSearch(varToMinMax : Var ↦ (Float, Float)):
2      allVars ⟵ list of all variables used in the code
3      varToBitwidth ⟵ {var ↦ defaultBitwidth} ∀ var ∈ allVars
4      varToScale ⟵ ComputeVarScales(varToMinMax, varToBitwidth)
5      ExploreScaleForX(varToScale, varToBitwidth)
6      varToDemotedScalePrecision ⟵ PartialDemote(varToScale, varToBitwidth, allVars)
7      varToDemote ⟵ CumulativeDemoteVariables(floatAccuracy, dropPermitted,
         varToDemotedScalePrecision, varToScale, varToBitwidth)
8      for var ∈ varToDemote do
9          varToBitwidth[var] ⟵ defaultBitwitdth/2
10         varToScale[var] ⟵ varToDemoteScalePrecision[var][0]
11     return varToBitwidth, varToScale
```

- **Stage I** *Assigning data-driven scales.* SHIFTRY sets the bitwidth of all variables to *defaultBitwidth* (set to 16) in Algorithm 11 line 3. Using this, SHIFTRY computes the scales for all variables using data-driven scaling (Algorithm 10) in Algorithm 11 line 4.
- **Stage II** *Computing Scale of input X.* SHIFTRY computes the scale for the classifier input X in Algorithm 11 line 5. SHIFTRY iterates over all possible scales in the range $[0, \mathit{defaultBitwidth}]$, compiles the code for each scale, and picks the one with the best precision, as described in Algorithm 12.

Algorithm 12: Subroutine for assigning scale for X

```
Function ExploreScaleForX(varToScale : Var ↦ Int, varToBitwidth : Var ↦ Int):
    scaleToPrecisionLoss : Int ↦ Precision
    for scaleX ∈ 0 : defaultBitwidth do
        modifiedScales ⟵ varToScale[X ↦ scaleX]
        scaleToPrecisionLoss[scaleX] ⟵ Evaluate(varToBitwidth, modifiedScales)
    varToScale[X] ⟵ ArgMax(scaleToPrecisionLoss)
    return
```

- **Stage III** *Demoting one variable at a time and finding its best scale.* For each variable v in the program, SHIFTRY generates a new output code P_v where v is demoted to a lower bitwidth. Since Algorithm 10 does not provide good scale assignments for 8-bits variables (Section 6.1), SHIFTRY explores multiple possible scales for the demoted variables. For each variable, SHIFTRY chooses the scale which gives the best precision. Algorithm 13 defines this function and Algorithm 11 line 6 invokes it.
- **Stage IV** *Demoting variables cumulatively maintaining reasonable accuracy.* SHIFTRY proceeds to demote the variables cumulatively, ensuring that the classification accuracy does not dip below a user-provided threshold. The variables v_i are arranged in decreasing precision of P_{v_i}, in an attempt to first demote the variables that decrease the classification accuracy the least. The relevant function is defined in Algorithm 14 and called in Algorithm 11 line 7.

Algorithm 13: Computing scale and evaluating precision for one demoted variable

Function `PartialDemote`(*varToScale* : `Var` $\mapsto$ `Int`, *varToBitwidth* : `Var` $\mapsto$ `Int`, *allVars* : `Var[]`):

- *varToDemotedScalePrecision* : `Var` $\mapsto$ (`Int`, `Precision`)
- **for** *var* $\in$ *allVars* **do**
  - *scaleToPrecisionLoss* : `Int` $\mapsto$ `Precision`
  - *newBitwidths* $\longleftarrow$ *varToBitwidth*[*var* $\mapsto$ *defaultBitwidth*/2]
  - *demoteScale* $\longleftarrow$ *varToScale*[*var*] $-$ *defaultBitwidth*/2
  - **for** *scale* $\in$ *demoteScale* : *demoteScale* + 3 **do**
    - *newScales* $\longleftarrow$ *varToScale*[*var* $\mapsto$ *scale*]
    - *scaleToPrecisionLoss*[*scale*] $\longleftarrow$ `Evaluate`(*newBitwidths, newScales*)
  - *varToDemotedScalePrecision*[*var*] $\longleftarrow$ (`ArgMax`(*scaleToPrecisionLoss*), `Max`(*scaleToPrecisionLoss*))
- **return** *varToDemotedScalePrecision*

Algorithm 14: Cumulatively demoting variables while maintaining accuracy

Function `CumulativeDemoteVariables`(*floatAccuracy* : `float`, *dropPermitted* : `float`, *varToDemotedScalePrecision* : `Var` $\mapsto$ (`scale` : `Int`, `precision` : `Precision`), *varToScale* : `Var` $\mapsto$ `Int`, *varToBitwidth* : `Var` $\mapsto$ `Int`):

- `Sort`(*varToDemotedScalePrecision, descending=True, key=precision*)
- *newBitwidths* $\longleftarrow$ `Copy`(*varToBitwidth*)
- *newScales* $\longleftarrow$ `Copy`(*varToScale*)
- *varToDemote* : `Var[]`
- **for** *var* $\mapsto$ (*scale, precision*) $\in$ *varToDemotedScalePrecision* **do**
  - *newBitwidths* $\longleftarrow$ *newBitwidths*[*var* $\mapsto$ *defaultBitwidth*/2]
  - *newScales* $\longleftarrow$ *newScales*[*var* $\mapsto$ *varToDemotedScalePrecision*[*var*]]
  - *accuracy* $\longleftarrow$ `Evaluate`(*newBitwidths, newScales*)
  - **if** *accuracy* $\leq$ *floatAccuracy* $-$ *dropPermitted* **then**
    - **break**
  - *varToDemote*.`Insert`(*varName*)
- **return** *varToDemote*

The output of these stages is a fixed-point code with 16-bit and 8-bit variables that has significantly less memory footprint compared to 32-bit floating-point code (Section 8). Next, we discuss our memory management mechanism to further reduce the RAM usage.

7 MEMORY MANAGEMENT

We describe the memory management mechanism of SHIFTRY that minimizes the RAM usage of a program by reusing the memory locations for temporally disjoint variables. In particular, the fixed-point code generated by SHIFTRY has temporary variables that have short but overlapping live ranges (e.g., Table 3); the variables with disjoint live ranges can use the same RAM locations.

Algorithm 16 is the top level algorithm. It takes as input the size of the available RAM, *memoryLimit*, and returns a mapping *varToBlockList*, which maps instructions to maps from variables

Algorithm 15: Data structure used for memory management

```
Class Memory:
    varToLocation : Var ↦ (start : Addr, end : Addr)
    memoryUsage ⟵ 0
    Function Collide((start1, end1) : (Addr, Addr), (start2, end2) : (Addr, Addr)):
        return end1 < start2 ∨ end2 < start1
    Function IsFree(start : Addr, end : Addr):
        for var ↦ (varStart, varEnd) ∈ varToLocation do
            if Collide((varStart, varEnd), (start, End)) then
                return false
        return true
    Function FreeDead(varToLiveRange : Var ↦ (start : Int, end : Int), inst : Int):
        for var ↦ (_, end) ∈ varToLiveRange) do
            if end < inst then
                delete varToLocation[var]
        return
    Function Allocate(var : Var, (start, end) : (Addr, Addr)):
        varToLocation ⟵ varToLocation[var ↦ (start, end)]
        memoryUsage ⟵ Max(memoryUsage, end)
        return
    Function MemoryUsage():
        return memoryUsage
    Function GetBlockForVar(var : Var):
        return varToLocation[var]
```

to their memory locations (i.e., the starting memory address and the ending memory address). Because of defragmentation (Algorithm 17), the same variable might be placed at different memory addresses at different instructions (Section Section 7.1). Although computing *varToBlockList* statically is impossible for arbitrary programs, here Shiftry knows the size of all the parameters at compile time (line 11 of Algorithm 16) that makes computing this mapping feasible. In Section 3, such a mapping is used to generate Pseudocode 5 from Pseudocode 4. If there is an instruction i where the sum of sizes of all live variables, $SumSize_i$, exceeds the *memoryLimit* then Algorithm 16 fails at line 21. By default, Shiftry sets *memoryLimit* as $\max_i SumSize_i$.

Algorithm 16 uses the `Memory` class described in Algorithm 15. This class maintains a map from variables to their start and ending memory addresses. It also records the maximum ending address of the allocated variables in *memoryUsage*. It encapsulates the following procedures:

- `IsFree`: Checks whether the given continguous memory block is occupied by some other currently live variable.
- `FreeDead`: Deallocates all dead variables using the information about live ranges.
- `Allocate`: Allocate the given memory block to the given variable. Assumes that `IsFree` returns *true* for the given memory block.

Algorithm 16: Reusing memory for temporary variables

```
   Function VarToMemoryLocation(memoryLimit : Int):
1      varToBlockList : Int ↦ Var ↦ (start : Addr, end : Addr)
2      varToBlock : Var ↦ (start : Addr, end : Addr)
3      varToLiveRange : Var ↦ (start : Int, end : Int)
4      for var ∈ allVars do
5          varToLiveRange[var].start ⟵ instruction number where var is first used
6          varToLiveRange[var].end ⟵ instruction number where var is last used
7      Sort (varToLiveRange, key=(start, end))
8      currentInstruction ⟵ 0
9      mem ⟵ Memory()
10     for var ↦ (startInstruction, _) ∈ varToLiveRange do
11         blockSize ⟵ ComputeBlockSize(Size(var))
12         currentInstruction ⟵ startInstruction
13         mem.FreeDead(currentInstruction, varToLiveRange)
14         i ⟵ min_{n≥0}(n : mem.IsFree(n*blockSize, (n+1)*blockSize))
15         block ⟵ (i ∗ blockSize, (i + 1) ∗ blockSize)
16         mem.Allocate(var, block)
17         if mem.MemoryUsage() > memoryLimit then
18             varToBlockList[currentInstruction] ⟵ varToBlock
19             mem, migrateList ⟵ Defragment(mem, var)
20             if mem.MemoryUsage() > memoryLimit then
21                 throw Unable to fit in memory limit
22             varToBlock ⟵ Copy(mem.varToLocation)
23         varToBlock[var] ⟵ mem.GetBlockForVar(var)
24     varToBlockList[currentInstruction] ⟵ varToBlock
25     return varToBlockList
```

At a high level, Algorithm 16 works as follows. First, it determines the live ranges [Aho et al. 2006] and then sorts the variables based on the first[2] instruction they are live (line 7). Then, Algorithm 16 assigns memory blocks to the variables. It iterates through the sorted list of variables and for each variable, it computes a *blockSize*, by rounding the size of the variable to the next multiple of the most frequently occurring variable size in the program. For example, a variable which needs 25 bytes is assiged a block size of 32 if most variables have a size of 16. Next, we perform the following steps:

- Since the variables are arranged in ascending order of the starting instruction, if we arrive at a variable, say x, all variables that are live before x have some memory assigned to them. Specifically, variables whose ending instruction is less than x's starting instruction are dead and we do not need to store their values anymore. Algorithm 16 deallocates the memory blocks of these dead variables on line 13.
- We look for a contiguous block of memory (line 14) of x's *blockSize* which is not assigned to another live variable. We only look for empty blocks aligned to an integral multiple of

[2] We do not consider declarations (Section 5.1) while computing the live ranges.

blockSize. For example, for a variable with block size 32, we only check if addresses 0 to 32, or 32 to 64, or 64 to 96 etc. are free. This heuristic ensures that small variables are assigned memory blocks close by and once freed, create a large contiguous chunk of memory to accommodate the larger variables.

- We assign the first available block found in the previous step to x, and continue the loop (lines 10 to 23) until all variables are handled. We also check whether the allocation overflows the specified *memoryLimit*. If an overflow occurs, we run a defragmentation procedure (Section 7.1) that arranges the variables more compactly and makes space for x. If we fail to allocate x, even after defragmentation, then Shiftry raises an exception.

Once Algorithm 16 has computed the map *varToBlockList*, Shiftry uses it to replace variable names with the memory addresses. For example, in Section 3, Pseudocode 5, variable names have been replaced by memory blocks (for example t_1 is replaced by the 4-byte access $mem_{0:4}$). Note that this memory management mechanism is only applied to the (mutable) temporaries and is not applied to (read-only) model parameters as the parameters reside in the Flash.

7.1 Defragmentation

Algorithm 17: Defragmentation

```
Function Defragment(oldMem : Memory, lastVar : Var):
    newMem ⟵ Memory()
    Sort(oldMem.varToLocation, key=start, order=ascending)
    filledMemory ⟵ 0
    migrateList : (Addr, Addr, Int)[]
    for var ↦ (varStart, varEnd) ∈ oldMem.varToLocation do
        blockSize ⟵ varEnd − varStart
        if var ≠ lastVar ∧ varStart ≠ filledMemory then
            migrateList.Insert(varStart, filledMemory, blockSize)
        newMem.Allocate(var, (filledMemory, filledMemory + blockSize))
        filledMemory += blockSize
    return newMem, migrateList
```

Fragmentation is a well-known problem that any memory management mechanism must address. For example, suppose we have 96 bytes of available RAM. First, we allocate addresses 0 through 31 for variable x_1, 32 through 63 for variable x_2, and 64 through 95 for variable x_3. Next, suppose x_1 and x_3 become dead and the memory assigned to them is deallocated, Finally, we try to allocate a variable x_4 that needs 64 bytes. Although, 64-bytes of RAM is free, the memory has been *fragmented* by x_2 and we fail to allocate x_4. We propose a memory *defragmentation* method in Algorithm 17, which is called on line 19 of Algorithm 16, that helps Shiftry guarantee the absence of allocation failures due to fragmentation. In particular, defragmentation can *migrate* x_2 to occupy addresses 0 through 31 that allows x_4 to be allocated at addresses 32 through 95.

The method `Defragment` in Algorithm 17 takes as input a fragmented memory *oldMem* and the variable *lastVar* allocating which caused the *memoryUsage* to exceed the *memoryLimit* (Algorithm 16). It returns two objects: a new `Memory` (Algorithm 15) object *newMem* which is the defragmented memory and a list of 3-tuples called the *migrateList*. For example, if the state of the memory before defragmentation was $\{x_1 \mapsto (4:7), x_2 \mapsto (16:31)\}$, *newMem* may have the state $\{x_1 \mapsto (0:3), x_2 \mapsto (4:19)\}$. Every tuple (a, b, c) in the *migrateList* encodes that the live variable which was stored at addresses a through $a + c$ before defragmentation in *oldMem* is stored at addresses b through $b + c$ in the defragmented memory *newMem*. The defragmentation process runs in the following steps:

- Once Algorithm 16 recognizes that allocating a new variable has overflowed the memory limit, it invokes Algorithm 17 with the current memory object *oldMem*.
- Algorithm 17 computes the defragmented `Memory` object *newMem* by pushing variables towards lower addresses if possible. This ensures the most compact placement for all variables.
- Algorithm 16 continues further allocation with *newMem* providing the updated mapping from variables to their memory locations. Moreover, at the program instruction that required defragmentation, it injects a call to Library 7's method `ShiftVars` in the output code with the *migrateList* as argument. At execution time, this method migrates variables from their locations in *oldMem* to their locations in *newMem*.

Althought the first two steps are static, the last step adds a linear pass over the variables as runtime overhead. However, defragmentation is only seldomly required in practice. In particular, for our benchmarks, defragmentation is not required at all as our block-based allocation scheme leads to little fragmentation when compiling the ML models used in our evaluation.

8 EVALUATION

We evaluate on two types of ML models. First, we compare Shiftry with SeeDot [Gopinath et al. 2019a,b], the state-of-the-art compiler to generate code for ML models targetting KB-sized devices. For this comparison, we use two simple yet powerful models, Bonsai [Kumar et al. 2017] and ProtoNN [Gupta et al. 2017], for which SeeDot can generate efficient code. In short, our results show that Shiftry generates code that is smaller, faster, and more accurate. Second, we consider Recurrent Neural Networks (RNNs), a powerful class of ML models suited for inference tasks on sensor data. For these models, no prior work can generate code that can run in devices with few KBs of memory. Shiftry is the first compiler to automatically generate code for RNN models that can run on tiny IoT devices. For this evaluation, we use FastGRNN [Kusupati et al. 2018], an RNN model specifically designed for IoT applications.

Shiftry is implemented in 10K lines of Python and 5K lines of C++. The compilation time of our benchmarks varies between 1 minute and 20 minutes on an Intel Core i7-6700 machine with 32GB RAM and 8 cores. We run all our experiments on an Arduino Uno [Banzi and Shiloh 2014]. It has an 8-bit, 16 MHz Atmega328P microcontroller, with 2 KB SRAM and 32 KB of Flash memory. Shiftry

Dataset	Float	Shiftry			Homogenous 8-bit			Homogenous 16-bit		
	Accuracy	Accuracy	Time	Size	Accuracy	Time	Size	Accuracy	Time	Size
DSA-19	77.8	74.5	19	19	18.4	4.6	18	76.7	×	31
INDUSTRIAL-72	90.0	88.9	0.6	14	64.3	0.1	12	89.9	×	19
GOOGLE-12	93.0	92.4	44	20	8.7	6.0	18	92.9	×	32
GOOGLE-30	84.8	84.2	54	23	3.7	7.7	22	85.1	×	39
HAR-2	91.7	91.3	47	21	50.9	8.0	15	91.6	×	25
HAR-6	92.0	89.0	45	16	14.3	8.2	15	91.7	×	26
MNIST-10	98.0	97.0	15	19	11.4	2.2	17	98.0	×	31
WAKEWORD-2	99.0	98.7	10	15	95.7	2.2	13	99.2	26	22

Table 4. Performance of Shiftry on FastGRNN models. The number of classes follows the dataset name. Accuracy is in percent, time is in seconds, and size is in kilobytes. An × in the time column indicates that configuration did not fit on the target device.

outputs C++-code which is compiled by the Arduino IDE [Banzi and Shiloh 2014] to assembly code that can run on the Uno. Arduino IDE also provides libraries that emulate floating-point arithmetic in software, thus making it possible to execute floating-point C-code on the Uno.

We evaluate on Bonsai and ProtoNN models on the same datasets as used by SeeDot [Gopinath et al. 2019a]: cifar [Krizhevsky 2009], character recognition (cr) [de Campos et al. 2009], curet [Varma and Zisserman 2005], letter [Hsu and Lin 2002], mnist [LeCun et al. 1998], usps [Hull 1994], and ward [Yang et al. 2009]. For the RNN experiments, we use models for the following (more challenging) datasets used in FastGRNN [Kusupati et al. 2018]: dsa [Altun et al. 2010], google [Warden 2018], har [Anguita et al. 2012], mnist [LeCun et al. 1998], and wakeword. These tasks include activity recognition with data from motion sensors or smartphones, and detecting wakewords and commands to voice assistants like Google Assistant and Microsoft's Cortana. We also evaluate a benchmark from an industrial partner who has deployed RNNs on the bat of a bat-and-ball game to provide feedback on the quality of the shots. On these benchmarks, we evaluate Shiftry using the following metrics: classification accuracy (Section 8.1), latency (Section 8.2), Flash usage (Section 8.3), and RAM usage (Section 8.4). We also demonstrate the general applicability of Shiftry by compressing a much larger RNN-based architecture into the memory limits of an ARM Cortex M4 class device (Section 8.5).

8.1 Classification accuracy

For these experiments, we define the *accuracy drop* for a particular tool as the difference between the classification accuracy (on the testing set) of the floating-point code and the code generated by the tool. Table 4 compares the accuracy of Shiftry to that of the floating-point, only 8-bit and only 16-bit models on RNN benchmarks. Figures 6 and 7 compare the accuracy drop of Shiftry with that of SeeDot for ProtoNN and Bonsai, respectively.

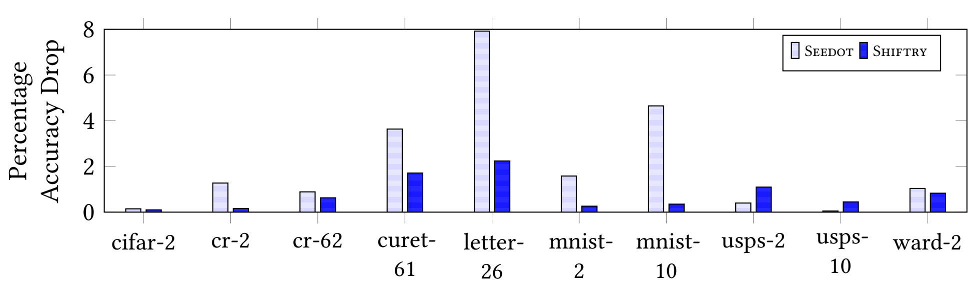

Fig. 6. Accuracy Drop for ProtoNN (lower is better)

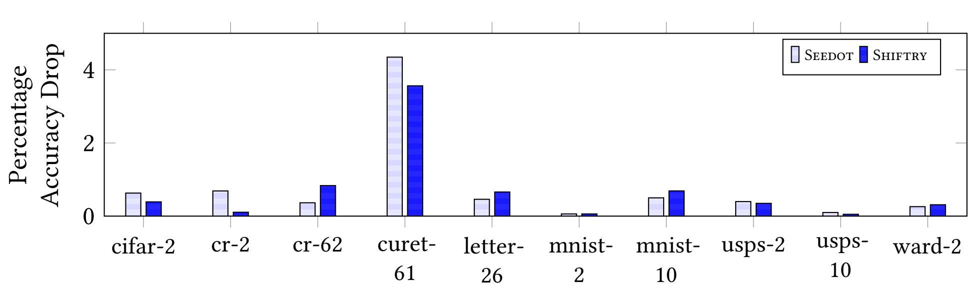

Fig. 7. Accuracy Drop for Bonsai (lower is better)

For all three models, the average (arithmetic mean) accuracy drop of Shiftry is less than 1%, showing that Shiftry can generate code that has comparable accuracy with floating point models. For ProtoNN and Bonsai, Shiftry generates code that is typically more accurate than the code generated by SeeDot. Specifically, the average (arithmetic mean) accuracy drop of Shiftry for ProtoNN/Bonsai is 0.7%/0.8% compared to that of SeeDot, 0.8%/2.3%.

8.2 Latency

Since the RNN benchmarks can only be run on an Uno with Shiftry, we do not have a baseline comparison point. For Shiftry, the RNN inference latency varies between 0.6 seconds and a minute (Table 4). Although the latency can be further improved by hardware acceleration (e.g., Sno [alorium [n. d.]] combines Arduino Uno and FPGAs), we focus on memory usage and such approaches are beyond the scope of this work.

Figures 8 and 9 show the improvement in the inference latency of both Shiftry and SeeDot compared to the floating-point implementation for ProtoNN and Bonsai. The *speedup* for a tool is computed as

$$\text{SpeedUp}(\text{tool}) = \frac{\text{Inference Time}(\text{floating-point code})}{\text{Inference Time}(\text{code generated by tool})}$$

On the ProtoNN dataset, Shiftry performed inference on an average (geometric mean) 3.5× faster than the floating-point implementation, compared to a speedup of 1.7× for SeeDot. Similarly fot the Bonsai dataset, Shiftry is 3.4× faster than the floating-point code, whereas SeeDot is 2.5× faster. Thus, Shiftry significantly improves upon the state-of-the-art in inference latency.

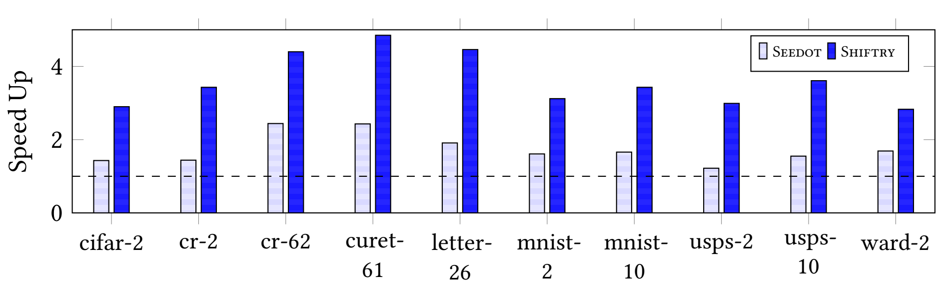

Fig. 8. Speedup for ProtoNN (higher is better)

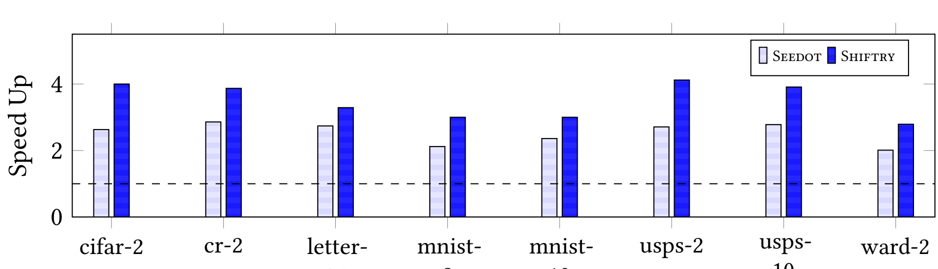

Fig. 9. Speedup for Bonsai (higher is better)

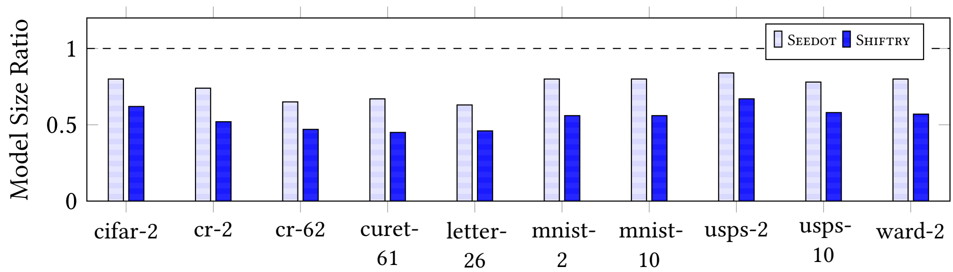

Fig. 10. Relative Model Size for ProtoNN (lower is better)

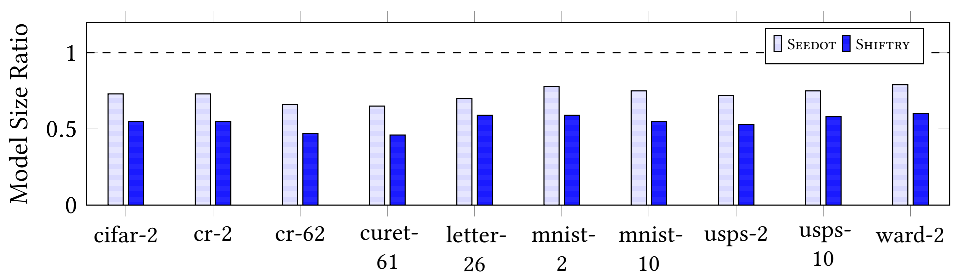

Fig. 11. Relative Model Size for Bonsai (lower is better)

8.3 Model Size Compression

To measure the Flash usage, we use the *sketch size* as measured by the Arduino IDE, the programming environment for Uno devices. The sketch size provides the total Flash usage, which includes all the Arduino boilerplate as well. We use the sketch size here because it directly dictates whether a program would fit on the device or not. In particular, programs with sketch size that exceed 32 KB fail to run on the Uno.

For the baseline RNNs, the Arduino IDE gives a compilation error that the sketch size is too large for the device. Hence, we only report the sketch sizes of Shiftry-generated RNNs in Table 4; observe that they are all comfortably below 32 KB.

Figures 11 and 10 present the *relative model size* on Arduino Uno for Bonsai and ProtoNN, respectively. We define *relative model size* for a tool as:

$$\text{Relative Model Size(tool)} = \frac{\text{Sketch Size(code generated by tool)}}{\text{Sketch Size(floating-point code)}}$$

In addition to giving better accuracy and providing faster latency, Shiftry outputs code that has smaller sketch size than SeeDot, which enables potentially larger models like RNNs to fit on the device. On an average (geometric mean), the relative size of Shiftry-generated code is 55% for Bonsai and ProtoNN. In comparison, the relative model size of SeeDot is 73% for Bonsai and 75% for ProtoNN. This extra compression is achieved as Shiftry demotes some model parameters to 8-bits but SeeDot must use 16-bits for all variables. When we used SeeDot to generate 8-bit code, the accuracy is close to that of a random classifier.

8.4 RAM usage

On a tiny device like Arduino Uno, in addition to the sketch size, it is also important to optimize the RAM usage. We must use the limited 2 KB of RAM judiciously as exceeding it leads to undefined

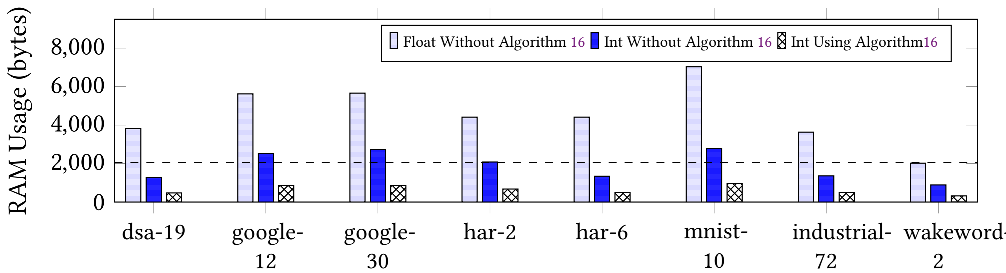

Fig. 12. RAM usage for FastGRNN (lower is better)

behavior that often manifests as non-termination at runtime. For ProtoNN and Bonsai, the models are simple enough that the RAM usage is not an issue. However, for RNNs, the complex internal computations (Appendix A) can overflow the RAM. Figure 12 shows that both floating-point and Shiftry generated fixed-point code (without memory management) exceed the available RAM. Because there is no good way to measure the precise RAM usage on an Arduino Uno, we estimate the RAM usage from the sizes of the major temporary variables (large matrices that store intermediate results) used in the program, disregarding function call overheads, scalars, etc. Hence, even though some bars of "Int Without Algorithm 16" (fixed-point code without the memory management mechanism) appear to be below the 2KB limit, they still exceed the available RAM and fail to run. In particular, most of the RNNs without Shiftry 's memory management mechanism fail to fit within the RAM.

Shiftry's memory management mechanism dramatically reduces the RAM required for the computations, and enables the Shiftry-generated programs for all RNNs to run correctly. Figure 12 demonstrates the reduction in RAM usage. Shiftry's code is able to reduce estimated RAM usage to 38% of the floating-point code without using its memory management mechanism, and with the mechanism, the estimated RAM usage fell to 13% of the floating-point RAM usage.

8.5 Generality

Although our evaluation has focused on Arduino Uno and models that can fit on it, Shiftry is a general compiler that can generate code for richer models as well. To demonstrate this generality, we implement an RNN-based architecture [Saha et al. 2020] for Face Detection in Shiftry DSL. For this model, the floating-point implementation's RAM usage is 6.9MB; Shiftry's memory management reduces it to 225KB, a 97% reduction. The floating-point model's Flash usage is 1.3 MB, which Shiftry reduced to 405KB, a 69% reduction. This enables us to fit the Face Detection algorithm on an ARM Cortex M4 class device [STMicroelectronics 2020] with 256 KB of RAM and 512 KB of Flash.

9 RELATED WORK

The closest related work to Shiftry is SeeDot [Gopinath et al. 2019a] that uses a uniform fixed bitwidth for all variables and assign scales using static analysis, that are fine tuned using profiling data. Shiftry assigns different bitwidths to different variables and the scales are directly learned [Mitchell 1997] from profiling data. SeeDot fails to meet the Flash or the RAM constraints required to run RNNs on Uno-class devices. Moreover, on the ML models SeeDot has been evaluated on, Shiftry-generated code has better latency and accuracy (Section 8).

Shiftry is closely related to work that aims to run ML on tiny microcontrollers. ProtoNN [Gupta et al. 2017] is a variant of k-nearest-neighbors and Bonsai [Kumar et al. 2017] is a variant of decision trees. These models are designed to provide good accuracy on simple classification tasks with models of minimal size. For more sophisticated ML tasks, we need more powerful classifiers like FastGRNN [Kusupati et al. 2018], which provides state-of-the-art gated-RNN accuracies in KB-sized models. Although the authors claim that FastGRNN is compatible with the Uno, their evaluation uses microcontrollers that have 16X more memory than the Uno. To run on an Uno-class device, one needs to address the memory management issues and thus this paper is the first to provide an evaluation of RNNs running on Uno. In particular, the Arduino sketches written manually in [Kusupati et al. 2018] fail to run on Uno because they exceed the Flash memory or the RAM.

There are many approximate computing frameworks for floating-point [Baek and Chilimbi 2010; Rubio-González et al. 2013; Schkufza et al. 2014; Sidiroglou-Douskos et al. 2011; Zhu et al. 2012]. Existing float-to-fixed converters like Darulova and Kuncak [2014, 2017]; Darulova et al. [2013]; Jacob et al. [2017] lack support for multiple bitwidths which is required in our benchmarks to compress model sizes while maintaining accuracy. Although, float-to-fixed converters for digital signal processors (DSPs) like [Babb et al. 1999; Banerjee et al. 2003; Bečvář and Štukjunger 2005; Brooks and Martonosi 1999; Menard et al. 2002; Nayak et al. 2001; WILLEMS 1997] can support multiple bitwidths, they use high bitwidth operations (natively supported by DSPs) in intermediate steps that are expensive on tiny microcontrollers. Shiftry-generated code is an order of magnitude faster than the latency reported for the code generated by float-to-fixed routines of MATLAB by [Gopinath et al. 2019a].

Shiftry can be considered as a *quantization* framework: In ML, quantization techniques help produce models that use low bitwidths. These techniques can be divided into three categories (in the order of increasing requisites). The first class of techniques work purely statically on a floating-point ML model [Krishnamoorthi 2018; Meller et al. 2019; Nagel et al. 2019]. Although, such techniques are attractive because of their minimal requirements, their expressiveness is extremely poor. For instance, [Nagel et al. 2019] works only for CNNs with ReLU activations and is not applicable to any of our benchmarks. The second category includes techniques that use a validation set to help with quantization. Both Shiftry and the "post-training-quantization" routine of Tensorflow-Lite [Jacob et al. 2017] fall in this category. Although the latter has good support for CNNs, its support for RNNs is preliminary. In particular, it lacks a quantization technique for the cells that are used by our RNN benchmarks. Apart from expressiveness, Tensorflow-Lite is not designed to be run on Uno-class devices; it uses an interpreter that requires over 10KB RAM.

The rest of the quantization literature falls in the third category, i.e., the techniques require backpropagation and retraining. These works do not propose mechanisms to quantize a floating-point model. Rather, they use a modified training algorithm that generates binary/integer models at the time of training (e.g., [Chen et al. 2019; Gong et al. 2019; He and Fan 2019; Hou et al. 2019; Li et al. 2017; Louizos et al. 2019; Martinez et al. 2018; Sakr and Shanbhag 2019; Zhao et al. 2019; Zhou et al. 2018]). This is still an active research area and an overwhelming majority of ML training algorithms still generate floating-point models. Moreover, these approaches do not generate quantized models that can fit in a Uno-class device and have only been evaluated on MB/GB-sized models. It is well-known that models with fewer parameters need larger bitwidths [Fromm et al. 2018]. While aggressive quantization to small bitwidths like 1-bit or 1.5-bits ([Courbariaux and Bengio 2016; Hubara et al. 2016; Lin et al. 2015; Rastegari et al. 2016]) can be made to work for large models with millions of parameters, it has not been shown to be successful for KB-sized models that can only have hundreds or thousands of parameters.

Finally, Shiftry focuses on targeting low bitwidth integer arithmetic. One can potentially use custom low-bitwidth floating-point numbers [Chen et al. 2017; Gudovskiy and Rigazio 2017; Johnson 2018; Köster et al. 2017; Miyashita et al. 2016; Zhou et al. 2017] to reduce memory, however their latency is terrible in the absence of native hardware support. Similarly, works like [Iandola et al.

2016] that save models as low-bitwidth integers on disk but convert these parameters to floating-point during computation also suffer from huge slowdowns on tiny microcontrollers that lack floating-point units.

10 CONCLUSION

We described Shiftry, a compiler that takes an ML model as input and generates code that has minimal memory footprint, which makes running ML on tiny devices feasible. In particular, we have demonstrated the first empirical evaluation of RNNs on Arduino Uno. While prior work aims to reduce inference latency while maintaining accuracy, Shiftry is designed to minimize memory usage while maintaining good accuracy and latency. To reduce Flash usage, we use low-bitwidth integers with data-driven scaling that help Shiftry outperform state-of-the-art systems in both latency and accuracy. Finally, Shiftry provides a memory management mechanism to reduce RAM usage that enables running RNNs in 2KB of RAM. As future work, we would like to add an FPGA-backend to Shiftry that would allow running ML models on FPGAs with small form factor. Such FPGAs have extremely low energy consumption and are desirable for IoT.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback on the paper. We also thank Sridhar Gopinath, Prateek Jain, Shikhar Jaiswal, Praneeth Netrapalli, Oindrila Saha, Harsha Vardhan Simhadri and Shubham Ugare for their helpful feedback and discussions.

Appendices

A RECURRENT NEURAL NETWORKS

Pseudocode 18: FastGRNN inference algorithm

$X \leftarrow input;\ h_0 \leftarrow 0$
$W, U \leftarrow model\ parameters$
$b_z, b_h \leftarrow model\ parameters$
$\zeta, \nu \leftarrow model\ parameters$
$FC, timeSteps \leftarrow model\ parameters$
for $t \in [1 : timeSteps]$ **do**
  $z_t \leftarrow \mathtt{sigmoid}(W \times X[t] + U \times h_{t-1} + b_z)$
  $\tilde{h}_t \leftarrow \mathtt{tanh}(W \times x[t] + U \times h_{t-1} + b_h)$
  $h_t \leftarrow (\zeta(1 - z_t) + \nu) \odot \tilde{h}_t + z_t \odot h_{t-1}$
$res \leftarrow h_{timeSteps} \times FC$
return $\mathtt{argmax}(res)$

Pseudocode 19: FastGRNN in Shiftry DSL

$X := \mathtt{file}(99, 1, 32);\ H := \mathtt{zeros}(1, 100)$
$W := \mathtt{file}(32, 100);\ U := \mathtt{file}(100, 100)$
$Bz := \mathtt{file}(1, 100);\ Bh := \mathtt{file}(1, 100)$
$Zeta := \mathtt{file}();\ Nu := \mathtt{file}()$
$FC := \mathtt{file}(100, 30)$
$\mathtt{float[1][100]}\ a, b, c$
for $i \in [1 : 99]$ **do**
  $a = X[i] \times W + H \times U$
  $b = \mathtt{sigmoid}(a + Bz)$
  $c = \mathtt{tanh}(a + Bh)$
  $H = (Zeta \times (1.0 - b) + Nu) \odot c + b \odot H$
return $\mathtt{argmax}(H \times FC)$

Recurrent neural networks are a popular architecture that perform computations on long chains of data by reusing parameters. For example, FastGRNN [Kusupati et al. 2018] takes advantage of sparsity to generate models with relatively few parameters. We show the classification pseudocode of FastGRNN in Pseudocode 18. Even though the input X may be long, the same parameters are reused in different timesteps to save space. In Pseudocode 18, $\times$ represents matrix multiplication, $\odot$ represents Hadamard product, $+$ represents matrix addition.

This algorithm, when written in Shiftry, results in the code in Pseudocode 19 for the Google-30 dataset. Note that this code is very similar to its mathematical description in Pseudocode 18. This code uses an extended syntax compared to the one presented in Section 5. The function call $X := \texttt{file}(n_1, n_2)$ denotes that the variable X will be a matrix of dimensions $n_1 \times n_2$ read from a file "*X.npy*". If the argument list is empty, it denotes the value being read is a scalar. The function `zeros()` returns a matrix of zeros of the given dimension. In addition we also allow compound expressions and multiple declarations ($\texttt{float[1][100]}\ a, b, c$) in the extended syntax. We also allow broadcasted additions and subtractions $(1.0 - b)$, scalar to matrix multiplications $(Zeta \times (1.0 - b))$, element-wise multiplications ($\odot$), and pointwise application of `sigmoid` and `tanh` to matrices.

B ALL SUPPORTED OPERATORS

In the following, a vector is a 1-D array, a matrix is a 2-D array, and a tensor refers to any N-D array. The following is a complete list of all operators supported by Shiftry: transposing a matrix, reshaping a tensor, reading or writing to subtensors, i.e., splices of tensors, maxpool, ReLU, exponentiation, argmax, signum, hyperbolic tan, sigmoid, convolution, the ternary ? : operator, "for" loops, tensor addition and subtraction, matrix multiplication, Hadamard product (point-wise multiplication), and sparse matrix vector multiplication.

REFERENCES

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.

alorium. [n. d.]. Sno FPGA Module Arduino-Compatible, FPGA Based. https://www.aloriumtech.com/sno/.

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 (IWAAL'12)*. Springer-Verlag, Berlin, Heidelberg, 216–223. https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones

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.

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.

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.

T. Chakraborty, S.N. Akshay Uttama Nambi, R. Chandra, R. Sharma, Z. Kapetanovic, M. Swaminathan, and J. Appavoo. 2018. Fall-curve: A novel primitive for IoT Fault Detection and Isolation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software (SenSys '18)*.

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.

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.

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 (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.

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

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)*.

Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019a. 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

Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019b. Microsoft EdgeML Repository. https://github.com/microsoft/EdgeML/tree/51c5ae0b81ab259f3bcab12741d7b489b7fe49de

Denis A. Gudovskiy and Luca Rigazio. 2017. ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks. *CoRR* abs/1706.02393 (2017).

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.

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)*.

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.

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.

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).

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CoRR* abs/1712.05877 (2017). arXiv:1712.05877 http://arxiv.org/abs/1712.05877

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

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).

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

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

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.

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.

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

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

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)*.

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).

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 (CASES '02)*. Association for Computing Machinery, New York, NY, USA, 270–276. https://doi.org/10.1145/581630.581674

Tom M. Mitchell. 1997. *Machine Learning*. McGraw-Hill, New York. http://www.cs.cmu.edu/~tom/mlbook.html

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

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.

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.

Shishir Patil, Don Kurian Dennis, Chirag Pabbaraju, Rajanikant Deshmukh, Harsha Simhadri, Manik Varma, and Prateek Jain. 2018. *GesturePod: Programmable Gesture Recognition for Augmenting Assistive Devices*. Technical Report. Microsoft. https://www.microsoft.com/en-us/research/publication/gesturepod-programmable-gesture-recognition-augmenting-assistive-devices/

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 (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. arXiv:cs.CV/2002.11921

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

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.

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 (ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 124–134. https://doi.org/10.1145/2025113.2025133

STMicroelectronics. 2020. STM32 Nucleo-144 development board with STM32F439ZI MCU, supports Arduino, ST Zio and morpho connectivity. https://www.st.com/en/evaluation-tools/nucleo-f439zi.html

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.

Y. Wang, M. Chen, X. Wang, R. H. M. Chan, and W. J. Li. 2018. IoT for Next-Generation Racket Sports Training. *IEEE Internet of Things Journal* 5, 6 (2018), 4558–4566.

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

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.

Yiren Zhao, Xitong Gao, Daniel Bates, Robert Mullins, and Cheng-Zhong Xu. 2019. 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.

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)*.

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 (POPL '12)*. Association for Computing Machinery, New York, NY, USA, 441–454.