Compiling KB-Sized Machine Learning Models to Constrained Hardware

Sridhar Gopinath
Microsoft Research, India
t-srg@microsoft.com

Nikhil Ghanathe
Microsoft Research, India
v-nighan@microsoft.com

Vivek Seshadri
Microsoft Research, India
visesha@microsoft.com

Rahul Sharma
Microsoft Research, India
rahsha@microsoft.com

Abstract

Recent breakthroughs in machine learning (ML) have produced models that can directly run on constrained IoT devices. This approach allows systems to avoid expensive communication between the IoT devices and the cloud, thereby enabling energy-efficient real-time analytics. However, ML models are expressed typically in floating-point, and IoT hardware typically does not support floating-point. Therefore, running these models on IoT devices requires simulating IEEE-754 floating-point using software, which is very inefficient.

This paper presents SeeDot, a domain-specific language to express ML inference algorithms and an associated compiler which compiles SeeDot programs to fixed-point code that can efficiently run on constrained IoT devices. We propose 1) a novel compilation strategy that reduces the search space for some key parameters used in fixed-point code, and 2) new efficient implementations of expensive operations. For microcontrollers, our evaluation shows that SeeDot-generated programs have comparable accuracy, are 2.4×–11.9× faster than floating-point implementations, and up to two orders of magnitude faster than the code generated by a commercial float-to-fixed converter. SeeDot-based FPGA implementations are 18.7×–211.3× faster than microcontroller-based implementations and 5.2×–9.8× faster than FPGA implementations generated by commercial high-level synthesis tools.

Keywords Programming Languages, FPGA, IoT, Machine Learning

1 Introduction

In recent years, we have seen an increase in automation systems that deploy sensors to collect data and analyze the data using machine learning (ML) algorithms. Few examples of such systems include simple health monitoring through wearable sensors [54, 58, 59], self-driving cars [20, 40, 51], large-scale monitoring of big cities [9, 42, 49], and so on. Typical systems use sensor devices (also referred to as IoT devices) that only *collect* data and run the ML algorithms in the cloud [25, 31, 37]. However, running the ML classifiers *directly* on the IoT device has three known advantages [26, 45]. First, it improves security and privacy of user data by keeping it at the source rather than communicating it to the cloud. Second, it significantly reduces the amount of data that needs to be communicated between the IoT device and the cloud, thereby reducing energy consumption. Third, running the algorithms on the IoT device significantly reduces the latency of prediction, enabling real-time analysis.

Despite these benefits, running complex ML algorithms on IoT devices is challenging since these devices usually have limited compute and memory resources. More importantly, the hardware on these devices typically does not support floating-point operations [5]. This lack of support for floating-point is problematic as most ML algorithms are expressed in floating-point—this is true even for recently-proposed classifiers [26, 45] that are *specifically* designed to run on devices with limited resources. Existing tool-chains (e.g., the Arduino IDE) deal with this problem by emulating floating-point operations in software. This approach results in poor performance as a faithful software emulation of floating-point must take into account all the vagaries of the IEEE-754 standard [36]: ±0, NaNs, denormals, infinites, etc.

We focus on the scenario where an ML model is trained in the cloud and an IoT maker wants to deploy the trained model directly on the IoT device. In this paper, we describe the first framework that generates efficient code for ML inference algorithms that target constrained hardware with no floating-point support. To this end, we make the following contributions.

First, we propose SeeDot, a domain-specific language for expressing ML inference algorithms. SeeDot is high-level, easy to comprehend, and has intuitive semantics. It provides language support for standard matrix operations, which are the natural abstractions used in ML algorithms. With these features, SeeDot can express the LeNet convolution neural network for object detection on the CIFAR-10 dataset in ten lines of code (Section 7.3). In contrast, the corresponding C program spans hundreds of lines. Consequently, SeeDot

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

improves programmer productivity by making it easy for ML researchers to specify their algorithms. We formally define SeeDot and its semantics in Section 5.

Second, we design a compiler that transforms SeeDot programs to fixed-point C code for microcontrollers. The fixed-point code operates only on integers and is much more efficient than emulating floating-point in software. Our compiler uses two key ideas. First, each fixed-point number is associated with a *scale* parameter. The naïve approach for setting this parameter results in an unacceptable loss in precision. Our compiler uses a simple heuristic to identify the scale values that results in precise and efficient fixed-point code. Second, we observe that vanilla approaches to compute the exponentiation function (e^x) on constrained hardware are very inefficient. We propose an approach that computes e^x as a product of two values that are looked up from two pre-computed tables. This implementation significantly improves the execution time with a negligible loss in the accuracy of the model. With these techniques, SeeDot-generated fixed-point code significantly outperforms code generated by state-of-the-art float-to-fixed compiler from MATLAB by up to 82.2×. Section 3 provides a motivating example and Section 5.3 describes our optimizations in detail.

Third, we observe that IoT devices are often deployed for specific scenarios. While the models may undergo updates, they do not change significantly in structure and complexity. This makes them an ideal target for hardware acceleration using Field Programmable Gate Arrays (FPGAs) [56]. We augment SeeDot with a backend that generates code to run on a low-end, power-efficient Xilinx FPGA with no floating-point support. Our compiler uses the high-level synthesis (HLS) compiler provided by Xilinx along with two optimizations. First, our compiler uses a hand-optimized Verilog code for Sparse-Matrix-Vector (SpMV) multiplication, a frequently-occurring operation in ML inference. Second, our compiler automatically generates hints for the HLS compiler to parallelize other operations. Section 6 describes the FPGA backend in more detail. To the best of our knowledge, this is the first demonstration of automatically compiling ML algorithms specified in a high-level language to low-end FPGAs.

We evaluate SeeDot using two recently-proposed ML algorithms for constrained devices: ProtoNN [26] and Bonsai [45]. We use trained models for these two algorithms on ten standard datasets. In the microcontroller setting, we compare the performance of SeeDot-generated code to code generated by 1) the native Arduino IDE, and 2) the MATLAB float-to-fixed converter. Our evaluations show that SeeDot-generated programs achieve comparable classification accuracy (average accuracy loss is 0.5%) for the ML algorithms with 2.4×–82.2× reduction in execution time. For the FPGA setting, we compare the performance of our compiler to directly using the HLS compiler without any optimizations. Our results show that our approach is 5.2×–9.8× faster than the baseline. As expected, the FPGA implementations outperforms the microcontroller implementations by 18.7×–211.3×.

Finally, to evaluate the benefit of SeeDot in the real world, we consider two case studies where ML models have been deployed in the wild: 1) a fault detection system in which an ML model is used to detect whether a soil temperature/moisture sensor deployed in a remote farm has malfunctioned, and 2) a sensor pod that reacts in real-time to gestures performed by persons with visual impairments using their white cane. For both scenarios, SeeDot-generated fixed-point code for the ML algorithms has comparable classification accuracy and much better performance than the deployed implementations.

2 Background

In this section, we provide a brief background on machine learning and fixed-point arithmetic.

2.1 ML Preliminaries

An ML classifier takes an input data point (e.g., an image) and assigns it a *label* (e.g., "cat image" or "dog image"). A typical ML dataset has a *training* set and a *testing* set. The training set is used to learn a *model*. The performance of the trained model is judged by its *classification accuracy*, the proportion of points in the testing set that the classifier labels correctly. For example, in the linear classifier $w * x > 0$, the vector w is the trained model, the vector x is the input data which needs to be classified, the possible labels are *true* and *false*, and $*$ is the inner-product operation. In this paper, we focus on running ML *classifier* on constrained devices. Therefore, w is stored on device's memory, x is a run-time input, and the device performs the computation $w * x > 0$. To generate efficient code for a classifier, the SeeDot compiler has access to the SeeDot program, the trained model, and the training set to learn certain parameters for the compiled code. We use the testing set *only* to evaluate the performance of the code generated by our compiler and not for generating the code.

2.2 Accuracy Metric

ML classifiers are typically specified as expressions over Reals. As Real arithmetic requires infinite precision, modern processors approximate Real numbers using floating-point or fixed-point numbers for efficiency. For ML classifiers, the correctness of these implementations can be judged by two metrics: classification accuracy and numerical accuracy. The latter bounds the error between an implementation and a Real specification over all possible inputs. It is well-known that the best numerical accuracy does not necessarily result in the best classification accuracy. In fact, prior work [39, 61] observes that using fewer bits of precision can improve classification accuracy; precision reduction can be seen as a form of regularization that reduces over-fitting. However,

the choice of an accuracy metric is orthogonal to the implementation of the compiler and SeeDot can work with any metric. In particular, the example in Section 3 uses numerical accuracy. Other metrics like recall, precision, and F1-score can be used as well. In this paper, we consider an implementation of a classifier to be *satisfactory* if it has good classification accuracy, regardless of the numerical accuracy.

2.3 Fixed-Point Preliminaries

Fixed-point arithmetic represents a Real number r using an integer[1] $\lfloor r * 2^{\mathsf{P}} \rfloor$. The quantity P is called the *scale*. If $\mathsf{P} > 0$ then we say that r has been scaled up by P. If $\mathsf{P} < 0$ then we say that r has been scaled down by $|\mathsf{P}|$.

The choice of scale is critical when using a fixed-point representation. E.g., consider 8-bit integers and $r = \pi = 3.1415\ldots$. A scale of $\mathsf{P} = 5$ is optimal since it produces the most accurate result: $\lfloor \pi * 2^5 \rfloor = 100$ which represents the Real $100/2^5 = 3.125$, the most precise 8-bit fixed-point representation of π. If the scale is too high, e.g., if $\mathsf{P} = 6$ then $\lfloor \pi * 2^6 \rfloor = 200$, which when written as an 8-bit integer corresponds to -56. This situation is an *overflow*. Here, the most significant bits are lost when converting to 8-bit integers and the result is garbage. If the scale is too low, e.g., if $\mathsf{P} = -2$ then $\lfloor \pi * 2^{-2} \rfloor = 0$ and all the significant bits are lost.

Next, we show how the scale parameter can affect the precision of fixed-point addition and multiplication. Consider the real numbers $r_1 = \pi$ and $r_2 = e = 2.71828\ldots$. The corresponding 8-bit fixed-point representation using a scale $\mathsf{P} = 5$ are $y_1 = \lfloor r_1 * 2^{\mathsf{P}} \rfloor = 100$ and $y_2 = \lfloor r_2 * 2^{\mathsf{P}} \rfloor = 86$. To add the two numbers, simply computing $y_1 + y_2$ is *unsafe*, as the operation results in an overflow ($y_1 + y_2 = -70$). A naïve approach to avoid the overflow is to first scale down both the numbers by 1, and then computing the sum. With this approach, the computed fixed-point result is $\frac{y_1}{2} + \frac{y_2}{2} = 93$ with a scale $\mathsf{P} = 4$, which corresponds to the Real number $93/2^4 = 5.8125 \approx \pi + e$.

Similarly, to multiply the two numbers r_1 and r_2, computing $y_1 * y_2$ results in an overflow. The naïve approach to avoid overflow while multiplying two d-bit fixed-point numbers is to scale them down by $\frac{d}{2}$ before the multiplication, i.e., we evaluate $\frac{y_1}{2^{d/2}} * \frac{y_2}{2^{d/2}}$ in d-bit arithmetic.[2] This process would avoid any overflows due to multiplication as the result of multiplying two $\frac{d}{2}$ bit numbers would fit in d bits. The result of $\frac{y_1}{2^4} * \frac{y_2}{2^4} = 30$ with scale $2 * 5 - d = 2$, i.e., $\frac{30}{4} \approx \pi * e$.

While these naïve rules of performing fixed-point arithmetic are sufficient to guarantee the absence of overflows, we observe that they can result in a significant loss of precision. In fact, in our evaluations, applying these rules for all ML benchmarks resulted in implementations that return unacceptable results (same classification accuracy as a purely random classifier). In the following section, we describe this problem using an example.

3 Motivating Example

We use a linear classifier as a motivating example for our fixed-point compiler and to introduce the SeeDot language. The input to our example classifier described below is a vector $x \in \mathbb{R}^4$ and it returns a label $\ell \in \{true, false\}$. The classifier consists of a model $w \in \mathbb{R}^4$ and it computes $w * x > 0$, where $*$ is the inner-product of two vectors. The following program is how the classifier would be represented in SeeDot for specific values of x and w:

$$\begin{array}{l} \texttt{let } x = [0.0767; 0.9238; -0.8311; 0.8213] \texttt{ in} \\ \texttt{let } w = [[0.7793, -0.7316, 1.8008, -1.8622]] \texttt{ in} \\ w * x \end{array} \quad (1)$$

If we run this program in infinite precision Real arithmetic then $w*x$ would evaluate to -3.64214951. Floating-point arithmetic produces the approximately correct answer -3.642149448. Suppose we represent each Real number using a 8-bit fixed-point number (*bitwidth* = 8), then the best scale for each entry in x and w is 7 and 6 respectively. Choosing larger scales would result in overflows. If we mechanically apply the rules described in Section 2.3 then 1) for addition, we must scale down the inputs by 1, and 2) for multiplication, we must scale down the inputs by 4 (half the bitwidth). The resulting fixed-point code will be,

$$\begin{array}{l} \texttt{let } x = [\lfloor 0.0767 * 2^7 \rfloor, \lfloor 0.9238 * 2^7 \rfloor \ldots] \texttt{ in} \\ \texttt{let } w = [\lfloor 0.7793 * 2^6 \rfloor, \lfloor -0.7316 * 2^6 \rfloor, \ldots] \texttt{ in} \\ \frac{(\frac{w_1/2^4 * x_1/2^4}{2}) + (\frac{w_2/2^4 * x_2/2^4}{2})}{2} + \frac{(\frac{w_3/2^4 * x_3/2^4}{2}) + (\frac{w_4/2^4 * x_4/2^4}{2})}{2} \end{array} \quad (2)$$

This code loses valuable significant bits and computes an imprecise result of -2.625. In contrast, the code generated by SeeDot does the following computation:

$$\begin{array}{l} \texttt{let } x = [\lfloor 0.0767 * 2^7 \rfloor, \lfloor 0.9238 * 2^7 \rfloor \ldots] \texttt{ in} \\ \texttt{let } w = [\lfloor 0.7793 * 2^6 \rfloor, \lfloor -0.7316 * 2^6 \rfloor, \ldots] \texttt{ in} \\ ((w_1/2^4 * x_1/2^4) + (w_2/2^4 * x_2/2^4) + \ldots \end{array} \quad (3)$$

The computed value of $w * x$ by this procedure is -98 with a scale of 5 and represents the Real value $\frac{-98}{2^5} = -3.0625$. This value is a significantly better approximation of the ideal result.

Before describing how SeeDot produces the better code shown in (3), we describe an (impractical) approach to generate the optimal implementation. Suppose we non-deterministically guess the best scale that every sub-expression needs to have, we can then perform the appropriate scale-up/down operations and obtain the most accurate implementation. This non-determinism can be removed by enumerating over all possible choices of scales for all sub-expressions. This enumeration space is huge and there are over 10^{20} possibilities for our tiny example in (1).

[1] We only consider integers with fixed number of bits (e.g., 8, 16, 32, etc.).
[2] If the hardware has support for $2d$-bit multiplication then another option is to extract top d bits of $(y_1 * y_2)$ and discard the lower d bits.

4 SeeDot Design

SeeDot avoids enumerating over all the possibilities by evaluating only a very small heuristically selected subset of this vast enumeration space. To this end, our heuristic identifies a parameter *maxscale*, $\mathcal{P}$, such that the upper bound for the intermediate values is $2^{d-\mathcal{P}-1}$, where d is the bitwidth. Given a $\mathcal{P}$, SeeDot uses this information to avoid scale down operations that lose significant bits. In particular, the operands to addition and multiplication are not scaled down if their scale is below $\mathcal{P}$.

In our example, the magnitude of all intermediate values computed by the expression is less than 4 (= 2^{8-5-1}). Hence, SeeDot uses $\mathcal{P} = 5$ to generate the program in (3). Consider the sub-expression $y_1 + y_2$, where $y_i = w_i/2^4 * x_i/2^4$. Here, y_1 and y_2 both have a scale of 5. SeeDot needs to decide whether a scale down operation needs to be performed. If we are being conservative then this addition can potentially overflow. Therefore, we should perform $\frac{y_1}{2} + \frac{y_2}{2}$, thus decreasing the scale of the result to 4. However, since $\mathcal{P} = 5$, we know that the magnitude of the result is below 4 and can safely be represented using the scale of 5. Thus, we can compute $y_1 + y_2$ without performing the scale down operation and also guaranteeing no overflows. This method helps in saving significant bits in the result.

If $\mathcal{P} = 3$ then the intermediate results have a magnitude below $2^{8-3-1} = 16$. Hence, there is a possibility that $y_1 + y_2$ might produce overflows. To avoid this, the scale down operation is performed to reduce the scale to 4.

To identify the best $\mathcal{P}$, SeeDot generates a classifier program for *each* $\mathcal{P}$ in $\{0, 1, \ldots, d-1\}$ and then picks the program that achieves the best classification accuracy on the training set. In particular, the program in (3) corresponds to $\mathcal{P} = 5$ and (2) corresponds to $\mathcal{P} = 3$. For our example program, SeeDot picks $\mathcal{P} = 5$. In general, since ML datasets have outliers, we have observed that using a $\mathcal{P}$ that lets the outliers overflow but preserves significant bits on most inputs leads to better accuracy than using a $\mathcal{P}$ that ensures no overflows on all inputs. Therefore, evaluating all possible choices of $\mathcal{P}$ helps SeeDot pick the best program.

5 Formal development

SeeDot is a declarative language whose expressions specify computations over Reals. It has been designed for expressing ML inference algorithms. In this section, we describe 1) the syntax of the core language of SeeDot, 2) its type system, and 3) compilation of programs written in SeeDot to fixed-point code. We describe our new implementation for computing exponentials in Section 5.3.1, and our mechanism to determine critical parameters (e.g., $\mathcal{P}$) in Section 5.3.2. The readers who are interested in using the compiler as a black box can move directly to the Section 5.3.1.

5.1 Syntax

Figure 1 describes the syntax of SeeDot using a grammar. The values in SeeDot are integer scalars n, real scalars r, matrices in dense representation M_d, and matrices in sparse representation M_s. An example of M_d is $[[1, 2, 3]; [4, 5, 6]]$, which represents the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 6 \end{bmatrix}$$

A sparse matrix is a record consisting of two lists: a list `val` of non-zero values and a list `idx` of positions of these non-zero values in the matrix. A new identifier x is created using the `let` keyword. Run-time inputs can be modeled by free variables that are not bound by `let`. Free variables of SeeDot expressions get their values and types from environments they are executed and compiled under.

Expressions can be added or multiplied. The operator $*$ represents dense matrix multiplication and $\times$ represents multiplying a two dimensional sparse matrix with a (dense) vector. Exponentials can be computed via the `exp` keyword and the index of the maximum element of a vector can be obtained using `argmax`. The full SeeDot language has additional constructs for reshaping matrices, `for` loops, and CNN [57] specific operators such as convolutions, ReLU, and maxpool. We omit these from Figure 1 as they do not offer additional insights.

5.2 Static Semantics

We describe the type system of SeeDot in Figure 2. The possible types are the following:

$$\tau ::= \mathbb{Z} \mid \mathbb{R} \mid \mathbb{R}[n_1] \mid \mathbb{R}[n_1, n_2] \mid \mathbb{R}[n_1, n_2]^s$$

A SeeDot expression can have a type integer ($\mathbb{Z}$), or a scalar Real number ($\mathbb{R}$), or a k-dimensional matrix of Reals where $k \in \{1, 2\}$. The type of the matrix in Equation 5.1 is $\mathbb{R}[2, 3]$. Two dimensional sparse matrices with n_1 rows and n_2 columns are assigned the type $\mathbb{R}[n_1, n_2]^s$. We restrict the maximum dimension of matrices to two for the ease of presentation.

We use Γ to denote the typing environment, which is a map from variables to types. The judgement $\Gamma \vdash e : \tau$ is read as follows: under the typing environment Γ, the expression e is well-typed and has a type τ. The rule *T-Var* is standard: a variable x is well-typed if it belongs to the domain of Γ. The rule *T-Let* is also standard and adds variables to Γ. *T-Add* and *T-Mult* ensures that we only add and multiply matrices of compatible dimensions. *T-SparseMult* checks that the two arguments of $\times$ are a two dimensional sparse matrix and a vector. If a SeeDot developer multiplies or adds matrices

$$\begin{aligned} e \quad ::= \quad & n \mid r \mid M_d \mid M_s \mid x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ & \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 \times e_2 \mid \texttt{exp}(e) \mid \texttt{argmax}(e) \end{aligned}$$

Figure 1. Syntax of the core language of SeeDot

$$\frac{x \in domain(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ T-Var \qquad \frac{}{\vdash r : \mathbb{R}}\ T-Real$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ T-Let$$

$$\frac{\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2] \qquad \Gamma \vdash e_2 : \mathbb{R}[n_1, n_2]}{\Gamma \vdash e_1 + e_2 : \mathbb{R}[n_1, n_2]}\ T-Add$$

$$\frac{\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2] \qquad \Gamma \vdash e_2 : \mathbb{R}[n_2, n_3]}{\Gamma \vdash e_1 * e_2 : \mathbb{R}[n_1, n_3]}\ T-Mult$$

$$\frac{\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2]^s \qquad \Gamma \vdash e_2 : \mathbb{R}[n_2]}{\Gamma \vdash e_1 \times e_2 : \mathbb{R}[n_1]}\ T-SparseMult$$

$$\frac{\Gamma \vdash e : \mathbb{R}[n_1, n_2] \quad n_1 = n_2 = 1}{\Gamma \vdash e : \mathbb{R}}\ T-M2S$$

$$\frac{\Gamma \vdash e : \mathbb{R} \quad n_1 = n_2 = 1}{\Gamma \vdash e : \mathbb{R}[n_1, n_2]}\ T-S2M$$

$$\frac{\Gamma \vdash e : \mathbb{R}}{\Gamma \vdash \texttt{exp}(e) : \mathbb{R}}\ T-EXP \qquad \frac{\Gamma \vdash e : \mathbb{R}[n]}{\Gamma \vdash \texttt{argmax}(e) : \mathbb{Z}}\ T-ArgMax$$

Figure 2. Type system

with non-compatible dimensions then a compile time error is generated. The rules *T-M2S* and *T-S2M* coerce a 1×1-matrix to a scalar and vice versa. The exp operator takes a scalar argument and returns a scalar result. We only support exponentiation of scalars; argmax returns an integer. These rules are expected from any strongly-typed language for matrix algebra. However, the widely used production DSLs for matrix algebra (e.g., MATLAB) are dynamically typed and can only catch the errors described above at run-time.

A well-typed SeeDot expression can be executed by targeting it to computer algebra systems (e.g., Mathematica [75]) that can perform arbitrary precision Real arithmetic. Although exact Real arithmetic is useful for debugging at development time, production systems rely on approximations such as floating-point or fixed-point arithmetic to ensure high efficiency. We describe the fixed-point code generator next.

5.3 Fixed-point Compilation

The compilation rules provided in Figure 4 translate SeeDot programs to a sequence of procedure calls. These rules use the auxiliary functions described in Algorithm 1 and the pseudo-code for the procedures is described in Algorithm 2. The auxiliary functions are parameterized. The functions used in addition and multiplication rules are parameterized by $\mathcal{P}$, which was introduced in Section 4. We use $\mathcal{B}$ to denote the bitwidth. The functions for exponentiation rule require some other parameters ($\mathbb{T}$, m, and M) that are described in Section 5.3.1. In this section, we assume that the compiler has been given a valuation of these parameters by an oracle. Given such a valuation, the compilation rules can be applied to generate a fixed-point implementation statically. We discuss our strategy to set these parameters in Section 5.3.2.

The compilation environment κ maps a variable x to a unique location η and a scale P. The judgment $\kappa \vdash e : (C, \eta, \mathsf{P})$ is read as follows: under an environment κ, a SeeDot expression e is compiled to a code C, which is a sequence of procedure calls. The return value of C is stored at location η, which has a scale P. We use ϵ to denote a no-op code in Figure 4. The rules use these standard functions: function $max(W)$ returns the maximum element of a matrix W; $abs(W)$ returns a matrix containing the magnitude of each entry in the input matrix W; dim returns the dimensions of a matrix.

We discuss Figure 4 using examples. Consider the simple SeeDot program: $\texttt{let}\ x = 1.23\ \texttt{in}\ x$. Compiling this program involves using the rules *C-Let*, *C-Var*, and *C-Val*. *C-Val* uses the auxiliary function GetP to compute the scale of 1.23. If the bitwidth $\mathcal{B} = 16$, i.e., 16-bit integers are used to represent the Real numbers, then GetP returns 14 and the value 1.23 compiles to $\eta = 20152$, where $20152 = \lfloor 1.23 * 2^{14} \rfloor$. The rule *C-Let* updates the environment to map x to $(\eta, 14)$. The rule *C-Var* for variables is standard where under an environment κ, the variable x compiles to an empty program and the return value of the expression is stored at the location η with scale 14. Overall, this example compiles to $\eta = 20152; \epsilon$, where the return value in η has a scale of 14.

If our example is $\texttt{let}\ x = 1.23\ \texttt{in}\ x + x$ then the rule *C-MatAdd* is applicable. This example compiles to $\eta = 20152; \eta_1 = MatAdd(\eta, \eta, 0, 1)$, where the return value in η_1 has a scale of 13. The scales P_1 and P_2 in *C-MatAdd* are the scales of x, i.e., 14. The function AddScale computes the scale of the result of addition, which is first set to $14 - 1 = 13$. Recall, addition can result in larger numbers that can overflow. Hence, the scale needs to be reduced. Accordingly, S_{add} specifies that both the arguments of addition need to be divided by $2^{\mathsf{S}_{add}} = 2$ before adding them together. However, if $\mathcal{P}$ is large then there would be no need to reduce scale and S_{add} would be zero. The compiled fixed-point code evaluates to 20152 with a scale of 13, i.e., 2.4599609375, which is a good approximation of the exact result 2.46.

Dense and sparse matrix multiplications follow the same pattern as addition. In particular, the intermediate results need to be scaled down before they are added or multiplied. An intermediate step of matrix multiplication requires summation over a sequence of values.

We use the TreeSum procedure of Algorithm 2 that minimizes the precision loss.

5.3.1 Computing exponentials

There are two standard techniques for computing e^x: either compute a Taylor series expansion in floating-point or use a look-up table [64]. Both of these approaches are unsatisfactory. The former simulates floating-point in software

and has high latency. The latter approach has low latency but consumes a lot of memory. In particular, a look-up table for 16-bit fixed-point arithmetic would have 2^{16} entries of 16-bits each, i.e., would consume 256 KB. Therefore, this table cannot fit on the resource constrained devices that we consider in this paper. In this paper, we propose an approach that uses only 0.5KB of memory and performs 23.2× faster than emulating floating-point on an Arduino Uno.

At a high level, our approach implements e^x as a product of two values that are looked up from two tables. Specifically, we first divide x into four parts as shown in Figure 3: the sign bit, two parts of $\mathbb{T}$ each (a and b), and remaining least significant k bits (c). For the following discussion, we assume x is positive. Then we have,

$$x = 2^{\mathbb{T}+k}a + 2^k b + c$$
$$\implies e^x = e^{2^{\mathbb{T}+k}a+2^k b+c} \approx e^{2^{\mathbb{T}+k}a}.e^{2^k b} = f(a).g(b)$$

where $f(a) = e^{2^{\mathbb{T}+k}a}$ and $g(b) = e^{2^k b}$. Our idea is to implement the functions f and g using two look-up tables, T_f and T_g, respectively. For bitwidth $\mathcal{B} = 16$, $\mathbb{T} = 6$, and $k = 3$, each table has 64 entries with 2 bytes each. Therefore, the total cost of the two tables is just 256 bytes. We can use the same approach and use two additional tables to compute e^x for negative values of x.

5.3.2 Auto-tuning parameters

There are five parameters in SeeDot compiler: $\mathcal{P}$, $\mathcal{B}$, $\mathbb{T}$, m, and M. We show how these parameters are set. In our evaluation, we keep $\mathbb{T} = 6$ and the other parameters need to be explored. There are two main strategies for setting parameters: brute force and run-time profiling. In the former, we exhaustively try all possible values of the parameter and choose the one that works the best. This evaluation is performed by measuring classification accuracy on the *training* set. In the latter, we observe the runs of the ML classifier on the training data and set the parameters according to the observations. Brute force provides optimal performance and ideally we want to set all parameters this way.

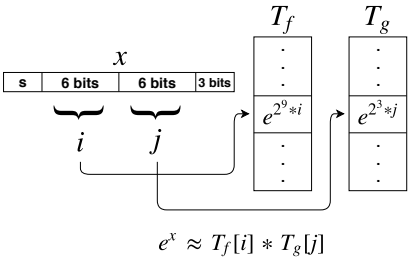


Figure 3. Computing e^x—where x is a 16-bit fixed-point number with sign bit "s"—using precomputed tables T_f and T_g.

Algorithm 1 Auxiliary functions

```
1:  function GetP(n)
2:      return (B - 1) - ⌈log2(n)⌉
3:  function MulScale(P1, P2)
4:      S_mul ← B
5:      P_mul ← (P1 - S_mul / 2) + (P2 - S_mul / 2)
6:      if P_mul < P then
7:          S_mul ← max(B - (P - P_mul), 0)
8:          P_mul ← (P1 - S_mul / 2) + (P2 - S_mul / 2)
9:      return P_mul, S_mul
10: function AddScale(P)
11:     S_add ← 1
12:     P_add ← P - 1
13:     if P_add < P then
14:         S_add ← 0
15:         P_add ← P
16:     return P_add, S_add
17: function TreeSumScale(P1, n)
18:     S_add ← ⌈log2(n)⌉
19:     P_add ← P - S_add
20:     if P_add < P then
21:         S_add ← max(S_add - (P - P_add), 0)
22:         P_add ← P - S_add
23:     return P_add, S_add
24: function ExpTable(P, m, M)
25:     k ← ⌈log2(M − m)⌉
26:     P1 ← GetP(e^m)
27:     P2 ← GetP(1)
28:     for i in 0:2^T do
29:         T_f[i] ← ⌊e^((m+2^(k−T)*i)/2^P) * 2^P1⌋
30:         T_g[i] ← ⌊e^((2^(k−2*T)*i)/2^P) * 2^P2⌋
31:     return T, P1, P2, k
```

We set the following parameters by brute force: the bitwidth $\mathcal{B}$, i.e., the number of bits assigned to each variable and the maxscale $\mathcal{P}$. The time taken for this exploration is dependent on the size of the training set and is usually within a couple of minutes. Although we would like to set (m, M), the range of inputs for e^x, via brute force as well, this would increase the enumeration space significantly. Therefore, (m, M) are set by profiling. In particular, we run the SeeDot program using floating-point arithmetic on the training set. We monitor the calls to exponentiation and select a small range in which *most* (more than 90%) of the inputs lie. By excluding the outliers, this process produces satisfactory implementations.

6 Accelerating SeeDot Using FPGA

Field Programmable Gate Arrays (FPGA) are re-configurable chips that can be used to build custom hardware accelerators for important applications. FPGAs offer better performance

$$\frac{\kappa \vdash e_1 \rightarrow (C_1, \eta_1, \mathsf{P}_1) \qquad \kappa[x \mapsto (\eta_1, \mathsf{P}_1)] \vdash e_2 \rightarrow (C_2, \eta_2, \mathsf{P}_2)}{\kappa \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightarrow ((C_1; C_2;), \eta_2, \mathsf{P}_2)}\ C-Let$$

$$\frac{\kappa(x) = (\eta, \mathsf{P})}{\kappa \vdash x \rightarrow (\epsilon, \eta, \mathsf{P})}\ C-Var \qquad \frac{\mathsf{P} = \textsc{GetP}(max(abs(v_1))) \qquad v_2 = \lfloor v_1 * 2^{\mathsf{P}} \rfloor}{\kappa \vdash v_1 \rightarrow ((\eta = v_2;), \eta, \mathsf{P})}\ C-Val$$

$$\frac{\begin{array}{c}\kappa \vdash \mathsf{e}_1 \rightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1) \quad \kappa \vdash \mathsf{e}_2 \rightarrow (\mathsf{C}_2, \eta_2, \mathsf{P}_2) \quad (I, J) = dim(\eta_1) \quad (J, K) = dim(\eta_2) \\ (\mathsf{P}_{mul}, \mathsf{S}_{mul}) = \textsc{MulScale}(\mathsf{P}_1, \mathsf{P}_2) \quad (\mathsf{P}_3, \mathsf{S}_{add}) = \textsc{TreeSumScale}(\mathsf{P}_{mul}, J)\end{array}}{\kappa \vdash \mathsf{e}_1 * \mathsf{e}_2 \rightarrow ((\mathsf{C}_1; \mathsf{C}_2; \eta_3 = \textsc{MatMul}(\eta_1, \eta_2, \mathsf{S}_{mul}, \mathsf{S}_{add});), \eta_3, \mathsf{P}_3)}\ C-MatMul$$

$$\frac{\begin{array}{c}\kappa \vdash \mathsf{e}_1 \rightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1) \quad \kappa \vdash \mathsf{e}_2 \rightarrow (\mathsf{C}_2, \eta_2, \mathsf{P}_2) \\ (I, J) = dim(\eta_1) \quad (\mathsf{P}_{mul}, \mathsf{S}_{mul}) = \textsc{MulScale}(\mathsf{P}_1, \mathsf{P}_2) \quad (\mathsf{P}_3, \mathsf{S}_{add}) = \textsc{TreeSumScale}(\mathsf{P}_{mul}, J)\end{array}}{\kappa \vdash \mathsf{e}_1 \times \mathsf{e}_2 \rightarrow ((\mathsf{C}_1; \mathsf{C}_2; \eta_3 = \textsc{SparseMatMul}(\eta_1, \eta_2, \mathsf{S}_{mul}, \mathsf{S}_{add});), \eta_3, \mathsf{P}_3)}\ C-SparseMatMul$$

$$\frac{\kappa \vdash \mathsf{e}_1 \rightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1) \quad \kappa \vdash \mathsf{e}_2 \rightarrow (\mathsf{C}_2, \eta_2, \mathsf{P}_2) \quad (I, J) = dim(\eta_1) \quad \mathsf{P}_2 >= \mathsf{P}_1 \quad (\mathsf{P}_3, \mathsf{S}_{add}) = \textsc{AddScale}(\mathsf{P}_1)}{\kappa \vdash \mathsf{e}_1 + \mathsf{e}_2 \rightarrow ((\mathsf{C}_1; \mathsf{C}_2; \eta_3 = \textsc{MatAdd}(\eta_1, \eta_2, \mathsf{P}_2 - \mathsf{P}_1, \mathsf{S}_{add});), \eta_3, \mathsf{P}_3)}\ C-MatAdd$$

$$\frac{\kappa \vdash \mathsf{e} \rightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1) \quad (T_f, T_g, \mathsf{P}_{11}, \mathsf{P}_{12}, k) = \textsc{ExpTable}(\mathsf{P}_1, m, M) \quad (\mathsf{P}_2, \mathsf{S}_{mul}) = \textsc{MulScale}(\mathsf{P}_{11}, \mathsf{P}_{12})}{\kappa \vdash \texttt{exp}(\mathsf{e}) \rightarrow ((\mathsf{C}_1; \eta_2 = \textsc{Exp}(\eta_1 - m, T_f, T_g, \mathsf{S}_{mul}, k);), \eta_2, \mathsf{P}_2)}\ C-Exp$$

$$\frac{\kappa \vdash \mathsf{e} \rightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1)}{\kappa \vdash \texttt{argmax}(\mathsf{e}) \rightarrow ((\mathsf{C}_1; \eta_2 = \textsc{ArgMax}(\eta_1);), \eta_2, 0)}\ C-ArgMax$$

Figure 4. Compilation rules

and power-efficiency compared to general-purpose processors; Unlike Application-Specific Integrated Circuits (ASICs), FPGAs can be *reprogrammed* to handle updates to algorithms. These features make them a natural choice for IoT devices. To this end, we explore the potential for accelerating SeeDot programs using FPGAs.

6.1 SeeDot to FPGA: Overview

Traditionally, FPGA programmers write code in a Hardware Description Languages (HDL) like VHDL [34] or Verilog [35]. This process requires significant expertise in digital design and is extremely time consuming. To improve programmer productivity, FPGA vendors have developed High-Level Synthesis (HLS) tools that allow programmers to write code in a language like C which is then translated to Verilog code.

At a high-level, we translate a SeeDot program to an input for an HLS tool. A naïve approach is to directly use SeeDot-generated fixed-point code as input. Although this approach significantly outperforms Arduino Uno (on an FPGA of comparable power consumption), it does not fully exploit the optimization opportunities presented by FPGAs. To address this, we present two optimizations.

6.2 Optimizations to Improve FPGA Utilization

Figure 5 shows the flowchart for compiling a SeeDot program to FPGA using two optimizations: 1) accelerating Sparse-Matrix-Vector (SpMV) multiplication using hand-written Verilog code, and 2) automatically generating parallelization hints to the HLS compiler.

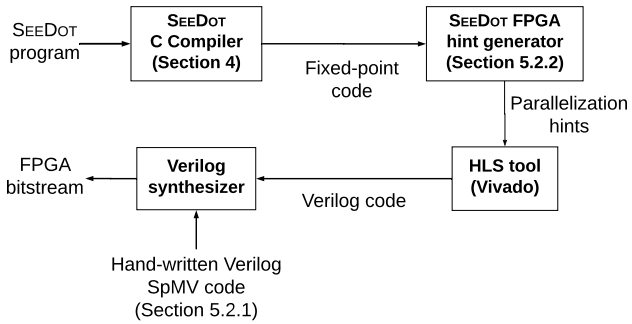
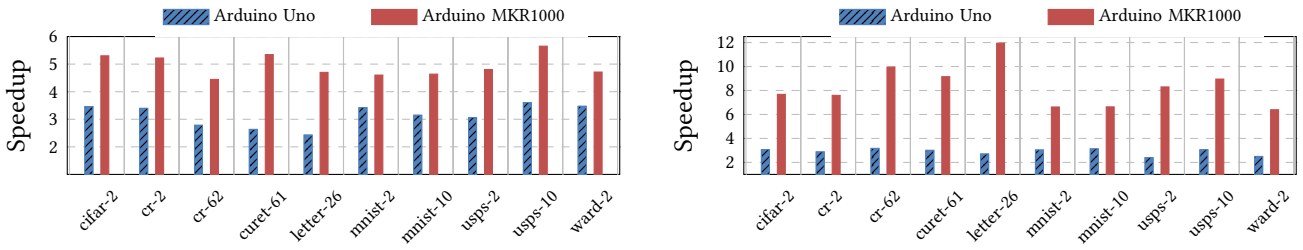


Figure 5. Flowchart for compiling SeeDot to FPGA

6.2.1 Accelerating SpMV Multiplication.

Many ML algorithms use sparse matrices to compress the models [1, 23, 77]. In our evaluation, we observe that SpMV multiplication consumes a significant fraction of the execution time (56% on average). This is the $\times$ operation described in Section 5. In order to accelerate this operation, we implemented it in Verilog. Our implementation creates multiple *processing elements* (PEs) on the FPGA where each PE can perform one fixed-point multiply-accumulate operation per cycle. Then, the columns in the sparse matrix are partitioned and assigned to PEs to compute the result. To avoid workload imbalance across the PEs, a small portion (about a quarter) of matrix columns is retained for dynamic assignment to PEs which complete the work first. The remaining portion is assigned statically. Our implementation of SpMV multiplication is significantly faster, 3×–13.6×, than the version generated by the HLS compiler.



(a) Bonsai model. **(b)** ProtoNN model.

Figure 6. Speedup of SeeDot-generated fixed-point code over hand-written floating-point code.

6.2.2 Hints to the HLS Compiler.

The code generated by the SeeDot compiler is inherently sequential. As a result, when directly fed to the HLS compiler, it generates Verilog code that does not fully utilize the FPGA, and results in low performance. Fortunately, the HLS compiler allows programmers to provide hints to perform optimizations such as loop unrolling, pipe-lining, partitioning, etc. to exploit the parallelism offered by FPGAs. However, prior work [21] has shown that inserting these hints still requires some knowledge of hardware design, which many application developers do not possess. Our second optimization automatically generates these hints for the HLS compiler by adding appropriate annotations to the SeeDot-generated C code. Among our optimizations, we observe that loop unrolling increases the resource requirements, which may sometimes exceed the available budget on the FPGA. Our hint generator uses a simple heuristic that limits the degree of unrolling to ensure that the code fits on the FPGA. This optimization significantly increases resource utilization, thereby improving overall performance.

7 Evaluation

We evaluate SeeDot in three different settings: Arduino boards, FPGAs, and real IoT devices. Through empirical evaluation, we aim to justify the following claims:

- SeeDot-generated fixed-point code is much more efficient than emulating floating-point in software. In particular, we compare the performance of fixed-point and floating-point code on two Arduino boards.
- SeeDot's novel compilation strategy beats state-of-the-art float-to-fixed converters on the task of compiling ML to resource constrained devices. We show that SeeDot-generated code for Arduino Uno has much better performance than commercial MATLAB toolboxes that cost more than $30000 per license to achieve the same task.
- SeeDot generates FPGA implementations that are much more efficient than both microcontroller-based implementations and FPGA implementations obtained from high-level synthesis (HLS) tools directly.
- SeeDot can express various ML inference algorithms. In particular, SeeDot can express recently-published ML classifiers for constrained devices as well as convolution neural networks (CNNs) used in computer vision tasks.
- SeeDot is helpful in the real-world and improves the performance of smart IoT devices that are used in the wild. We consider devices deployed on agricultural farms and pods attached to white canes of persons with visual impairments.

We use two types of Arduino boards for our evaluation: Uno and MKR1000. The Uno has an 8-bit, 16-MHz Atmega328P microcontroller, with 2KB of SRAM and 32KB of read-only flash memory. MKR1000 has more powerful hardware: a 32-bit 48-MHz ARM Cortex-M0+ microcontroller, 32KB of SRAM and 256KB of read-only flash. These devices are much more resource-constrained than the floating-point equipped embedded devices considered in prior work (e.g., Raspberry Pi [53], ARMv7 [28], etc.).

The FPGA device we target is the Xilinx Arty board which has 225KB of on-chip memory, 5200 logic slices consisting of 20800 LUTs and a peak operating frequency of 450MHz. For our experiments, we operate the FPGA at 10MHz. For synthesis we use the Vivado HLS tool provided by Xilinx.

We consider three types of ML classifiers: Bonsai [45], ProtoNN [26], and CNNs [57]. We use standard ML datasets that have been used by these works [26, 45]: cifar [43], character recognition (cr) [16], curet [73], letter [32], mnist [47], usps [33] and ward [76]. The binary classification tasks of these datasets are from [38].

7.1 Arduino Evaluation

We compare SeeDot-generated fixed-point code against floating-point code and MATLAB-generated fixed-point code.

7.1.1 Comparison with Floating-Point

Emulating floating-point operations in software is inefficient. On an Uno, addition and multiplication operations on integers are 11.3× and 7.1× faster than the respective floating-point operations. This results in high performance of SeeDot-generated code. Figure 6a and Figure 6b show

Algorithm 2 CODEGEN PROCEDURES

```
1:  procedure MatAdd(A, B, n, S_add)
2:      P, Q ← dim(A)
3:      var C[P, Q]
4:      for i in 0:P do
5:          for j in 0:Q do
6:              C[i][j] ← (A[i][j] / 2^S_add) + (B[i][j] / 2^(n+S_add))
7:      return C
8:  procedure ArgMax(A)
9:      P, Q ← dim(A)
10:     var maxI ← 0, max ← A[0][0]
11:     for i in 0:P do
12:         for j in 0:Q do
13:             if A[i][j] > max then
14:                 max ← A[i][j], maxI ← j
15:     return maxI
16: procedure MatMul(A, B, S_mul, S_add)
17:     P, Q ← dim(A)
18:     Q, R ← dim(B)
19:     var T[Q], C[P, R]
20:     for i in 0:P do
21:         for j in 0:R do
22:             for k in 0:Q do
23:                 T[k] ← (A[i][j] / 2^(S_mul/2)) * (B[i][j] / 2^(S_mul/2))
24:             C[i][j] ← TreeSum(T, S_add)
25:     return C
26: procedure TreeSum(A, S_add)
27:     n ← dim(A)
28:     var k ← n/2, s ← 1
29:     while k > 1 do
30:         if S_add- - <= 0 then s ← 0
31:         for i in 0:k do
32:             A[i] ← (A[2*i] / 2^s) + (A[2*i+1] / 2^s)
33:         if n%2 != 0 then
34:             A[k] ← A[2*k] / 2^s
35:         n ← (n+1)/2, k ← n/2
36:     return A
37: procedure SparseMatMul(A, B, S_mul, S_add)
38:     P, Q ← dim(A)
39:     Q ← dim(B)
40:     var C[P, 1]
41:     var i_idx ← 0, i_val ← 0
42:     for i in 0:Q do
43:         j ← A.idx[i_idx++]
44:         while j != 0 do
45:             var u ← 2^(S_mul/2)
46:             var t ← (A.val[i_val++] / u) * (B[i] / u)
47:             C[j-1][0] ← C[j-1][0] + t / 2^S_add
48:             j ← A.idx[i_idx++]
49:     return C
50: procedure Exp(x, T_f, T_g, S_mul, k)
51:     var i ← bits_T(x, k), j ← bits_T(x, k − T)
52:     var e ← (T_f[i] / 2^(S_mul/2)) * (T_g[j] / 2^(S_mul/2))
53:     return e
```

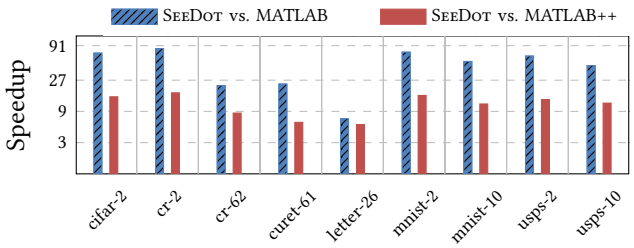
the speedup of SeeDot-generated implementations for Bonsai and ProtoNN respectively over the baseline floating-point implementations from [26, 45]. The size of all models is within 32KB and they fit on both Uno and MKR. The mean speedup for Bonsai is 3.1× on Uno and 4.9× on MKR. For ProtoNN, the mean speedup is 2.9× on Uno and 8.3× on MKR. Thus, fixed-point implementations are much more efficient for both these ML inference algorithms. The average loss in classification accuracy on the *testing* set caused by using fixed-point arithmetic for Bonsai is 0.345% on Uno and 0.127% on MKR. Similarly, for ProtoNN the loss is 1.855% and 0.051% respectively. The MKR implementations are more precise because they use 32-bit integers and the Uno implementations use 16-bit integers. We note that, in most cases, the MKR implementations have better classification accuracy than the corresponding floating-point implementations. The average accuracy loss reported above includes only the cases where the floating-point implementations are more precise.

7.1.2 Comparison with MATLAB

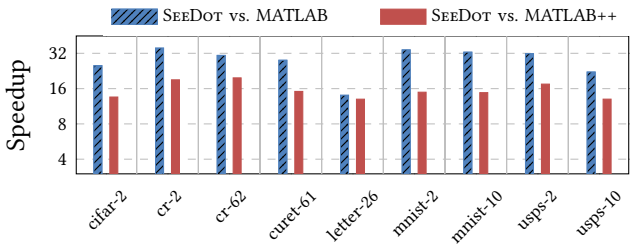
MATLAB provides toolboxes that can compile ML inference algorithms to fixed-point C code for Arduino Uno. This task requires the following MATLAB toolboxes: MATLAB Coder, Embedded Coder, and Fixed-Point Designer. MATLAB uses run-time profiling to assign bitwidths to each variable. Arithmetic operations on these variables are computed over large bitwidths to guard against overflows. The result is then truncated to lower bitwidth before assigning to an output variable. Performing large bitwidth operations on microcontrollers causes huge slowdowns.

Most implementations of ML algorithms support special representations of sparse matrices for performance. However, the Fixed-Point designer toolbox of MATLAB lacks support for sparse matrices which results in the generation of inefficient fixed-point code. On the other hand, SeeDot has language support for sparse matrices. As a side contribution, to be more fair to the techniques being used by MATLAB, we spent significant development effort in adding support for sparse matrices in the MATLAB tool-chain. This improves the performance of MATLAB-generated code by up to 4.8×.

Figure 7a and Figure 7b use the MATLAB-generated fixed-point code for Bonsai and ProtoNN as the baseline and shows the speedups of SeeDot-generated code on Uno. The y-axis is in log scale. MATLAB++ represents MATLAB with sparse matrix support. Without sparse matrix support, the mean speedup is 51× for Bonsai and 28.2× for ProtoNN. With sparse matrix support, the speedups are still quite high with a mean speedup of 11.6× for Bonsai and 15.6× for ProtoNN. We note that, in some cases, the classification accuracy of MATLAB-generated code is extremely poor (similar to that of a purely random classifier). In contrast, SeeDot-generated implementations have comparable accuracy to the floating-point code for all classifiers.



(a) Bonsai model.



(b) ProtoNN model.

Figure 7. Speedup of SeeDot-generated code over MATLAB-generated fixed-point code on an Arduino Uno. MATLAB++ denotes MATLAB with sparse matrix support.

7.2 FPGA Evaluation

This section describes our experience of accelerating ML inference using low-end FPGAs. We compare SeeDot generated FPGA implementations with the Uno implementations described in the previous section and handwritten floating-point Vivado HLS C-code.

The evaluation on Bonsai models is shown in Figure 8. The results for ProtoNN are similar and are omitted. The FPGA implementations are bit-wise equivalent to the Uno implementations and thus have the same classification accuracy. We observe that the FPGA implementations are 18.7×–211.3× faster than the Uno implementations. To the best of our knowledge, this is the first empirical comparison of ML inference running on constrained devices versus low-end FPGAs. We observe that acceleration using FPGAs can provide significant benefits in this setting. We also observe that the optimizations described in Section 6 have a significant impact on performance. In particular, the SeeDot-generated FPGA implementation with hints and-hand optimized sparse multiplication are 5.2×–9.8× faster than the floating-point FPGA-based implementations generated by HLS.

At the clock frequency of 10MHz that we use in this evaluation, both a floating-point operation and a fixed-point operation take one clock cycle. The benefits at higher clock frequencies are even larger. At higher frequencies, floating-point operations consume multiple cycles whereas fixed-point operations can still be completed within a single cycle. For instance, consider fixed-point ProtoNN code generated by SeeDot with all optimizations described in Section 6 disabled. At low clock frequencies, we expect this fixed-point code to be slower as it performs more number of operations than a floating-point code. Indeed, at 10MHz, this code is about 2× slower than the HLS implementation (Figure 9). However, at 100MHz, the same code is about 1.5× faster.

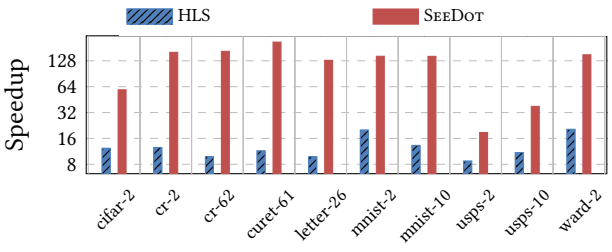


Figure 8. Performance of FPGA implementations generated by HLS and SeeDot (with our optimizations) for Bonsai, with SeeDot-generated Uno implementations as baseline

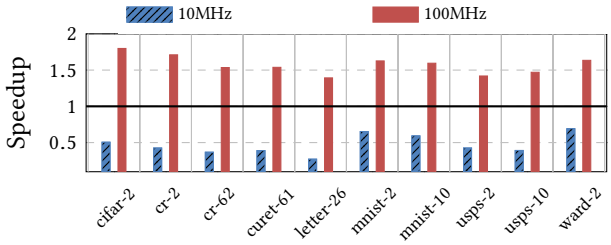


Figure 9. Performance of FPGA implementations for ProtoNN generated by SeeDot (without our optimizations) at 10MHz and 100MHz, with HLS as baseline.

7.3 Expressiveness

SeeDot can express a variety of ML models. Bonsai is based on decision trees and ProtoNN is based on nearest-neighbors. These can be expressed in 11 lines and 5 lines of SeeDot code respectively. Since SeeDot provides language support for standard operations in linear algebra, we believe that it can express most ML inference algorithms.

To demonstrate the expressiveness of SeeDot, we implemented a convolution neural network (CNN) [57] in SeeDot. CNNs are widely used in computer vision and are being deployed on embedded devices for various applications: pedestrian detection [69], enhancing driver safety [52, 65], traffic management [48, 79], etc. For our evaluation, we use LeNet [70], a popular CNN architecture, which passes an

Model size	Bitwidth	Accuracy loss	Speedup
50K	16	2.45%	2.5×
50K	32	0.00%	3.3×
105K	16	1.16%	∞

Table 1. Comparison of SeeDot-generated code with hand-written floating-point code for LeNet models on MKR1000.

input image through a number of convolution layers followed by a number of fully-connected layers. We trained LeNet models for the CIFAR-10 dataset and deployed them on an MKR. Since these models are large, they did not fit on an Uno. CIFAR-10 requires labeling RGB images with ten possible labels (e.g., cat, dog, truck, etc.) and is one of the most widely used datasets in computer vision [17, 29, 44]. LeNet can be expressed in 10 lines of SeeDot code. Whereas, the hand-written C code is several hundred lines long.

Table 1 summarizes the results on two LeNet models with different sizes. On the smaller model with 50K parameters, SeeDot-generated 16-bit fixed-point code performs 2.5× better than the baseline floating-point code with a small loss in accuracy (2.45%). To obtain better precision, we tested the model with SeeDot-generated 32-bit fixed-point code which has no accuracy loss and performs 3.3× better. For testing the larger network with 105K parameters, the floating-point model is too large to fit on a MKR. Therefore, we measured its accuracy by running it on an x86 processor. In contrast, the fixed-point model can fit on a MKR. To the best of our knowledge, this is the first implementation of an ML inference algorithm running on such a small microcontroller that provides high accuracy (above 70%) on a practical computer vision task. Therefore, we believe that SeeDot can facilitate new ML applications in the future.

7.4 Real-world Case Studies

Our evaluation till now has focused on standard ML datasets. Next, we show how SeeDot improves performance of ML inference algorithms that have been deployed on real IoT devices using two case studies.

7.4.1 Farm sensors

Chakraborty et al. [11] have recently deployed 20 IoT devices on a few agricultural farms to enable data-driven farming. Each device contains multiple sensors, deployed at different soil depths, to collect soil moisture and soil temperature data. Given the likelihood of sensor failures, it is necessary to ensure the fidelity of the collected data. Hence, the device contains an Arduino Uno which runs ML inference to detect whether some sensor has malfunctioned. Since the farms are large, the devices neither have network connectivity nor are connected to power supplies. Thus, the devices need to be power efficient and use constrained hardware like the Uno.

The deployed devices use a floating-point ProtoNN classifier that can detect sensor failures with an accuracy of 96.9%. For this classifier, SeeDot-generated code uses 32-bit integers and has an accuracy of 98.0%, which is higher than the floating-point classifier. Moreover, SeeDot generated code is 1.6× faster than the floating-point implementation.

7.4.2 Interactive cane

Gesturepod [60] is an IoT device that can be attached to white canes carried by people with visual impairments (VIs). When a person makes a gesture with the cane, e.g., taps it twice on the ground, the pod uses ML to recognize the gesture and communicates it to a smart-phone app. The smart-phone can then perform a task, e.g, read the recent notifications. User studies with 12 people with VIs have shown that the pod can recognize gestures with high accuracy and help complete smart-phone tasks up to 9 times faster [60].

The classification accuracy of the floating-point ProtoNN model used by the pod is 99.86%, which is comparable to the 99.79% accuracy of SeeDot's 16-bit fixed-point implementation. The pod uses a MKR1000 on which SeeDot-generated code is 9.8× faster than the deployed implementation.

8 Related Work

SeeDot can be considered as an approximate computing framework [3, 62, 63, 68, 80]. However, the prior frameworks do not deal with fixed-point arithmetic and their techniques are complementary.

SeeDot is a compiler that translates Real expressions to fixed-point code. The previous compilers to achieve this task are too restrictive to be useful for ML tasks. In particular, Darulova et al. [13–15] can only express arithmetic over scalar variables and provide no support for matrix operations.

There are a lot of tools in digital signal processing (DSP) that convert floating-expressions to fixed-point [2, 4, 6, 8, 50, 55, 74]. MATLAB provides commercial tools that are based on this line of work. Our evaluation demonstrates that these techniques are far from ideal in the setting of compiling ML inference algorithms to low-end microcontrollers. These tools use high-bitwidth operations to compute intermediate results. Unlike DSPs, microcontrollers do not have hardware support for such operations and hence such operations cause huge slowdowns. However, MATLAB has some powerful techniques that are absent in SeeDot. E.g., it assigns various bitwidths to variables in the program [7, 67]. Incorporating these techniques in SeeDot will improve the performance of the generated code even further. SeeDot also uses a naïve rounding strategy. More intelligent rounding strategies are certainly possible (e.g., [27]) and would improve accuracy.

Fixed-point implementations have been developed by hand for several important algorithms by Bocchieri [71]. The Bonsai and ProtoNN papers also describe some fixed-point optimizations that were performed by hand [26, 45]. However, such hand-optimized implementation are not maintainable and quickly face bit rot: a small change in the input format can affect the scales of a large number of variables.

Developing ML classifiers for constrained hardware is an active research area [26, 30, 45, 46, 78]. There have been several efforts that explore acceleration of machine learning inference on FPGAs [10, 18, 19, 22, 66]. The closest work to SeeDot is by Guan et al. [24] which is a framework to accelerate DNN inference on FPGAs using a hybrid RTL + HLS approach. However, unlike SeeDot, these efforts focus on floating-point models. Quantization, if necessary, needs to be performed by the user. SeeDot automatically generates fixed-point FPGA implementations that accelerate ML inference on low-end FPGAs that lack floating-point support.

An ML programmer who attempts to use high-level synthesis tools such as Intel's FPGA SDK for OpenCL and Xilinx's Vivado HLS faces a steep learning curve. The SeeDot compiler hides this complexity and makes FPGAs more accessible to an end-user. For example, Embedded FPGAs (eFPGAs) are gaining traction in real-world embedded systems. They are present in various domains such as bio-medical [72], computer vision [41], traffic monitoring [79], and industrial safety [12]. We believe that SeeDot would be useful in making eFPGAs more accessible to programmers who are inexperienced in digital design.

9 Conclusion

SeeDot is a DSL for specifying ML inference algorithms that improves productivity of ML programmers. In particular, state-of-the-art ML algorithms can be concisely expressed in a few lines of SeeDot program which is an order of magnitude shorter than the corresponding C code. The SeeDot compiler uses novel techniques to generate fixed-point code which is both precise and efficient. E.g., it auto-tunes key parameters using the training set and computes exponentiation entirely in fixed-point. On resource-constrained microcontrollers, SeeDot-generated code is 2.4×–11.9× faster than the handwritten floating-point implementations and has comparable accuracy on the testing set. In fact, for MKR1000, SeeDot-generated 32-bit fixed-point implementations have better classification accuracy than floating-point implementations on most benchmarks. SeeDot outperforms commercial tool-chains that generate fixed-point code by huge margins. In particular, SeeDot-generated code is 5.7×–82.2× faster than MATLAB toolboxes that cost over $30000 per license. We plan to release the SeeDot compiler publicly as free software.

We demonstrate the benefits of accelerating ML inference using low-end FPGAs; the FPGA-based implementations are 18.7×–211.3× faster than the microcontroller-based implementations. Furthermore, the optimizations in the SeeDot FPGA backend lead to code which is 5.2×–9.8× faster than FPGA implementations obtained by passing floating-point code through high-level synthesis tools.

We demonstrate the first implementation of a CNN running on a resource-constrained microcontroller with no floating-point unit. We show that SeeDot-generated code can be 9.8× faster than the production code running on real IoT devices that have been deployed in the wild. We believe that SeeDot can facilitate new ML applications by significantly reducing the cost of performing inference on IoT devices and we will explore this direction in the future.

References

[1] T. Araki. 2017. Accelerating Machine Learning on Sparse Datasets with a Distributed Memory Vector Architecture. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*. 112–121. https://doi.org/10.1109/ISPDC.2017.21

[2] 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*. 70.

[3] 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*. 198–209.

[4] 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.* 263–264. https://doi.org/10.1109/FPGA.2003.1227262

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

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

[7] P. Belanovic and M. Rupp. 2005. Automated floating-point to fixed-point conversion with the fixify environment. In *16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*. 172–178. https://doi.org/10.1109/RSP.2005.15

[8] 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*. 13–22.

[9] Pablo Samuel Castro, Daqing Zhang, and Shijian Li. 2012. Urban traffic modelling and prediction using large scale taxi GPS traces. In *International Conference on Pervasive Computing*. Springer, 57–72.

[10] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 7.

[11] 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)*.

[12] L. M. Contreras-Medina, R. J. Romero-Troncoso, J. R. Millan-Almaraz, and C. Rodriguez-Donate. 2008. FPGA based multiple-channel vibration analyzer embedded system for industrial applications in automatic failure detection. In *2008 International Symposium on Industrial Embedded Systems*. 229–232. https://doi.org/10.1109/SIES.2008.4577705

[13] 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*. 235–248.

[14] 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

[15] 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

[16] 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*. 273–280.

[17] Peter Eckersley and Yomna Nasser. [n. d.]. AI Progress Measurement | Electronic Frontier Foundation. https://www.eff.org/ai/metrics. (Accessed on 08/04/2018).

[18] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. 2009. CNP: An FPGA-based Processor for Convolutional Networks. In *International Conference on Field Programmable Logic and Applications*. 32–37. https://doi.org/10.1109/FPL.2009.5272559

[19] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA.2018.00012

[20] Mario Gerla, Eun-Kyu Lee, Giovanni Pau, and Uichin Lee. 2014. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE, 241–246.

[21] N.P. Ghanathe, A. Madorsky, H. Lam, D.E. Acosta, A.D. George, M.R. Carver, Y. Xia, A. Jyothishwara, and M. Hansen. 2017. Software and firmware co-development using high-level synthesis. *Journal of Instrumentation* 12, 01 (2017), C01083. http://stacks.iop.org/1748-0221/12/i=01/a=C01083

[22] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. 2014. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 696–701. https://doi.org/10.1109/CVPRW.2014.106

[23] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk. 2015. Accelerating SpMV on FPGAs by Compressing Nonzero Values. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 64–67. https://doi.org/10.1109/FCCM.2015.30

[24] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 152–159. https://doi.org/10.1109/FCCM.2017.25

[25] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems* 29, 7 (2013), 1645–1660.

[26] 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*. 1331–1340.

[27] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 1737–1746.

[28] Karen Zita Haigh, Allan M. Mackay, Michael Cook, and Li Lin. 2015. Machine Learning for Embedded Systems : A Case Study.

[29] Ben Hamner. [n. d.]. Popular Datasets Over Time | Kaggle. https://www.kaggle.com/benhamner/popular-datasets-over-time/code. (Accessed on 08/04/2018).

[30] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015).

[31] Moeen Hassanalieragh, Alex Page, Tolga Soyata, Gaurav Sharma, Mehmet Aktas, Gonzalo Mateos, Burak Kantarci, and Silvana Andreescu. 2015. Health monitoring and management using Internet-of-Things (IoT) sensing with cloud-based processing: Opportunities and challenges. In *2015 IEEE international conference on services computing (SCC)*. IEEE, 285–292.

[32] 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.

[33] Jonathan J. Hull. 1994. A database for handwritten text recognition research. *IEEE Transactions on pattern analysis and machine intelligence* 16, 5 (1994), 550–554.

[34] IEEE. 2000. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2000* (2000), i–290. https://doi.org/10.1109/IEEESTD.2000.92297

[35] IEEE. 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 01–560. https://doi.org/10.1109/IEEESTD.2006.99495

[36] IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935

[37] Zhanlin Ji, Ivan Ganchev, Máirtín O'Droma, Li Zhao, and Xueji Zhang. 2014. A cloud-based car parking middleware for IoT-based smart cities: Design and implementation. *Sensors* 14, 12 (2014), 22372–22393.

[38] Cijo Jose, Prasoon Goyal, Parv Aggrwal, and Manik Varma. 2013. Local deep kernel learning for efficient non-linear SVM prediction. In *International conference on machine learning*. 486–494. http://manikvarma.org/code/LDKL/download.html

[39] Jongbin Jung, Connor Concannon, Ravi Shroff, Sharad Goel, and Daniel G Goldstein. 2017. Simple rules for complex decisions. *arXiv preprint arXiv:1702.04690* (2017).

[40] Hillol Kargupta. 2010. Onboard driver, vehicle and fleet data mining. US Patent 7,715,961.

[41] A. S. Khalil, M. Shalaby, and E. Hegazi. 2017. A hardware design and implementation for accelerating motion detection using (System On Chip) SOC. In *2017 12th International Conference on Computer Engineering and Systems (ICCES)*. 411–416. https://doi.org/10.1109/ICCES.2017.8275343

[42] JM Ko and YQ Ni. 2005. Technology developments in structural health monitoring of large-scale bridges. *Engineering structures* 27, 12 (2005), 1715–1725.

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

[44] Alex Krizhevsky and G Hinton. 2010. Convolutional deep belief networks on CIFAR-10. *Unpublished manuscript* 40, 7 (2010).

[45] 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*. 1935–1944.

[46] Matt J. Kusner, Stephen Tyree, Kilian Q. Weinberger, and Kunal Agrawal. 2014. Stochastic Neighbor Compression. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing,*

China, 21-26 June 2014. 622–630.

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

[48] Xiaolei Ma, Zhuang Dai, Zhengbing He, Jihui Ma, Yong Wang, and Yunpeng Wang. 2017. Learning traffic as images: a deep convolutional neural network for large-scale transportation network speed prediction. *Sensors* 17, 4 (2017), 818.

[49] Nicolas Maisonneuve, Matthias Stevens, Maria E Niessen, Peter Hanappe, and Luc Steels. 2009. Citizen noise pollution monitoring. In *Proceedings of the 10th Annual International Conference on Digital Government Research: Social Networks: Making Connections between Citizens, Data and Government*. Digital Government Society of North America, 96–103.

[50] 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)*. ACM, New York, NY, USA, 270–276. https://doi.org/10.1145/581630.581674

[51] Luz Elena Y Mimbela and Lawrence A Klein. 2000. Summary of vehicle detection and surveillance technologies used in intelligent transportation systems.

[52] Pavlo Molchanov, Shalini Gupta, Kihwan Kim, and Jan Kautz. 2015. Hand gesture recognition with 3D convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 1–7.

[53] A. Monteiro, M. de Oliveira, R. de Oliveira, and T. da Silva. 2018. Embedded application of convolutional neural networks on Raspberry Pi for SHM. *Electronics Letters* 54, 11 (2018), 680–682. https://doi.org/10.1049/el.2018.0877

[54] Subhas Chandra Mukhopadhyay. 2015. Wearable sensors for human activity monitoring: A review. *IEEE sensors journal* 15, 3 (2015), 1321–1330.

[55] 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.

[56] John V Oldfield and Richard C Dorf. 1995. *Field-programmable gate arrays: reconfigurable logic for rapid prototyping and implementation of digital systems*. Wiley.

[57] Keiron O'Shea and Ryan Nash. 2015. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458* (2015).

[58] Alexandros Pantelopoulos and Nikolaos G Bourbakis. 2010. A survey on wearable sensor-based systems for health monitoring and prognosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 40, 1 (2010), 1–12.

[59] Sungmee Park and Sundaresan Jayaraman. 2003. Enhancing the quality of life through wearable technology. *IEEE Engineering in medicine and biology magazine* 22, 3 (2003), 41–48.

[60] 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/

[61] Aswin Raghavan, Mohamed R. Amer, and Sek M. Chai. 2017. BitNet: Bit-Regularized Deep Neural Networks. *CoRR* abs/1708.04788 (2017).

[62] 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)*. ACM, New York, NY, USA, Article 27, 12 pages. https://doi.org/10.1145/2503210.2503296

[63] 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.

[64] Nicol N Schraudolph. 1999. A fast, compact approximation of the exponential function. *Neural Computation* 11, 4 (1999), 853–862.

[65] Pierre Sermanet and Yann LeCun. 2011. Traffic sign recognition with multi-scale convolutional networks. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE, 2809–2813.

[66] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From High-level Deep Neural Models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 17, 12 pages. http://dl.acm.org/citation.cfm?id=3195638.3195659

[67] Changchun Shi and R. W. Brodersen. 2003. An automated floating-point to fixed-point conversion methodology. In *Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on*, Vol. 2. II–529–32 vol.2. https://doi.org/10.1109/ICASSP.2003.1202420

[68] 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)*. ACM, New York, NY, USA, 124–134. https://doi.org/10.1145/2025113.2025133

[69] Mate Szarvas, Akira Yoshizawa, Munetaka Yamamoto, and Jun Ogata. 2005. Pedestrian detection with convolutional neural networks. In *Intelligent vehicles symposium, 2005. Proceedings. IEEE*. IEEE, 224–229.

[70] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Computer Vision and Pattern Recognition (CVPR)*. http://arxiv.org/abs/1409.4842

[71] Z. Tan and B. Lindberg (Eds.). 2008. *Automatic Speech Recognition on Mobile Devices and over Communication Networks*. Springer, London. https://doi.org/10.1007/978-1-84800-143-5

[72] N. M. Thamrin, M. A. Haron, and Fazlina Ahmat Ruslan. 2011. A field programmable gate array implementation for biomedical system-on-chip (SoC). In *2011 IEEE 7th International Colloquium on Signal Processing and its Applications*. 187–191. https://doi.org/10.1109/CSPA.2011.5759870

[73] 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.

[74] 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/

[75] Stephen Wolfram et al. 1996. *Mathematica*. Cambridge university press Cambridge.

[76] 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, 436–443.

[77] L. Yavits and R. Ginosar. 2018. Accelerator for Sparse Machine Learning. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018), 21–24. https://doi.org/10.1109/LCA.2017.2714667

[78] Kai Zhong, Ruiqi Guo, Sanjiv Kumar, Bowei Yan, David Simcha, and Inderjit S. Dhillon. 2017. Fast Classification with Binary Prototypes. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*.

1255–1263.

[79] Y. Zhou, Z. Chen, and X. Huang. 2016. A system-on-chip FPGA design for real-time traffic signal recognition system. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1778–1781. https://doi.org/10.1109/ISCAS.2016.7538913

[80] 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)*. ACM, New York, NY, USA, 441–454. https://doi.org/10.1145/2103656.2103710