Sound Mixed Fixed-Point Quantization of Neural Networks

DEBASMITA LOHAR, MPI-SWS, Saarland Informatics Campus, Germany
CLOTHILDE JEANGOUDOUX, MPI-SWS, Germany
ANASTASIA VOLKOVA, Nantes Université, France
EVA DARULOVA, Uppsala University, Sweden

Neural networks are increasingly being used as components in safety-critical applications, for instance, as controllers in embedded systems. Their formal safety verification has made significant progress but typically considers only idealized real-valued networks. For practical applications, such neural networks have to be quantized, i.e., implemented in finite-precision arithmetic, which inevitably introduces roundoff errors. Choosing a suitable precision that is both guaranteed to satisfy a roundoff error bound to ensure safety and that is as small as possible to not waste resources is highly nontrivial to do manually. This task is especially challenging when quantizing a neural network in fixed-point arithmetic, where one can choose among a large number of precisions and has to ensure overflow-freedom explicitly.

This paper presents the first sound and fully automated mixed-precision quantization approach that specifically targets deep feed-forward neural networks. Our quantization is based on mixed-integer linear programming (MILP) and leverages the unique structure of neural networks and effective over-approximations to make MILP optimization feasible. Our approach efficiently optimizes the number of bits needed to implement a network while guaranteeing a provided error bound. Our evaluation on existing embedded neural controller benchmarks shows that our optimization translates into precision assignments that mostly use fewer machine cycles when compiled to an FPGA with a commercial HLS compiler than code generated by (sound) state-of-the-art. Furthermore, our approach handles significantly more benchmarks substantially faster, especially for larger networks.

CCS Concepts: • **Computer systems organization → Embedded software**; • **Computing methodologies → Neural networks**; • **Theory of computation → Rounding techniques**; **Constraint and logic programming**; • **Software and its engineering → Software performance**; **Formal software verification**.

Additional Key Words and Phrases: fixed-point arithmetic, mixed precision, mixed integer linear programming

ACM Reference Format:
Debasmita Lohar, Clothilde Jeangoudoux, Anastasia Volkova, and Eva Darulova. 2023. Sound Mixed Fixed-Point Quantization of Neural Networks. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2023), 25 pages. https://doi.org/10.1145/3609118

1 INTRODUCTION

Neural networks (NN) are increasingly being considered as components in safety-critical systems. For example, neural network controllers have been shown to be effective for a variety of closed-loop systems, including simple car and airplane models, adaptive cruise control, and aircraft collision

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2023.
Authors' addresses: Debasmita Lohar, dlohar@mpi-sws.org, MPI-SWS, Saarland Informatics Campus, Germany; Clothilde Jeangoudoux, clothilde.jeangoudoux@mpi-sws.org, MPI-SWS, Germany; Anastasia Volkova, anastasia.volkova@univ.nantes.fr, Nantes Université, France; Eva Darulova, eva.darulova@it.uu.se, Uppsala University, Sweden.

1539-9087/2023/1-ART1
https://doi.org/10.1145/3609118

avoidance [15, 16, 34]. Ensuring their correctness for safety-critical applications is nontrivial, but several recent approaches and tools can already automatically verify the safety of limited-size but interesting systems [27, 31, 49, 51].

To make such neural networks practical, they need to be safe but also efficient. While NNs are typically trained in high-precision floating-point arithmetic on server-like machines with graphics processing units (GPUs), this high precision can be prohibitively expensive for resource-constrained embedded systems. To obtain efficiency, e.g., in terms of area, latency, or memory usage, the trained NNs are quantized to use, for example, low-precision fixed-point arithmetic [21, 23].

The increased efficiency due to low precision comes at the cost of reduced accuracy of the NN computations since each operation potentially incurs a (larger) roundoff error. To ensure the overall correctness of systems with NNs, we thus need to choose the precision for quantization such that the safety of the overall system can still be guaranteed, i.e., the roundoff error is within some application-specific bound. While the safety proofs of NN controllers, for instance, do not take into account finite-precision roundoff errors directly and assume exact real-valued arithmetic for performance reasons, they typically allow to account for bounded errors stemming from a noisy environment or from inaccurate implementations.

Numerous approaches have been proposed for the quantization of neural networks, demonstrating promising results on standard machine learning benchmarks [21, 23, 37, 44, 47, 48]. However, these techniques are not applicable to safety-critical closed-loop control systems for two primary reasons. First, they are specifically tailored for neural network *classifiers*, focusing on dynamically comparing classification accuracy on specific test datasets. As a result, they cannot handle neural network controllers that do not perform classifications tasks and rather implement *regression* tasks and compute (continuous) control values. Secondly, they are *not sound* which means they cannot guarantee (classification) accuracy for all possible inputs, which is essential for safety-critical systems.

On the other hand, the available tools that do focus on sound quantization or precision tuning are primarily designed for general-purpose arithmetic code [10, 11]. When applied to neural networks, they often provide overly conservative results or are inefficient, as they do not leverage the unique characteristics, structures and optimizations specific to neural networks.

In this paper, we present the first sound and fully automated mixed-precision quantizer for fixed-point arithmetic that specifically targets fully connected feed-forward deep neural networks. We support an arbitrary number of layers with ReLU or linear activation functions and efficiently minimize the number of bits needed to implement a network while guaranteeing a provided error bound wrt. an idealized real-valued implementation. While our implementation targets these networks as they are sufficient for many circumstances in neural network controllers (all our benchmarks are real-world controllers that were used in the context of *sound* verification), the proposed technique is not fundamentally limited to feed-forward networks; it handles networks with (sparse) matrix multiplications and linearized activations.

We focus on *mixed-precision fixed-point* arithmetic because it allows for efficient implementations on resource-constrained systems. Fixed-point arithmetic can be implemented with standard integer operations (in particular it avoids specialized floating-point hardware), and when implemented on configurable hardware such as FPGAs allows operations to use an arbitrary number of bits. Thus, unlike floating-point arithmetic, which is limited to typically 16, 32, or 64 bits of precision, fixed-point arithmetic allows for a larger scope for optimization.

Optimizing mixed-precision fixed-point arithmetic efficiently and accurately is challenging for two main reasons. First, fixed-point arithmetic is fundamentally discrete, and the continuous abstractions that allow to use efficient continuous optimization techniques for floating-point arithmetic [10] are not applicable. Secondly, due to the large choice (of combinations) of different

precisions for individual operations, the search space for quantization is enormous and heuristic search techniques can explore this space only limitedly.

To overcome these challenges, we phrase sound fixed-point quantization of feed-forward neural networks as a mixed integer linear programming (MILP) problem. A naive formulation, however, results in non-linear constraints that are computationally intractable. We show how to use over-approximations to relax the problem to linear constraints that are quickly solvable. Furthermore, we leverage the special structure of NNs to efficiently, yet accurately to optimize the dot product operations by building on an existing technique for their correctly rounded implementation [13].

We implement our approach in a prototype tool called Aster that we will release as open source. Aster takes as input a trained neural network written in a small real-valued domain-specific language, a specification of the possible inputs, and an error bound on the result, and outputs a mixed fixed-point precision assignment that is guaranteed to satisfy the error bound (wrt. the real-valued input) and to minimize a (customizable) cost function.

We evaluate Aster on NN embedded controller benchmarks used for verification from the literature [22, 34, 40] and compare it with an existing sound quantizer [11]. Our results show that Aster can generate feasible implementations for significantly more benchmarks. We also show that for most of our benchmarks, Aster's implementations take fewer machine cycles when compiled by Xilinx Vivado HLS [52] for an FPGA. In addition, Aster is substantially faster, especially for larger networks with thousands of parameters, with improvements in optimization time on average of $\sim 67\%$.

Contributions. To summarize, in this paper we present:

(i) a novel MILP-based mixed fixed-point quantization approach that guarantees a given roundoff error bound,
(ii) an experimental comparison against the state-of-the-art to demonstrate the effectiveness of our approach on a benchmark set of neural network controllers, and
(iii) a prototype tool called Aster available on Zenodo: https://doi.org/10.5281/zenodo.8123416.

2 OVERVIEW AND BACKGROUND

Consider a unicycle model of a car [34] that models the dynamics of a car with 4 parameter variables—the Cartesian coordinates (x, y), the speed, and the steering angle. We use a neural network that was trained as a controller for this model as our initial example. It is a fully connected feed-forward neural network with 1 hidden layer. The network is fed as input a 4-parameter vector, denoted by $\overline{x}_0 = \begin{bmatrix} x_0^1 & x_0^2 & x_0^3 & x_0^4 \end{bmatrix}$ where each parameter is constrained by real-valued intervals: $x_0^1 \in [-0.6, 9.55]$, $x_0^2 \in [-4.5, 0.2]$, $x_0^3 \in [-0.6, 2.11]$, and $x_0^4 \in [-0.3, 1.51]$, each specifying valid input values to the network. The inputs are propagated through each layer by successive application of the dot product operations, bias additions and activation functions:

$$\text{layer 1:} \quad \overline{x}_1 = ReLU(W_1 \cdot \overline{x}_0 + \overline{b}_1) \qquad \text{output:} \quad \overline{x}_2 = Linear(W_2 \cdot \overline{x}_1 + \overline{b}_2) \tag{1}$$

$\overline{x}_1$ is the output of the first layer, which is then fed as input to the next, output layer. The output of the overall network is $\overline{x}_2$. Each layer is parameterized by a weight matrix W_i and a bias vector $\overline{b_i}$:

$$W_1 = \begin{bmatrix} -0.037 & \cdots & 0.129 \\ -0.003 & \cdots & 0.099 \\ -0.128 & \cdots & 0.047 \\ 0.045 & \cdots & -0.166 \end{bmatrix}, \ \overline{b}_1 = \begin{bmatrix} 0.028 & \cdots & 0.342 \end{bmatrix}, \quad W_2 = \begin{bmatrix} 0.052 & 0.137 \\ \cdots & \cdots \\ -0.200 & 0.154 \end{bmatrix}, \ \overline{b}_2 = \begin{bmatrix} 0.273 & 0.304 \end{bmatrix}$$

While the network has only a single hidden layer, it has 500 neurons (elided above) and thus results in a non-trivial size of the overall network.

To be deployed in a safety-critical system, such a neural network controller needs to be proven safe before implementation, for instance, one may need to prove that the system reaches a set of safe states within a given time window [34]. Since exact reasoning about finite-precision arithmetic does not scale, existing verification techniques assume real-valued parameters and arithmetic operations for the network [27, 31, 49, 51], but can typically deal with *bounded* uncertainties, from the implementation or measurements. We will thus assume that a bound on the output error ϵ is given.

As a controller is primarily executed on resource-constrained hardware, using floating-point arithmetic may be overly expensive, as it requires either additional floating-point processor support or slow software emulations. The alternative is to quantize the NN controller in fixed-point arithmetic.

Fixed-point Quantization. In a fixed-point implementation, all program variables and constants are implemented using integers, and have an (implicit) representation $<Q, \pi>$, consisting of the total word length $Q \in \mathbb{N}$ (overall number of bits including a sign bit), and the position of the binary point $\pi \in \mathbb{N}$ counting from the least significant bit. Arithmetic operations on fixed-point variables can be implemented efficiently using only integer arithmetic and bit-shifting [3], or can use equivalent efficient hardware implementations, e.g. on FPGAs. While fixed-point arithmetic is not standardized, we employ the most commonly used fixed-point representation [55] and review the parts relevant for our paper here.

This representation effectively divides the number of overall bits into an *integer part* $I = Q - \pi - 1$, and a *fractional part* π. The integer part needs enough bits to ensure that it can hold large enough values to not overflow, i.e. the range of representable numbers is $[-2^I, 2^I]$. The fractional part controls the *precision* of a variable or an operation—the larger the number of fractional bits, the more precisely can the value be represented. Assuming that each operation uses truncation as the rounding mode, the maximum roundoff error with π fractional bits is $2^{-\pi}$. In this paper, we use truncation as the rounding mode as it is the most commonly used one (the default with circuit design and synthesis compilers like Xilinx), and more efficient at runtime than e.g. rounding to nearest.

In our example (Equation 1), the input x_0^1 is in the range $[-0.6, 9.55]$. The number of integer bits required to hold this range, i.e. to represent the maximum absolute value 9.55 without overflow, is 5 (including the sign bit). Assuming 32 bits are available for wordlength ($Q = 32$), then we have $(32 - 5) = 27$ bits remaining for the fractional part. Hence, the maximum roundoff error for the input is 2^{-27}.

We need to assign each operation enough integer and fractional bits to guarantee *overflow-freedom* and sufficient *overall accuracy* of the final result. To ensure that the overall roundoff error does not exceed a given bound ϵ, we need to propagate and accumulate the errors of individual operations, which in general happens in nontrivial ways and is challenging to do manually. At the same time, we want to assign only as many bits as are actually needed to not waste resources.

Mixed-Precision Tuning. Using uniform fixed-point precision, i.e. the same word length for all operations, can be suboptimal; if one precision is not enough even at a single point in the program, we need to upgrade *all* operations to the next higher precision. However, not all layers necessarily have the same effect on the overall accuracy. Hence, it can be more resource-efficient to assign different precisions (word lengths) to different operations to achieve a target error bound, and thus implement the model in *mixed* precision.

Existing sound techniques applicable to fixed-point arithmetic rely on a heuristic search that repeatedly evaluates roundoff errors for different precision assignments [11]. While this technique works well for small programs, the repeated global roundoff error analysis quickly becomes expensive as we show in our evaluation in Section 5. Moreover, these techniques are designed for general-purpose straight-line programs, and do not take into account the structure of neural networks. To

apply existing tools, one needs to assign all weight matrix and bias vector elements to individual scalar variables, and unroll all loops over these, resulting in an enormous straight-line expression.

For the unicycle example, only computing the roundoff error for a uniform 32-bit implementation using the state-of-the-art tool Daisy [11] takes 5.91 minutes, and mixed-precision tuning of this example takes *more than 2.7 hours*.

Our Approach. We encode fixed-point precision tuning as a mixed integer linear programming (MILP) problem, and perform several over-approximations to generate a linearized problem from the fundamentally *nonlinear* constraints (see Section 3). Our MILP constraints optimize the number of fractional bits such that overflow-freedom is ensured even in the presence of errors and a cost function is minimized. Our approach is parametric in the cost function to be optimized, but for the purpose of evaluation we follow Daisy's cost function and count the total number of bits needed.

Assuming $\epsilon = 0.001$ as the error bound for the unicycle example, Aster with our MILP-based mixed-precision tuning assigns different precisions (using 18, 19, 20, 21, 24, 30, and 34 bits) to different variables, constants, and operations in just under 50 seconds (i.e. significantly less than the 2.7 hours taken by Daisy). When compiled for an FPGA architecture with Xilinx' Vivado HLS tool, Aster's generated code takes *only 27 machine cycles* to execute, whereas Daisy's generated code, both uniform and mixed-precision, take 178 cycles.

3 MILP-BASED MIXED-PRECISION TUNING

In this work we specifically focus on feed-forward neural networks with 'relu' and 'linear' activation functions that solve regression problems and compute continuous outputs. Such networks may, for instance, be utilized in closed-loop control systems, where they can be proven safe with different techniques [27, 31, 49, 51] that typically assume real-valued arithmetic operations, inputs and constants. Our objective is to minimize the resource usage of the *implemented* networks, while preserving safety by ensuring that computed (control) outputs remain within a specified error bound (that e.g. arises from a safety proof).

Problem Definition. Given a real-valued neural network architecture, ranges of inputs, and a roundoff error bound at the output, the goal is to generate a fixed-point mixed-precision neural network implementation that minimizes the precision of the variables and constants while ensuring that the roundoff errors at the output remain within the specified bound.

Our approach is inspired by successful mixed-precision optimization techniques for *floating-point* programs [10] that phrase precision tuning as an *optimization* problem (which it ultimately is). They rely on the dynamic ranges of floating-point arithmetic that allows to bound floating-point roundoff errors by nonlinear continuous abstractions, which, in turn, enables continuous, purely real-valued optimization techniques for precision tuning.

However, such *continuous* techniques cannot be applied to fixed-point arithmetic programs, because the ranges of individual operations need to be fixed at compile time, i.e. the number of integer bits for each operation has to be determined for all possible inputs up-front. Hence, a continuous abstraction is not possible. Additionally, while floating-point arithmetic supports only a small number of precisions (typically 16, 32, 64 bits), fixed-point arithmetic allows any number of bits to be used for any operation and a more precise encoding is necessary to capture this.

We can capture different numbers of bits precisely by using *integer* constraints. However at the same time, we also need to guarantee that a target error bound is satisfied, and this error is a non-integer. Hence, we choose to encode fixed-point precision tuning as an *Mixed-Integer Linear Programming* (MILP) problem [4]. In MILP, some decision variables are constrained to be integers, and other variables can be real-valued. Integers allow us to directly encode the discrete decisions

about how many bits to use for operations, and real-valued constraints effectively express error constraints.

The primary constraint of our optimization problem guarantees that the total roundoff error remains inside the target error bound. At the same time, we also need to ensure no overflow. Encoding these two conditions together would result in *non-linearity* because of the dot operations in a neural network that perform multiplications. Unfortunately, only linear constraints can be efficiently handled by state-of-the-art MILP solvers; nonlinear mixed integer arithmetic is in general NP-hard, with non-convex problems being undecidable. Hence, we soundly over-approximate and linearize constraints as and when necessary to make our solution efficient and feasible.

There are two primary sources of non-linearity in the optimization problem: 1) computation of ranges of all arithmetic operations in order to ensure that there is no overflow, and 2) optimization of fractional bits for all variables and constants to guarantee the error bound.

To avoid the first case, we pre-compute a sound over-approximation of real-valued ranges for all operations efficiently using interval arithmetic (Section 3.1). From these we compute the integer bits needed to represent the real-valued ranges. However, we still need to ensure that the finite-precision ranges (real-valued ranges + roundoff errors) do not overflow. We encode this as a linear range constraint in our optimization problem (Section 3.2.4), thus ensuring no overflow even in the presence of errors.

For the latter, instead of optimizing for all intermediate variables and constants individually, we treat the dot product as a single operation and encode only the assignment of the fractional bits of the dot product *results*. However, to generate the implementation, we still need to assign precisions to all intermediate variables and constants such that the roundoff errors in intermediate computations do not affect the result. For this, we utilize existing techniques [6, 13] (discussed in more detail in Section 3.3) to determine the number of fractional bits of the intermediate variables in a dot product while guaranteeing an overall error bound.

Provided that the cost function is also linear, we can then encode the precision tuning problem for neural networks with ReLU and linear activation functions with purely linear constraints that can be solved efficiently by state-of-the-art MILP solvers. Other activation functions like sigmoid need to be linearized first before encoding them into our MILP based quantization method. As an additional performance optimization, we consider uniform bit lengths for the operations within each individual layer. That is, within one layer, all dot products will be assigned the same bit length (and similarly for bias and activations), but the bit lengths will vary from layer to layer. This choice is to limit the number of the constraints, but it is not a fundamental limitation of our approach.

Overall, our technique consists of three steps:

(1) computing the integer bits of program variables using interval arithmetic,
(2) optimizing the fractional bits of dot, bias, and activation operations by reducing it to MILP, and
(3) computing the precision of all constants and intermediate variables in dot products.

Our tool Aster performs all these steps fully automatically. We will explain each of these steps in detail next. Though Aster is primarily designed for continuous feed-forward neural network controllers, our proposed approach can be extended to other types of neural networks that have sparse matrices and activation functions that can be piece-wise linearized [54].

Running Example. We will illustrate our approach using the following small (artificial) neural network in Figure 1 as our running example: The input to the network is a vector $\overline{x_0}$ consisting of 2 elements, whose ranges are provided by the user: $x_0^1 = [-10, 10]$ and $x_0^2 = [-5, 5]$. The neural network produces a single output x_2. To generate an implementation of the network, the user needs to provide the network architecture (i.e. the weight matrices and bias vectors, as well as activation functions for each layer) as input to Aster. Additionally, the user specifies the precision of the input

$$\begin{bmatrix} x_1^1 \\ x_1^2 \end{bmatrix} = ReLU\left(\underbrace{\begin{bmatrix} 0.1 & 0.2 \\ 0.2 & 0.15 \end{bmatrix}}_{W_1} \cdot \begin{bmatrix} x_0^1 \\ x_0^2 \end{bmatrix} + \underbrace{\begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix}}_{\overline{b_1}} \right), \quad x_2 = Linear\left(\underbrace{\begin{bmatrix} 0.1 & 0.2 \end{bmatrix}}_{W_2} \cdot \begin{bmatrix} x_1^1 \\ x_1^2 \end{bmatrix} + \underbrace{\begin{bmatrix} 0.5 \end{bmatrix}}_{\overline{b_2}} \right) \tag{2}$$

Fig. 1. Running example for MILP modeling

and if the input is represented *exactly*, which incurs no roundoff error in the input. The precision of inputs is typically known from the specification of sensors or similar in the context of embedded systems, but our approach also supports inputs with initial roundoff errors.

Assume, for the running example, the inputs are exactly represented with 10 bits. The input specification implies that 5 bits are required to represent the maximum absolute value of the ranges, which is 10, leaving 5 bits for the fractional part of the inputs, with no initial roundoff error. The goal is to generate a quantized mixed-precision implementation of the NN such that its cost is minimized and the output error is bounded by a user-specified target error $\epsilon_{target} = 0.1$.

3.1 Step 1: Computing Integer Bits

Aster starts by computing the integer bits for all program variables and constants using a forward data-flow analysis that tracks the real-valued ranges at each abstract syntax tree (AST) node. For this purpose Aster utilizes the widely used and efficient interval arithmetic [42] that computes intervals for each basic arithmetic operation and the activation functions as follows:

$$x \bullet y = [\mathtt{min}(x \bullet y), \mathtt{max}(x \bullet y)], \text{ where } \bullet \in \{+, -, *, /\}$$
$$ReLU(x) = [\mathtt{max}(lo, 0), \mathtt{max}(hi, 0)], \quad Linear(x) = [lo, hi]$$

Here lo and hi denote the lower and upper bounds of the interval of x, respectively. With $ReLU$, the resulting interval is the same as x's directly if both lo and hi are positive; otherwise, it returns 0 for the negative part. After applying the *linear* activation, the interval is the same as the input x.

Given the initial ranges of the inputs, Aster uses interval arithmetic to propagate the intervals through the program and computes an interval for all variables, constants, and intermediate results that soundly over-approximates the real-valued ranges. Finally, Aster uses a function `intBits` [1] that computes, in the two's complement binary representation, the number of integer bits needed for these real-valued ranges such that all possible values can be represented without overflow.

In principle, the finite-precision ranges, i.e. real-valued ranges *with* roundoff errors, may need more integer bits to be represented than the real-valued ranges alone. However, as the errors are usually small, they typically do not affect the integer bits in practice and are thus a good estimate. In addition to this, our MILP constraints (detailed next) ensure that these initial estimates of the integer bits are sufficient to avoid overflow, i.e. these estimates only need to be good approximations. These integer bits are later added with the fractional bits to compute the total word length required to represent each program variable and constant.

For our running example in Figure 1, given the input ranges, Aster computes the real-valued ranges of the first layer as [-3, 3], and [-5, 5] after the dot operation and the bias addition, respectively. From this, Aster determines that 3 and 4 integer bits are needed to represent the signed integer part of the dot operation and bias addition, respectively, at layer 1. It analogously computes the integer bits for all operations in all layers.

[1] `intBit(x) = binary(abs(x).integerPart).numOfDigits`, where it converts the integer part of `x` into 2's complement binary representation and then determines the number of digits in the binary representation.

type	variables	definitions
user inputs	k_i	number of neurons at layer i
	Q_{W_i}	word length of maximum weight at layer i
	Q_{x_0}	maximum word length of inputs
	lo	lower bound on fractional bits π
	hi	upper bound on fractional bits π
pre-computed inputs	R_i^{op}	real-valued ranges of operation op at layer i
	I_i^{op}	number of integer bits of the finite-precision range of operation op at layer i
	A_i^{op}	maximum representable ranges of operation op without overflow at layer i
decision variables	ϵ_i^{prop}	propagation error at layer i
	ϵ_i^{new}	new roundoff error at layer i
	γ_i^{op}	cost of an operation op at layer i
	π_i^{op}	number of fractional bits of op at layer i
other variables	b_i^{op}	indicator variable for operation op at layer i
	opt_i^{op}	optimal variable for π^{op} at layer i

Table 1. Variable notations (op: dot product / bias addition / activation function)

3.2 Step 2: Optimizing Fractional Bits

Next, we reduce the problem of computing the fractional bits of dot product results, the addition of bias, and the activation functions to a mixed integer linear programming problem. We first provide an overview of the relevant variables that we will use to formulate our MILP problem in Table 1. The variables specific to a layer i are referenced along with the subscript i, and op can denote the dot product, the bias addition, or the activation function.

User inputs: The variable k_i, that denotes the number of neurons at layer i, is deduced by Aster from the neural network given by the user. The remaining values are directly provided by the user. The variables Q_{W_i} and Q_{x_0} denote the word length of the maximum weight and the largest input from the input vector $\overline{x}$. The input word length is used as is, however, the word lengths of maximum weights are required in the beginning only to make the optimization problem linear. We later deduce these word lengths automatically considering the result's precision of the dot product, which we will explain in Section 3.3. The user additionally specifies a range for the fractional bits $[lo, hi]$ (one for all operations and layers) that will be considered during optimization.

Pre-computed inputs: The values of these variables are pre-computed by the first step. R_i^{op} and I_i^{op} denote the real-valued range and the number of integer bits required for the finite-precision range of each operation op at layer i. A_i^{op} denotes the maximum representable range of each operation op without overflow at layer i.

Decision variables: The decision variables are those for which the MILP problem will be solved. The variables ϵ_i^{prop} and ϵ_i^{new} are the error variables representing the propagation error and newly introduced error at layer i. These variables constitute the error constraint. γ_i^{op} represents the cost of each operation op at layer i where op ranges over dot, bias and activation function, and is used to formulate the objective function. The variables π_i^{op} denote the fractional bits of all operations at layer i.

Other variables: The other variables are used internally by Aster to linearize non-linear constraints (without over-approximations). The indicator variables are used to select the optimized fractional bit length and the optimal variables are used to store the corresponding errors. We explain these variables in more detail in Section 3.2.5.

$$\textbf{minimize:} \quad \gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha}$$

$$\text{where,} \quad \gamma_i^{dot} = k_i * k_{i-1} * Q_{W_i} * \pi_i^{dot}, \quad \gamma_i^{bias} = \max\left(\pi_i^{dot}, \pi_i^{bias}\right), \quad \gamma_i^{\alpha} = \pi_i^{bias}$$

subject to:

$$C_1(\textbf{error}): \quad \epsilon_n = \sum_{i=1}^{n} (\epsilon_i^{prop} + \epsilon_i^{new}) \leq \epsilon_{target}$$

$$\text{where,} \quad \epsilon_i^{prop} \leq 2^{-\pi_0}, \text{ when } i = 1$$

$$= \left(\max_{j=1}^{k_i} \sum_{l=1}^{k_{i-1}} W_{i,jl}\right) * \epsilon_{i-1}, \text{ when } i > 1$$

$$\epsilon_i^{new} \leq 2^{-\pi_i^{dot}} + 2^{-\pi_i^{bias}}$$

$$C_2(\textbf{range}):$$

$$\forall i \in n, \quad A_i^{dot} \geq R_i^{dot} + \epsilon_i^{prop} + \epsilon_i^{dot}$$

$$\forall i \in n, \quad A_i^{bias} \geq R_i^{bias} + \epsilon_i$$

Fig. 2. The MILP formulation with non-linear constraints

3.2.1 Problem Formulation. Our objective is to minimize a cost (function γ) subject to error and range constraints denoted by C_1 and C_2, where the constraints ensure that the user-provided error bound is respected and that no overflow occurs, respectively. Figure 2 presents the overall formulation of the MILP constraints for optimizing fractional bits. We provide a detailed explanation of each constraint; we emphasize the references to individual constraints in Figure 2 using underlines.

3.2.2 Cost Function. In each layer, there are three operations: first the dot product is computed, then the bias vector is added, and finally, an activation function is applied. We compute the costs of these operations individually and add them up to compute the total cost of each layer. The total cost over all layers is then computed by adding up the costs of n layers (see γ in Figure 2).

This cost function closely follows previous work [33] and computes the total number of bits needed to implement the neural network. Our approach supports other cost functions, e.g. capturing performance or energy, as long as they can be expressed as linear expressions. Since the latter are highly hardware dependent, we implement the cost function used in previous work.

The cost of the dot operation at layer i, denoted by γ_i^{dot}, depends on the number of inputs k_{i-1}, the number of neurons k_i, and the weight matrix W_i of the layer. The dot product at layer i is defined by expanding it into multiplications and additions of the weights and inputs and adding them for all k_i neurons of the layer: $dot_i = \sum_{j=1}^{k_i} \left(\sum_{l=1}^{k_{i-1}} W_{i,jl} \times \overline{x}_{i,l}\right)$.

Recall that our approach takes as input the *maximum* word length of the weights Q_{W_i} for each layer. With the Q_{W_i} and the fractional bits of the dot product π_i^{dot} (decision variable), we can over-approximate the total cost of the dot operation at layer i by multiplying it with Q_{W_i}, the number of neurons of the previous layer (k_{i-1}) and the current layer (k_i) (γ_i^{dot} in Figure 2). For the first layer, instead of k_{i-1} we consider the number of input variables. Note that we only use an estimate of the maximum word length of W, as the actual word lengths of W are not known beforehand, and defining them as variables makes the cost non-linear. We later assign precisions for W from the solutions of the MILP problem, which we explain in Section 3.3.

As the bias vector is only added with the result of the dot product, the cost of this addition denoted as γ_i^{bias} is the maximum of the fractional bits of the result of the addition and fractional

bits of the dot operation (γ_i^{bias} in Figure 2). Technically, the 'max' function is also non-linear. We show how to linearize this function in Section 3.2.5.

Finally, the cost of the activation is a function of π_i^{bias}. We assume ReLU and linear activation functions for the networks. As these two activation functions are linear functions of π_i^{bias}, the cost is the same as π_i^{bias} (γ_i^{α} in Figure 2).

Let us assume for our running example in Figure 1 that $Q_W = 8$ for both layers. The network has 2 input variables, 2 neurons in the first layer and 1 neuron in the output layer. With our formulation, the cost of the whole network is defined as:

$$\underbrace{(2 * 2 * 8 * \pi_1^{dot}) + \max(\pi_1^{dot}, \pi_1^{bias}) + \pi_1^{bias}}_{\text{layer1}} + \underbrace{(2 * 1 * 8 * \pi_2^{dot}) + \max(\pi_2^{dot}, \pi_2^{bias}) + \pi_2^{bias}}_{\text{output}}$$

3.2.3 Error Constraint. The error constraint C_1 in Figure 2 states that the overall roundoff error of the full network ϵ_n needs to be bounded by the user specified error ϵ_{target}. For this constraint, we need to express the roundoff error as a function of the precisions of individual operations.

To compute the roundoff error at each layer i, we need to track the *propagated error* ϵ_i^{prop} from the previous layer and compute the *new roundoff error* ϵ_i^{new} committed by the operations at layer i. The total roundoff error ϵ_n is then defined as the sum of all the errors at all layers.

The initial error ϵ_0 is considered as the propagated error at layer 1. This error ϵ_0 is bounded by the function of fractional bits of inputs (ϵ_1^{prop} in Figure 2) that is determined by the word length of the inputs Q_{x_0} and the integer bits needed to represent the range of the inputs R_{x_0} computed using the intBits function: $\pi_0 = Q_{x_0} - \texttt{intBits}(R_{x_0})$.

The propagation error at layer > 1 depends on the errors from previous layers as well as the absolute magnitudes of the weights (a weight that is bigger than one will magnify an existing error). Thus, the propagation error ϵ_i^{prop} for layer > 1 is defined as the error at the previous layer ϵ_{i-1} multiplied by the sums of the absolute weights of each neurons. In practice, we sum the absolute weights of each neuron (absolute sum over the number of inputs k_{i-1}) and take the maximum result as a factor to amplify the propagation error (maximum over the number of neurons k_i) (see ϵ_i^{prop} in Figure 2).

This is a sound over-approximation of the total propagated error as we assume the maximum magnification of errors for all neurons in the previous layer. For this constraint, Aster computes the maximum values of the weights over the neurons for each layer beforehand and uses them as constants in the optimization problem, thus preserving linearity.

The new roundoff error at layer i is defined as the sum of the errors for the activation function, for the dot computation and the bias addition. The ReLU activation function ($\alpha(x) = \max(0, x)$) and the Linear activation function ($\alpha(x) = x$) do not affect the error. Thus the new error is bounded by $2^{-\pi_i^{dot}}$ and $2^{-\pi_i^{bias}}$, where $\pi_i^{dot} = Q_i^{dot} - I_i^{dot}$ and $\pi_i^{bias} = Q_i^{bias} - I_i^{bias}$. For the dot and bias operation, the error is computed considering the fractional bits of these operations.

This error constraint is non-linear ($2^{-\pi_i^{op}}$). We linearize the error constraint for the optimization *exactly* by considering the user-provided range $[lo, hi]$ for π_i^{op}. We explain the linearization process in Section 3.2.5. We also assume a roundoff error on the *result* of the dot product only, and do not account for the roundoff errors of the individual operations of the dot product in our MILP constraints. We do this to avoid nonlinearity in the constraints, and rely on the fact that the dot product can be computed with faithful rounding up to the chosen format using the technique from de Dinechin et.al. [13]. We explain the details of computing the intermediate formats in Section 3.3.

Let us derive the error constraint for our running example in Figure 1. We considered no input error here: $\epsilon_0 = 0.0$. We first compute the total error in layer 1. This error is then considered as the propagated error for the output layer by magnifying it with the maximum absolute sum of weights:

$\max((0.1+0.2),(0.2+0.15)) = 0.35$. Thus the overall error constraint is (assuming $\epsilon_{target} = 0.1$): $(2^{-\pi_1^{dot}} + 2^{-\pi_1^{bias}}) * 0.35 + (2^{-\pi_2^{dot}} + 2^{-\pi_2^{bias}}) \leq 0.1$.

3.2.4 Range Constraint. As we have only considered the real-valued ranges to compute the integer bits of the variables and constants in the first phase, we need to ensure that the finite ranges after each operation in each layer do not overflow. Our hypothesis is that the roundoff errors are small enough to keep the integer bits unaffected. Accordingly, in our range constraint C_2 in Figure 2, we want to ensure that the number of integer bits required to represent the finite-precision range after each operation I_i^{op} is enough to store the integer bits of the finite-precision result (real-valued ranges together with the roundoff errors).

However, directly implementing this constraint would result in applying the function `intBits` on error decision variables, which is non-linear. Hence, we reduce the problem of computing the integer bits to ensuring the ranges of the finite-precision result remains inside the maximum representable range (A_i^{op}) with the integer bits of the real-ranges after every operation at layer i.

We pre-compute the integer bits required for the real-valued result, and use them to generate the maximum representable range after each operation. For the dot operation, we have the propagation errors ϵ_i^{prop} from the previous layer and the operation itself introduces the new roundoff error ϵ_i^{dot}. Hence, after the dot operation, the finite range includes these two errors. For bias addition, however, we have the error from the dot operation as propagated error along with the new roundoff error ϵ_i^{bias} introduced by the addition. The total error after the bias addition is essentially the total error ϵ_i of the layer i as the activation error is zero for ReLU and Linear functions.

Let us consider our running example in Figure 1. As there is no initial error, for the first layer we simply add the roundoff error committed by the dot operation. For the bias, we add roundoff error for both the dot and the bias operation. The range constraints for the first layer is thus: $[-4.0, 4.0] \geq [-3.0, 3.0] + \epsilon_1^{dot}$ and $[-8.0, 8.0] \geq [-3.0, 6.0] + \epsilon_1^{dot} + \epsilon_1^{bias}$. Similarly, we generate range constraints for all the layers. Given the user inputs, our prototype tool Aster automatically encodes the objective cost function, the error and range constraints for the optimization problem.

3.2.5 Linearization of Constraints. However, we still have nonlinearity in the objective cost function which includes the nonlinear function `max`. The error constraints are also nonlinear in terms of the number of unknown fractional bits (π).

$$
\begin{aligned}
&C_{0(1)} : \gamma_i^{bias} \geq \pi_i^{dot} \\
&C_{0(2)} : \gamma_i^{bias} \geq \pi_i^{bias} \\
&C_{1(1)} : opt_i^{dot} = \sum_{j=1}^{m} (T_j \times b_j^{dot}) \\
&C_{1(2)} : opt_i^{bias} = \sum_{j=1}^{m} (T_j \times b_j^{bias}) \\
&C_{1(3)} : \sum_{j=1}^{m} b_j^{dot} = 1 \\
&C_{1(4)} : \sum_{j=1}^{m} b_j^{bias} = 1 \\
&C_{1(5)} : \epsilon_i^{new} \leq opt_i^{dot} + opt_i^{bias}
\end{aligned}
$$

Fig. 3. The linearization constraints

Hence, we linearize these constraints exactly without introducing any approximation. Figure 3 presents our linearization constraints for the optimization. Linearization of the cost objective is straight-forward. The `max` function of two variables π_i^{dot} and π_i^{bias} (as shown in Figure 2) in the form $\gamma_i^{bias} = \max(\pi_i^{dot}, \pi_i^{bias})$ can always be divided into a set of two linear constraints such that γ_i^{bias} is always bigger than or equal to all values of both π_i^{dot} and π_i^{bias}. We utilize this argument to linearize our `max` function as $C_{0(1-2)}$ in Figure 3.

Linearizing the error constraint is however more tricky. To linearize the error constraint, we use *indicator constraints* [7]. The nonlinear error constraint is defined in Figure 2 as: $\epsilon_i^{new} \leq 2^{-\pi_i^{dot}} + 2^{-\pi_i^{bias}}$ where π_i^{dot} and π_i^{bias} are unknown integers in an user-provided integer range $[lo, hi]$. Recall that, we want to compute the values of π_i^{dot} and π_i^{bias} that minimize our objective function.

As we already know the integer ranges of the variables, the possible values for π_i^{dot} and π_i^{bias} become finite (m) as they are exactly the same as the number of integers in the given range. With these m values, we define a set of m discrete reals that represent the set of possible values of $2^{-\pi_i^{dot}}$ and $2^{-\pi_i^{bias}}$: $T = [2^{-hi}, 2^{-(hi-1)}, \cdots, 2^{-lo}]$.

We introduce m binary indicator variables b^{dot} and b^{bias} for each valuation of $2^{-\pi_i^{dot}}$ and $2^{-\pi_i^{bias}}$ within the specified range. Intuitively, the indicator variables select only one specific value from the list T. We formulate two indicator constraints $C_{1(1-2)}$ for each layer i as presented in Figure 3.

The constraints $C_{1(1-2)}$ select the values of π_i^{dot} and π_i^{bias} that are optimal (opt_i^{dot} and opt_i^{bias}) respectively. They also state that if b_j^{dot} (or b_j^{bias}) is true, we select the value $2^{-(hi-j-1)}$ from the list T for opt_i^{dot} (or opt_i^{bias}). Obviously, we want only one of the b_j^{dot}(or b_j^{bias}) to be true. Hence, we add another two constraints $C_{1(3-4)}$ in Figure 3 to enforce that only one of these binary indicator variables is true. With these new indicator variables and constraints, finally, we linearize the original nonlinear error constraint ϵ_i^{new} in Figure 2 as $C_{1(5)}$ in Figure 3. Our tool Aster encodes these linearization constraints for all layers fully automatically.

Let us assume for our running example in Figure 1 that the range is provided as $[lo, hi] = [4, 8]$ which makes the possible values for π_i^{dot} and π_i^{bias}: $T = [2^{-8}, 2^{-7}, 2^{-6}, 2^{-5}, 2^{-4}]$. Next, we define 5 binary indicator variables for b^{dot} and b^{bias} each. The indicator constraints for dot product in the 1st layer are as follows: $opt_1^{dot} = \sum_{j=4}^{8} 2^{-j} \times b_j^{dot}$ and $\sum_{j=4}^{8} b_j^{dot} = 1$. Similarly we have indicator constraints for bias. If the solver picks $b_7^{dot} = 1$ and $b_8^{bias} = 1$, the corresponding new error is then bounded by $2^{-7} + 2^{-8}$, and the optimized fractional bit lengths of dot and bias are 7 and 8, respectively.

3.3 Step 3: Correctly Rounded Precision Assignment

After solving the MILP, we obtain the fractional bits required for the dot operation and the addition of bias, and we know that the integer bits from the first phase are enough to prevent overflow even in the presence of errors. However, the fractional bits computed for the dot product only apply to the *result* of the dot operation. To generate a complete executable fixed-point implementation, we must also compute the precision of the *intermediate* operations (sum of products of the dot) and the constants of weights. In particular, we need to determine their fractional bits such that the results are rounded correctly up to the precision determined by the MILP.

Our algorithm to compute the intermediate and constant word lengths is based on the fixed-point sum of products by constants (SOPC) algorithm [6, 13]. We first compute the fractional bits for the intermediate computation of dot. Assume x to be a vector of p fixed-point variables in formats (I_{x_i}, π_{x_i}) and c be a vector of p fixed-point constants in formats (I_{c_i}, π_{c_i}) where I denotes the integer bits. Our goal is to compute $y = \sum_{i=1}^{p} c_i \cdot x_i$ correctly.

The integer bits of the output I_y are already computed in the first step, such that no overflow occurs. The fractional bits of the output π_y are determined by MILP in the second step. These two combined represent the output precision and an accuracy requirement which ensures that the roundoff error is bounded by $2^{-\pi_y}$.

As it was shown in [13], if the integer bits of output I_y guarantee no overflow, and partial products $s_i = c_i \cdot x_i$ are performed exactly, then performing the summation in an extended format (I_y, π_{ext}) guarantees the output error bound. The number of extended fractional bits depends on the number of terms that need to be added: $\pi_{ext} = \pi_y + \lceil \log_2 p \rceil + 1$. Note that the integer bit positions for the intermediate results are not changed, which might lead to overflows in partial sums during the computation of the dot product. However, because of the properties of 2's complement, these

overflows do not influence the result accuracy as long as the output is representable with the output integer bit I_y [6].

Next, we must obtain the fractional bits of the weight constants ($w_i \in W$) such that the error bound holds. As we have mentioned before, the intermediate results of the partial sums needs to be done with π_{ext} fractional bits to ensure correct rounding. Now, in order to ensure that the accuracy of the product $w_i \times x_i$ is up to π_{ext}, the following property needs to hold: $\pi_{w_i} + I_{x_i} \leq \pi_{\text{ext}}$, which implies that the π_{w_i} needs to be at least $\pi_{\text{ext}} + I_{x_i}$.

Finally, the result of the dot product is added with the bias vector. From the MILP we obtained the fractional bit of the result of the addition π^{bias}. We need to compute the fractional bits of the bias constants ($b_i \in \overline{b}$). To ensure that the roundoff error at result is bounded by $2^{-\pi^{bias}}$, the fractional bits of b_i is set to be π^{bias}. As the formats of the operands might differ, we set the format of the bias upfront to ensure π^{bias} fractional bits in the result.

3.4 Soundness

THEOREM. Given a set of input ranges $\mathcal{R}$ and a specified error bound ϵ_{target} for a neural network, if the MILP-based mixed-precision optimization terminates successfully, it returns a fixed-point precision assignment for the network such that for all inputs $i \in \mathcal{R}$, the maximum roundoff error ϵ_n of the network (relative to a real-valued implementation) does not exceed ϵ_{target}, i.e., $\epsilon_n \leq \epsilon_{target}$.

Proof Sketch. Our MILP-based mixed-precision tuning procedure guarantees soundness by construction. We summarize the correctness argument for each of the three steps.

The first step employs sound interval arithmetic [42] to compute the integer bits of all program variables and constants, including intermediate ones. This computation proves the absence of overflow, ensuring that the resulting integer bits are valid and consistent.

The second step assigns the fractional bits of the dot and bias results as the solution of an MILP optimization problem, which bounds the overall roundoff error to be below the user-given error bound. The MILP error constraints soundly over-approximate the true roundoff errors (by assuming worst-case errors and error propagation at each step), and the range constraints ensure that no overflow is introduced due to roundoff errors. The linearization constraints are exact and maintain the soundness of the MILP encoding.

The third and final step assigns precisions to the intermediate variables that store the sum of products of the dot and the weight constants. This is achieved by utilizing the previously determined sound integer bits from the first step and following the fixed-point sum of products by constants (SOPC) algorithm. The correctness of the SOPC algorithm directly follows from [6].

Together, these steps assign both the integer and the fractional bits of all variables, constants and operations such that no overflow occurs and the overall error bound is satisfied. □

4 IMPLEMENTATION

Our tool Aster takes as input the network architecture and specification written in a small domain-specific language. The input corresponding to our running example from Figure 1 is shown in Figure 4. The '`require`' clause specifies the input ranges, and the '`ensuring`' clause specifies the overall error bound. In addition, Aster takes the maximum fractional bit length of the input vector, the word length of maximum weights, and the range of the fractional bits for the optimization as command-line user inputs. As explained in Section 3.2, we need these values to make the optimization problem linear.

Note that we have implemented Aster to handle feed-forward NNs with `relu` and `linear` activations fully automatically; however, Aster can straightforwardly be extended to handle neural

```
def runningExample(x:Vector): Vector = {
  require(lowerBounds(x, List(-10, -5)) && upperBounds(x, List(10, 5)))
  val weights1 = Matrix(List(List(0.1, 0.2), List(0.2, 0.15)))
  val weights2 = Matrix(List(List(0.1, 0.2)))
  val bias1 = Vector(List(1.0, 2.0))
  val bias2 = Vector(List(0.5))
  val layer1 = relu(weights1 * x + bias1)
  val layer2 = linear(weights2 * layer1 + bias2)
  layer2
} ensuring(res => res +/- 0.1)
```

Fig. 4. Network architecture in Aster's input format

networks with sparse matrix multiplication, and piece-wise linear activation functions with bounded domains.

Aster generates fully quantized code including (more) accurate precisions for the weights written in C using the `ap_fixed` library. This code can directly be compiled for an FPGA with the state-of-the-art HLS compiler by Xilinx [52]. Aster's generated code either expresses the matrix operations as for-loops, or alternatively it can fully unroll these loops.

Integer Bit Computation. We observed that a straight-forward interval analysis computation of the real-valued ranges quickly becomes expensive with increasing complexity of the network. We thus leverage the structure of the neural network for a more efficient, though over-approximate range analysis. Specifically, we abstract the input variables to each layer by a single range that soundly covers all individual variables, and do so similarly for the weights and biases at each layer. Doing so, we can compute the output range of a layer by computing a single dot product with one addition. This over-approximation makes the interval analysis scalable even for large networks with thousands of parameters, while we have not observed it to significantly effect the optimization results.

Choice of MILP Solver. For our fixed-point precision optimization, the values we encounter can be very small, e.g. for fixed 32 bit precision the roundoff errors are on the order of 1e-9. State-of-the-art MILP solvers use finite precision internally as well, and it is thus crucial to choose a solver that is precise enough to be able to distinguish its own internal roundoff errors from the values in our constraints. We integrate Aster with the SCIP optimization Suite [20] with the underlying SoPlex solver to solve our mixed-integer linear problem. SCIP internally uses extended-precision which goes beyond the limits of floating-point arithmetic, and thus allows us to deal with values as small as 1e-15. We found that other widely-used industrial solvers (such as CPLEX [29] and Gurobi [24]) have tolerances that are bounded by 1e-6. These are unfortunately too coarse for our fixed-point precision optimization. Note that SCIP's precision is not unlimited either. If the roundoff error goes beyond the tolerance limits (that is 1e-15), Aster cannot optimize and will report an error (i.e. it will *not* return an incorrect result).

5 EVALUATION

In this section we evaluate Aster based on the following research questions:
RQ1 How do different input parameters affect Aster's results and performance?
RQ2 How does Aster compare with the state-of-the-art in terms of implementation cost of NNs?
RQ3 How does Aster compare with the state-of-the-art in terms of optimization time?

Benchmarks. We have collected a set of 18 neural network controllers, consisting of 15 models from the competition at the Applied Verification for Continuous and Hybrid Systems (ARCH)

benchmarks	# in	# params	architecture
InvPendulum	4	60	$4 \times 10 \times 1$
MountainCar	2	336	$6 \times 15 \times 15 \times 2$
Acrobot	6	375	$6 \times 20 \times 20 \times 8$
MPC	6	720	$6 \times 20 \times 20 \times 8$
SinglePend	2	775	$2 \times 25 \times 25 \times 2$
DoublePendV1	4	825	$4 \times 25 \times 25 \times 2$
DoublePendV2	4	825	$4 \times 25 \times 25 \times 2$
ACC3	5	980	$5 \times 20 \times 20 \times 20 \times 1$
ACC5	5	1,820	$5 \times 20 \times 20 \times 20 \times 20 \times 20 \times 1$
VCAS	3	1,940	$3 \times 20 \times 20 \times 20 \times 20 \times 20 \times 9$
ACC7	5	2,660	$5 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 1$
AC6	12	3,457	$12 \times 64 \times 32 \times 16 \times 1$
Unicycle	4	3,500	$4\times 500 \times 2$
ACC10	5	3,920	$5 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 20 \times 1$
AC7	12	12,033	$12 \times 128 \times 64 \times 32 \times 1$
Airplane	12	13,540	$12 \times 100 \times 100 \times 20 \times 6$
ControllerTora	4	20,800	$4 \times 100 \times 100 \times 100 \times 1$
AC8	12	44,545	$12 \times 256 \times 128 \times 64 \times 1$

Table 2. Details of benchmark architectures, listing the number of inputs, and parameters (considering both weight and bias parameters) as well as the neurons in each layer

workshop from the years 2019 [40] and 2020 [34], where the networks were provided by academia as well as industry. Additionally, we included 3 controller models from the VNN-LIB standard benchmark set [22], which is widely used for the verification of neural networks. Verification of these controllers is becoming increasingly important due to their usage in safety and operational critical systems. We took all benchmarks for which we could extract all values of weights and biases from the repository. Table 2 provides details of the network architectures. We present here a brief description of the benchmarks; a more detailed discussion can be found in [14, 22, 34].

The InvPendulum, SinglePend, DoublePend, Acrobot (originally from [32]) are the neural network controllers for pendulums in different settings. The Unicycle and MountainCar, originally taken from [15] and [32] respectively, model cars to control two separate control objectives. We also have four different versions of ACC, an Adaptive Cruise Control system of a car, with 3, 5, 7, and 10 hidden layers. The MPC benchmark models a quadrotor (introduced in [32]) and the ControllerTora benchmark controls translational oscillations [16]. The Airplane and VCAS (originally from [35]) model a dynamical system and collision avoidance system ACAS X of an aircraft. The AC benchmarks are derived from a baseline case study involving a Drone controller [22] that regulates a single RPM value across all motors to facilitate takeoff and maintain a fixed hovering position.

We have extracted the safe input ranges, the weight matrices and bias vectors of these controllers from the competition's repository [1, 14], as well as the VNN-LIB repository [22]. The provided MATLAB files, hierarchical data format (HDF) files, and open format for ML models (ONNX) files were converted into the input format of Aster. The networks do not come with specified target error bounds, so we choose two target absolute error bounds uniformly for all networks, 1e-3 and 1e-5, that we believe to be in a reasonable range for embedded controllers, which are typically implemented in lower precision (16 bits or 32 bits).

Experimental Setup. All experiments are done on an Intel Core i5 Debian system with 3.3 GHz and 16 GB RAM. Aster uses SCIP Optimization Suite 7.0.3 as the external MILP solver. We set 1e-15 as the zero and feasibility tolerance limits for the SCIP as these tolerances are precise and efficient enough for our purpose. We further assume 32 as the initial guess for the word length of the maximum weights to start the optimization. We choose to compare Aster with the state-of-the-art precision tuner Daisy [11], as this is the only available *sound* mixed-precision tuner for fixed-point programs. For the comparison we use Daisy's version downloaded on 2^{nd} March, 2021 (there were no major commits since). We set a 5-hour time budget for each optimization run in our experiments. We believe that a total of 5-hour is a reasonable time for an analysis to generate a sound implementation once that can directly be synthesized on FPGAs. For synthesis of FPGA designs, we use Xilinx's Vivado HLS tool [52] (version v2020.1) downloaded on 27^{th} May, 2020, and set a timeout of 5 hours for our experiments.

5.1 RQ1: Parameterization

Aster has several input parameters that are needed to make the MILP optimization tractable, specifically the range of the fractional bits π and the input fractional bit lengths π_0, which determines the input error. These input parameters affect Aster's results and optimization time.

settings	π_0	initial error	π range
A	20	2^{-20}	[5, 32]
B	32	2^{-32}	[10, 32]
C	10	0	[5, 32]
D	17	0	[10, 32]

Table 3. Aster's settings

Choosing a wide range of π results in more variables and constraints, thus making it harder for the underlying SCIP optimizer to generate results. π_0 needs to be large enough to admit a valid solution to the optimization problem, but too large π_0 will, in general, result in a—potentially unnecessarily—higher overall cost. Note that since Aster computes an over-approximation of the error, the solver may report infeasibility even though a solution to the not-approximated problem may exist.

We determine suitable parameters for our benchmarks with a systematic empirical exploration using four different settings denoted by the letters 'A-D' shown on the left in Table 4, considering the two error bounds 1e-3 and 1e-5. Settings A and B both consider input errors; setting B is expected to be make more benchmarks feasible at the expense of potentially higher cost due to smaller initial roundoff error. Settings C and D set the input error to zero. These settings are useful when the user knows that the inputs are represented exactly and is only interested in considering the roundoff error during the internal computations.

Table 4 shows the overall cost of the precision assignment determined by Aster for these 4 different settings. Setting A is, in general, cost-effective with a larger error bound (1e-3), but results in infeasible error bounds for the larger networks. The reason for this is that the bigger initial error gets magnified along with computing new errors at each layer, thus making it impossible to achieve the target error bound. As expected, more benchmarks become feasible with setting B. However, the tradeoff is that setting B mostly computes a larger cost as it considers the largest initial fractional bit length.

Settings C and D are expected to compute the lowest cost, as the input variables in these settings incur no error, along with smaller initial fractional bit lengths. This is indeed the case for 9 and 13 of the benchmarks out of 18, with error bounds of 1e-3 and 1e-5, respectively. For the rest of the benchmarks settings C and D compute higher costs than settings A and B, though the costs are mostly close to those of setting A or B. We have observed that for these benchmarks, the optimizer finds some specific assignment configurations that work optimally given the optimization problem. These assignments cost more in the end when we compute the final costs of the whole program

benchmarks	target error = 1e-3			target error = 1e-5		
	A	B	C	A	B	D
InvPendulum	**20404**	**20404**	**20404**	36886	34966	**34006**
MountainCar	**151160**	154934	163486	inf	**207970**	**207970**
Acrobot	175601	**151103**	157546	274887	**248595**	**248595**
MPC	**250952**	**250952**	**250952**	inf	**250952**	**250952**
SinglePend	271918	362100	**263547**	**436565**	438992	438992
DoublePendV1	**352720**	420640	408280	581144	**540042**	**540042**
DoublePendV2	442416	**436810**	453224	inf	508924	**505722**
ACC3	444638	**442438**	**442438**	inf	**446838**	**446838**
ACC5	inf	**702228**	**702228**	inf	**702228**	**702228**
VCAS	inf	inf	inf	inf	inf	inf
ACC7	inf	inf	inf	inf	inf	inf
AC6	2069122	**1986944**	2009218	inf	**2069122**	**2069122**
Unicycle	**1134062**	**1134062**	**1134062**	1736074	**1636072**	**1636072**
ACC10	inf	inf	inf	inf	inf	inf
AC7	inf	**6660220**	**6660220**	inf	**6660220**	**6660220**
Airplane	inf	**5967462**	**5967462**	inf	inf	inf
ControllerTora	**10562548**	11605748	11632956	**13532966**	**13532966**	**13532966**
AC8	inf	21338216	**21261416**	inf	**21338216**	**21338216**

Table 4. Parameter evalutation of Aster's with settings 'A-D', inf denotes infeasibility

after assigning precision to all intermediate program variables and constants. If the user already knows how many bits are required to represent the inputs exactly, setting D is better, as Aster can take that into account.

The benchmarks VCAS, ACC7, ACC10 are infeasible with all the settings. Upon closer inspection, we found that this is due to intermediate ranges becoming very large (on the order of 1e+9). For example, if we reduce the range of one of the input variables of the VCAS benchmark slightly (from [-133, -129] to [-130, -129]), Aster's setting B *is* able to generate a precision assignment in 16.45 seconds for the target error bound of 1e-3. That said, since Aster cannot generate a precision assignment for the original input ranges, we remove these benchmarks from the later experiments.

We explore the generic settings A-D for our evaluation, but we expect that when using Aster on a specific application, the user may either know suitable parameter values up front, or may run Aster several times to explore different options. The latter is feasible due to Aster's small optimization times (see Section 5.3).

We use settings A and B with error bounds 1e-3 and 1e-5, respectively, for our comparison with the state-of-the-art in Section 5.2 except for ACC5, AC7, Airplane, and AC8. For these four benchmarks we will use setting B for *both* error bounds, as only with setting B, we could generate implementations. We do not include settings C and D as the state-of-the-art tuner does not support analysis without initial errors.

5.2 RQ2: Implementation Cost

We compare Aster with the state-of-the-art precision tuner Daisy [11] that focuses explicitly on optimizing the precision to satisfy given roundoff error bounds and generates mixed fixed-precision implementations of arithmetic programs that are *sound* considering all possible fixed-point precisions (not only, say, 4 or 8 bits). Note that dynamic quantization tools like SeeDot [21] and Shiftry [37] are not sound and are designed specifically for neural network classifiers, and do

not handle feed-forward neural network controllers that solve regression problems that we focus on in this work.

Daisy uses a forward data-flow analysis to soundly compute intermediate ranges, avoid overflow, and compute roundoff errors. It works on generic straight-line code without loops, and in particular, does not handle programs with data structures like matrices and vectors that are standard in neural networks. To use Daisy on these programs, we completely unroll the loops and data structures, i.e., manually assign individual matrix and vector elements to individual scalar variables.

Daisy supports the following two modes:

- **Uniform:** In this mode, Daisy computes the total error bound for a given fixed precision. To determine the lowest uniform precision that satisfies the error bound, we manually employ Daisy repeatedly in this mode and check if the computed error satisfies the given bound.
- **Mixed-precision:** This mode generates a mixed fixed-point precision assignment using a heuristic search based on delta-debugging [11] that repeatedly evaluates the roundoff errors for different precision assignments, starting from the lowest uniform precision that satisfies the given error bound. However, the computational cost of this heuristic search increases rapidly as the size of the neural network grows.

Daisy generates the tuned fixed-point code as a C program in the same format as Aster that can be directly compiled to an FPGA by the Xilinx compiler [52]. Just like the input, the output code of Daisy is fully unrolled straight-line code and potentially very large.

We compare Aster's setting A (fractional bits in the range $[5, 32]$ and the input fractional bit length = 20), setting B (fractional bits in the range $[10, 32]$ and the input fractional bit length = 32), and Daisy with uniform and mixed-precision tuning with a maximum bit length of 32 bits on all our benchmarks considering the 1e-3 and 1e-5 error bounds. For uniform precision, we use the lowest *uniform* precision that satisfies the error bounds. Note that we do not consider settings C and D, as in those settings we assume no initial error, which Daisy does not support.

Both Aster and Daisy share the common objective of minimizing the total number of bits utilized by the neural network (NN). However, they employ entirely different techniques to achieve this goal. Daisy employs a *heuristic* search method that involves multiple iterations of error analysis—one for each candidate precision assignment. In contrast, Aster adopts a global optimization-based approach: it creates a single optimization constraint and solves it for the precision assignment. Although both tools employ cost functions for mixed precision tuning, these functions are similar but not identical.

We cannot use Daisy's cost function directly in Aster as it would lead to non-linear constraints. Specifically, Aster uses assumptions (see Section 3.2) regarding the bit lengths of the NN structures (matrices, vectors) and the inputs to maintain linearity. These assumptions make sense when optimizing neural networks, for example, to keep the data structures intact in the final implementation. It is also not immediately possible to adapt Daisy to use Aster's cost function. Daisy considers the unrolled code as-is, optimizes each variable individually, and assigns precisions to all variables in a program, including the inputs. Hence, Daisy only has the unrolled code available while optimizing, but would need the higher-level data structure information for Aster's cost function. Hence, an entirely fair comparison of Daisy and Aster is impossible as their targeted and possible optimizations differ.

Latency of FPGA Implementations. Ultimately, what matters is the actual performance of the generated mixed-precision code. Since Daisy has been used to optimize the latency of fixed-point implementations on FPGAs [33], we compare the tools on that measure. We compile the code generated by Aster and Daisy for a (standard) FPGA architecture using the Xilinx' Vivado HLS tool and compare the running time in terms of machine cycles, i.e. latency, that the compiler reports for the final hardware implementation. (We cannot customize our cost function further for the HLS

benchmarks	target error: 1e-3			target error: 1e-5		
	Daisy		Aster (setting A)	Daisy		Aster (setting B)
	uniform	mixed		uniform	mixed	
InvPendulum	**12**	**12**	**12**	14	14	**13**
MountainCar	27	27	**25**	28	29	**25**
Acrobot	**24**	**24**	25	**24**	25	26
MPC	inf	inf	**35**	inf	inf	**37**
SinglePend	**22**	23	25	**24**	**24**	27
DoublePendV1	29	**28**	**28**	31	31	**28**
DoublePendV2	36	36	**27**	36	36	**30**
ACC3	49	49	**44**	inf	inf	**46**
ACC5	inf	inf	**72**$^{\#}$	inf	inf	**74**
AC6	62	62	**48**	inf	inf	**49**
Unicycle	178	178	**27**	inf	inf	**28**
AC7	TO	TO	**8310**$^{*\#}$	inf	inf	**8310**$^{*\#}$
Airplane	TO	TO	**9001**$^{*\#}$	TO	TO	inf
ControllerTora	TO	TO	**13158***	TO	TO	**13558***
AC8	TO	TO	×$^{*\#}$	TO	TO	×$^{*\#}$

Table 5. Latencies of implementations generated by Daisy and Aster considering errors 1e-3 and 1e-5 respectively (TO: timed out after 5 hours, inf: tools returned infeasible, #: used Aster's setting B, *: compiled with explicit loops (i.e. not unrolled code)), ×: Xilinx failed to compile the implementation

compiler, as it is commercial and its internal implementation choices are unknown.) Note that this reported latency is exact, and we thus do not measure the noisy runtime on actual hardware.

Table 5 shows the latencies of the generated code for our benchmarks considering the target errors 1e-3 and 1e-5. It shows that Aster generates feasible implementations for significantly more benchmarks, and for benchmarks where both tools successfully generate compilable code, Aster mostly produces code with lower latency than Daisy. Considering the latter benchmarks, we see that Aster matches the performance of Daisy's generated code for two benchmarks and improves on it for 5 benchmarks. Only for two benchmarks, Aster's code's latencies are (slightly) larger. Our results thus show that Aster's optimization is often able to produce faster implementations for neural network controllers than the state-of-the-art *heuristic* mixed-precision tuner of Daisy.

Comparing with Daisy's uniform precision assignments, we furthermore confirm that mixed precision is indeed overall beneficial for platforms such as FPGAs. This is particularly striking for the Unicycle benchmark, where Aster's reduction in latency is almost 84% considering the 1e-3 error as Aster is able to optimize some variables much more significantly than Daisy. The latency considering the smaller error bound is only slightly larger, whereas Daisy reports infeasibility.

Our results also show that Aster is more scalable than Daisy; generating a precision assignment for all benchmarks for the larger error bound (Daisy failing on 6), and reporting an infeasible result for one benchmark with the smaller error bound (Daisy failing on 9). Owing to our optimization problem formulation, the number of variables of the MILP grows linearly with the number of layers: one decision variable for the dot product computation and the other for adding bias. This formulation makes it possible for Aster to find a solution even for large networks. In contrast, Daisy's heuristic search becomes intractable with the increasing complexity of the network, leading to many variables and constants. In addition, Daisy performs the error analysis multiple times to ensure that the precision assignment meets the error requirement, thus timing out often.

benchmarks	target error: 1e-3, setting: A				target error: 1e-5, setting: B			
	Daisy's cost		Aster's cost		Daisy's cost		Aster's cost	
	u (%)	m (%)	u (%)	m (%)	u (%)	m (%)	u (%)	m (%)
InvPendulum	-6.44	-13.97	**15.99**	**17.80**	-23.52	-37.31	**17.08**	**13.45**
MountainCar	-43.14	-52.47	-2.64	-0.02	-19.68	-31.47	**14.18**	**17.84**
Acrobot	-25.70	-37.30	-21.84	-34.12	-6.10	-19.93	-8.21	-2.12
MPC	*	*	*	*	*	*	*	*
SinglePend	**1.89**	-11.12	**11.95**	**2.58**	**0.96**	-2.57	**10.81**	**10.69**
DoublePendV1	-16.36	-24.33	**6.00**	**3.61**	-14.12	-16.11	**5.01**	**10.01**
DoublePendV2	-32.61	-40.95	**0.92**	**6.22**	-40.71	-47.05	**24.56**	**24.38**
ACC3	-62.01	-65.50	**38.37**	**19.35**	*	*	*	*
ACC5	inf	inf	inf	inf	*	*	*	*
AC6	-66.95	-64.96	**18.13**	**17.45**	*	*	*	*
Unicycle	**24.10**	**9.50**	**48.15**	**39.20**	*	*	*	*
AC7	×	×	×	×	*	*	*	*
Airplane	inf	inf	inf	inf	inf	inf	inf	inf
ControllerTora	×	×	×	×	×	×	×	×
AC8	×	×	×	×	×	×	×	×

Table 6. Reduction (%) in cost using Aster's settings A and B w.r.t. Daisy's uniform (u) and mixed (m) analyses considering 1e-3 and 1e-5 error bounds using Daisy's and Aster's cost functions (×: Daisy times out after 5 hours, *: Daisy returns infeasible but Aster generates an implementation, inf: Aster returns infeasible)

We run the Xilinx Vivado compiler on the fully unrolled code as we have observed that it leads to smaller latencies (and Daisy generates fully unrolled code). The exceptions are the AC7, Airplane, ControllerTora, and AC8 benchmarks (marked with a star in Table 5), where the generated C code is too large to be compiled within the 5h timeout (~ 36K, 40K, 62K, and 134K lines of code). Aster can generate smaller programs (54, 58, 54, 54 lines of code) that preserve the original NN structures and loops, and we use these implementations for compilation with Xilinx. The benchmark AC8, however, is still too large in terms of the number of loop iterations to be compiled by Xilinx within 5 hours without further customization. Nonetheless, in general our results show that considering the high-level structure of NNs is beneficial especially for larger networks.

Comparison of Cost Functions. We also compare Daisy's and Aster's results using both cost functions: we generate unrolled code by Aster to use Daisy's cost function, and we use Aster's cost function to evaluate code generated by Daisy. Table 6 shows the cost reduction in % achieved by Aster with respect to Daisy's analyses. A negative result signifies that the cost of Aster's implementation is higher than Daisy's.

Not surprisingly, our results show that Aster outperforms Daisy's analyses considering Aster's cost function excluding both MountainCar and Acrobot with error bound of 1e-3 and only Acrobot with error bound of 1e-5. However, with Daisy's cost function, Daisy's analyses, both uniform and mixed outperform Aster, where the cost is computed on the unrolled program, with the exception of the Unicycle benchmark with 1e-3 error bound and the SinglePend with uniform precision analysis.

Aster's cost function on Daisy's implementation considers the largest bit lengths, one for weights and the other for bias in a layer, to compute a sound cost. This defeats Daisy's advantage of optimizing all variables, thus resulting in higher costs than Aster. Likewise, using Daisy's cost function for Aster's implementation is also suboptimal for Aster as it considers the same bit length

benchmarks	error: e-3			error: e-5		
	Daisy (unif)	Daisy (mix)	Aster (A)	Daisy (unif)	Daisy (mix)	Aster (B)
InvPendulum	1.72	4.19	**1.66**	1.70	5.42	**1.62**
MountainCar	5.16	43.68	**2.22**	5.11	38.79	**2.15**
Acrobot	6.25	196.93	**2.31**	6.04	95.41	**2.24**
MPC	-	-	**3.50**	-	-	**3.44**
SinglePend	17.63	237.83	**3.52**	17.23	115.25	**3.53**
DoublePendV1	19.42	127.96	**3.69**	19.69	201.13	**3.84**
DoublePendV2	20.55	246.64	**3.80**	20.66	334.03	**3.72**
ACC3	27.13	292.05	**7.28**	-	-	**4.84**
ACC5	-	-	**12.97**$^{\#}$	-	-	**12.85**
AC6	311.37	2033.23	**51.01**	-	-	**51.04**
Unicycle	354.71	9980.65	**49.92**	-	-	**45.78**
AC7	3999.54*	TO	**727.40**$^{\#}$	-	-	**729.05**
Airplane	**906.93***	TO	1060.92$^{\#}$	TO	TO	-
ControllerTora	12128.83*	TO	**2875.95**	TO	TO	**2521.21**
AC8	TO	TO	**13771.43**$^{\#}$	TO	TO	**13794.66**

Table 7. Optimization time (in seconds) averaged over 3 runs; TO: timed out after 5 hours, *: used 32 bit uniform precision analysis, #: used Aster's setting B instead of A, -: returns infeasible)

in a layer for linearity and to keep NN structures intact, thus improving scalability. We thus conclude that Daisy and Aster are each better for their intended use cases. Upon closer inspection of Aster's and Daisy's generated precisions, we further observed that Aster's assignment of precisions in the dot product is sometimes overly and unnecessarily conservative and can be improved in the future. That said, as we have seen in Table 5, unrolled code can become impractical for hardware implementations of large neural networks. In these cases, Aster's optimization and cost function are better suited, with which Aster performs significantly better than Daisy.

5.3 RQ3: Running Time

We compare the optimization time of Daisy's uniform and mixed-precision tuning with Aster's setting A for the target error of 1e-3 and Aster's setting B for the target error of 1e-5. For Daisy's uniform precision, we only show the time to run the roundoff error analysis once *after* we find the lowest uniform precision satisfying the error bound using Daisy's mixed precision analysis (we could not find a better way to generate this information). Also, Aster generates infeasibility with setting A for the ACC5, AC7, Airplane, and AC8 benchmarks with target error 1e-3, we show the running time with setting B. Table 7 shows the end-to-end optimization time measured by the bash time command of Daisy and Aster in seconds averaged over 3 runs.

In general, running uniform precision roundoff analysis once is quick for smaller benchmarks, but with the increased size of the network, it also becomes slower and even times out after 5 hours for the largest benchmark AC8 with both target error bounds. The running times are substantial even though Daisy assigns a uniform given bit length because it still needs to run the error analysis to ensure overflow freedom and appropriately assign integer bits. Daisy's mixed-precision tuning usually is a magnitude slower than the uniform precision analysis.

Aster outperforms Daisy's both analyses substantially in terms of running time for both error bounds for almost *all* benchmarks with the exception of Airplane where running Daisy's uniform analysis once is the fastest. Due to our formalization of the optimization problem, the number of constraints and variables only depends linearly on the number of layers and the range of fractional

bits (π) for Aster's settings. Our experiments show that solving an MILP problem with a limited number of variables is very efficient; when a solution was feasible the solver returned in at most 0.15 seconds. Our experiments thus confirm that constraint optimization is not a bottleneck of our mixed precision tuning and our approach is indeed a viable solution for fixed point computation, even for large networks with many parameters.

As setting B uses a smaller range of fractional bits ([10, 32]), the number of constraints in this setting is less than in setting A, thus, in general, resulting in smaller running time. Also, the number of variables and constraints increases only linearly with an increased number of layers. Solving an MILP problem with a limited number of variables is very efficient. The `Airplane`, `ControllerTora`, and `AC8` are the largest benchmarks with the most number of variables and constraints. For these MILP problems, where a solution is feasible (except `AC8` with setting A and `Airplane` in both settings), it was found in a maximum of 0.15 seconds. Our experiments thus confirm that constraint optimization is not a bottleneck of our mixed precision tuning. Our approach is indeed a viable solution for fixed point computation, even for large networks with many parameters.

6 RELATED WORK

Since there is a substantial body of work on efficient neural network inference, we focus here on a representative selection of the most immediately relevant techniques, and point to existing surveys [43, 53] for an overview of other complementary approaches (e.g., model and weight pruning). Verification techniques of other aspects of neural networks (e.g., adversarial examples) are, for instance, covered in the survey by Huang et.al. [28]. A more recent work [41] introduces a MILP formulation for verifying robustness of quantized neural networks. However, this work is orthogonal to ours as it focuses on post-quantization verification and does not consider the impact of roundoff errors. Another work [54] focuses on computing output ranges of feed-forward neural networks with non-linear activations (e.g., sigmoid and exponential linear units) in safety-critical systems, where it encodes the non-linear activations with linear constraints. It could be an interesting future direction for Aster to extend it for non-linear activations by combining their technique.

Quantization is a frequently applied technique to reduce the memory footprint of neural networks, but most techniques have been applied on neural network classifiers outside of safety-critical applications and unlike Aster do *not provide any accuracy guarantees*. We present a brief overview of these approaches in this paper, but note that they are fundamentally not comparable with our *sound* mixed-precision tuning for regression problems. These methods typically focus on selecting a particular uniform (custom) floating-point [36, 45] or fixed-point precision [21, 23, 39] and showing empirically that it performs well on a particular data/benchmark set. Shiftry [37] automatically chooses a mixed fixed-point precision by iteratively reducing the precision of variables in recurrent neural networks from 16 to 8 bits to run on memory restricted hardware. Another approach to mixed-precision is to dynamically adapt to outliers, i.e. particularly large values, in inputs or weights by providing specialized hardware architectures [44, 47, 48]. Alternative number representations have also been considered, such as stochastic computing [17], posits [9] and floating-points with different, tunable exponent and mantissa widths [19, 50].

Recent work [26] on quantization-aware training for embedded controllers offers new tools for automatic parameter compression or quantization by retraining the network and using a multiplier-less approach for dense layers. This work does not analyze the rounding error propagation, but it is complementary to our approach, and combining these two is a potential future work. There exist different techniques to build custom operators for sums of products by constants with an accuracy requirement. The first work of Dinechin et al. [13] presents an approach that permits to use only a necessary amount of bits for intermediate products, based on a worst-case error-analysis, and we use this approach in this paper. Another technique is a multiplierless approach: multiplications

by constants are replaced by bit-shifts and additions, and intermediate terms are shared when several constants are multiplied [38]. A new ILP-based solution for sound truncated multipliers was recently proposed [46], and it takes the result's format as its input. Aster could be extended to use this approach, instead of [13] for the Step 3, but the scalability of [46] is a major concern for word-lengths beyond 16 bits. For the best fine-tuned FPGA-based dot products the standard HLS tools have to be replaced by custom hardware code generators, such as FloPoCo tool [12]. But then, a whole new problem of a custom NN accelerator arises.

The tool POP finds mixed-precision assignments for floating-point arithmetic code by phrasing the problem as an ILP problem [2]. POP uses a dynamic analysis to infer variable ranges and thus does not guarantee complete soundness. A similar technique has been applied to tuning the floating-point precision of neural networks [30]. FPTuner [10] provides sound mixed-precision tuning for floating-point arithmetic by solving a nonlinear interval-valued optimization problem. Unfortunately, approaches for floating-point arithmetic are generally not applicable to fixed-point arithmetic due to its non-dynamic range.

The tool Daisy [11, 33] includes a sound mixed-precision tuning procedure for both floating-point and fixed-point arithmetic, based on an iterative search using delta-debugging. Other iterative approaches have been explored for mixed-precision tuning for fixed-point arithmetic, for example Min+1 or Max-1 [8], and a combination of Bayesian optimization and Min+1 [25]. Iterative approaches have been proposed to overcome difficulties in phrasing and more importantly solving sound global optimization problems. Alternatives are relaxing the integer to real-valued optimization [18], but work for Digital signal processors (DSPs) only. Generally, these techniques (with the exception of Daisy) treat errors as uncorrelated and additive or evaluate them dynamically with simulation, and thus do not guarantee soundness. We have shown that for the restricted class of neural networks of limited size and with appropriate approximations, a sound optimization is indeed feasible.

Fixed-point arithmetic was also recently formalized as a theory in the context of Satisfiability Modulo Theory (SMT) solvers [5]. However, to compute or bound roundoff errors, such a theory must be combined with the real-valued theory (representing the ideal program), which is in general inefficient.

7 CONCLUSION

This paper presents a novel sound quantization approach that assigns fixed-point mixed precision to neural networks that solve regression tasks, guaranteeing a user-provided roundoff error bound. We reduce the problem of precision optimization to an MILP problem and show how to effectively introduce sound over-approximations to be able to solve it efficiently using a state-of-the-art solver. Our experiments show that our method is fast, even for large networks with thousands of parameters. This efficiency indicates that constraint optimization is not a bottleneck here. Our proposed technique is viable for soundly quantizing neural networks that appear as verified embedded controllers. It is able to handle more and larger benchmarks than existing fixed-point tuners, and mostly generates more efficient code for custom hardware such as FPGAs. While this paper primarily focuses on neural network controllers which are typically regression models, there is potential for extending Aster's applicability to quantizing classifiers with more work on bridging the gap between numerical and classification accuracy.

ACKNOWLEDGMENTS

This work has received French government support granted to the CominLabs Excellence Laboratory and managed by the National Research Agency in the "Investing for the Future" program under reference ANR-10-LABX-07-01.

REFERENCES

[1] 2020. ARCH-COMP2020 Github Repository. https://github.com/verivital/ARCH-COMP2020
[2] Assalé Adjé, Dorra Ben Khalifa, and Matthieu Martel. 2021. Fast and Efficient Bit-Level Precision Tuning. In *SAS*.
[3] Adolfo Anta, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. 2010. Automatic verification of control system implementations. In *EMSOFT*.
[4] Christian Artigues, Oumar Koné, Pierre Lopez, and Marcel Mongeau. 2015. Mixed-integer linear programming formulations. In *Handbook on Project Management and Scheduling Vol. 1*. Springer, 17–41.
[5] Marek Baranowski, Shaobo He, Mathias Lechner, Thanh Son Nguyen, and Zvonimir Rakamarić. 2020. An SMT theory of fixed-point arithmetic. In *IJCAR*. Springer.
[6] Sylvie Boldo, Diane Gallois-Wong, and Thibault Hilaire. 2020. A Correctly-Rounded Fixed-Point-Arithmetic Dot-Product Algorithm. In *ARITH*.
[7] Pierre Bonami, Andrea Lodi, Andrea Tramontani, and Sven Wiese. 2015. On mathematical programming with indicator constraints. *Mathematical programming* 151, 1 (2015).
[8] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie. 2001. An Automatic Word Length Determination Method. In *ISCAS*.
[9] Zachariah Carmichael, Hamed Fatemi Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. 2019. Deep Positron: A Deep Neural Network Using the Posit Number System. In *DATE*.
[10] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2017. Rigorous floating-point mixed-precision tuning. In *POPL*.
[11] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound Mixed-Precision Optimization with Rewriting. In *ICCPS*.
[12] Florent de Dinechin. 2019. Reflections on 10 Years of FloPoCo. In *ARITH*.
[13] Florent de Dinechin, Matei Istoan, and Abdelbassat Massouri. 2014. Sum-of-product architectures computing just right. In *ASAP*.
[14] Diego Manzanas Lopez and Patrick Musau. 2019. ARCH-2019 Github Repository. https://github.com/verivital/ARCH-2019
[15] Souradeep Dutta, Xin Chen, and Sriram Sankaranarayanan. 2019. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *HSCC*.
[16] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Learning and verification of feedback control systems using feedforward neural networks. *IFAC-PapersOnLine* 51, 16 (2018).
[17] S. Rasoul Faraji, M. Hassan Najafi, Bingzhe Li, David J. Lilja, and Kia Bazargan. 2019. Energy-Efficient Convolutional Neural Networks with Deterministic Bit-Stream Processing. In *DATE*.
[18] Paul D. Fiore. 2008. Efficient Approximate Wordlength Optimization. *IEEE Trans. Computers* 57, 11 (2008).
[19] Marta Franceschi, Alberto Nannarelli, and Maurizio Valle. 2018. Tunable Floating-Point for Artificial Neural Networks. In *ICECS*.
[20] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, and Wei-Kun Chen et al. 2020. *The SCIP Optimization Suite 7.0*. Technical Report. http://www.optimization-online.org/DB_HTML/2020/03/7705.html
[21] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019. Compiling KB-Sized Machine Learning Models to Tiny IoT Devices. In *PLDI*.
[22] Dario Guidotti, Stefano Demarchi, Armando Tacchella, and Luca Pulina. 2023. The Verification of Neural Networks Library (VNN-LIB). https://www.vnnlib.org
[23] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *ICML*.
[24] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com
[25] Van-Phu Ha and Olivier Sentieys. 2021. Leveraging Bayesian Optimization to Speed Up Automatic Precision Tuning. In *DATE*.
[26] Tobias Habermann, Jonas Kühle, Martin Kumm, and Anastasia Volkova. 2022. Hardware-Aware Quantization for Multiplierless Neural Network Controllers. In *APCCAS*.
[27] Chao Huang, Jiameng Fan, Wenchao Li, Xin Chen, and Qi Zhu. 2019. ReachNN: Reachability Analysis of Neural-Network Controlled Systems. *ACM Trans. Embed. Comput. Syst.* 18, 5s (2019).
[28] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. 2020. A Survey of Safety and Trustworthiness of Deep Neural Networks: Verification, Testing, Adversarial Attack and Defence, and Interpretability. *Comput. Sci. Rev.* 37 (2020).
[29] IBM ILOG. 2009. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation* 46, 53 (2009).
[30] Arnault Ioualalen and Matthieu Martel. 2019. Neural Network Precision Tuning. In *QEST*.
[31] Radoslav Ivanov, Kishor Jothimurugan, Steve Hsu, Shaan Vaidya, Rajeev Alur, and Osbert Bastani. 2021. Compositional Learning and Verification of Neural Network Controllers. *ACM Trans. Embed. Comput. Syst.* 20, 5s (2021).
[32] Radoslav Ivanov, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. 2019. Verisig: verifying safety properties of hybrid systems with neural network controllers. In *HSCC*.

[33] Anastasiia Izycheva, Eva Darulova, and Helmut Seidl. 2019. Synthesizing Efficient Low-Precision Kernels. In *ATVA*.
[34] Taylor T Johnson, Diego Manzanas Lopez, Patrick Musau, Hoang-Dung Tran, Elena Botoeva, Francesco Leofante, Amir Maleki, Chelsea Sidrane, Jiameng Fan, and Chao Huang. 2020. ARCH-COMP20 Category Report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for Continuous and Hybrid Systems Plants. In *ARCH (EPiC Series in Computing)*.
[35] Kyle D Julian and Mykel J Kochenderfer. 2019. A reachability method for verifying dynamical systems with deep neural network controllers. *arXiv preprint arXiv:1903.00520* (2019).
[36] Urs Köster, Tristan J. Webb, Xin Wang, and Nassar et al. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *NIPS*.
[37] Aayan Kumar, Vivek Seshadri, and Rahul Sharma. 2020. Shiftry: RNN inference in 2KB of RAM. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).
[38] Martin Kumm. 2018. Optimal Constant Multiplication Using Integer Linear Programming. *IEEE Trans. Circuits Syst. II Express Briefs* 65-II, 5 (2018).
[39] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *PMLR*.
[40] Diego Manzanas Lopez, Patrick Musau, Hoang-Dung Tran, and Taylor T Johnson. 2019. Verification of closed-loop systems with neural network controllers. *EPiC Series in Computing* 61 (2019).
[41] Samvid Mistry, Indranil Saha, and Swarnendu Biswas. 2022. An MILP Encoding for Efficient Verification of Quantized Deep Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).
[42] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. 2009. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics.
[43] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. 2022. Machine Learning at the Network Edge: A Survey. *ACM Comput. Surv.* 54, 8 (2022).
[44] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. In *ISCA*.
[45] Kathirgamaraja Pradeep, Kamalakkannan Kamalavasan, Ratnasegar Natheesan, and Ajith Pasqual. 2018. EdgeNet: SqueezeNet like Convolution Neural Network on Embedded FPGA. In *ICECS*.
[46] Anastasia Volkova Remi Garcia and Martin Kumm. 2022. Truncated Multiple Constant Multiplication with Minimal Number of Full Adders. In *ISCAS*.
[47] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *ISCA*.
[48] Zhuoran Song, Bangqi Fu, Feiyang Wu, Zhaoming Jiang, Li Jiang, Naifeng Jing, and Xiaoyao Liang. 2020. DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration. In *ISCA*.
[49] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. 2019. Formal Verification of Neural Network Controlled Autonomous Systems. In *HSCC*.
[50] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. 2020. Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference. In *DAC*.
[51] Hoang-Dung Tran, Feiyang Cai, Diego Manzanas Lopez, Patrick Musau, Taylor T. Johnson, and Xenofon D. Koutsoukos. 2019. Safety Verification of Cyber-Physical Systems with Reinforcement Learning Control. *ACM Trans. Embed. Comput. Syst.* 18, 5s (2019).
[52] Vivado Lab Solutions. 2021. Vivado Design Suite. https://www.xilinx.com
[53] Erwei Wang, James J. Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter Y. K. Cheung, and George A. Constantinides. 2019. Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going. *Comput. Surveys* 52, 2 (2019).
[54] Zhiwu Xu, Yazheng Liu, Shengchao Qin, and Zhong Ming. 2022. Output Range Analysis for Feed-Forward Deep Neural Networks via Linear Programming. *IEEE Transactions on Reliability* (2022).
[55] Randy Yates. 2009. Fixed-point arithmetic: An introduction. *Digital Signal Labs* 81, 83 (2009).