electronics

MDPI

Article

CBin-NN: An Inference Engine for Binarized Neural Networks

Fouad Sakr [1,2], Riccardo Berta [1], Joseph Doyle [2], Alessio Capello [1], Ali Dabbous [1], Luca Lazzaroni [1,*] and Francesco Bellotti [1]

[1] Department of Naval, Electrical, Electronics and Telecommunications Engineering, University of Genoa, 16145 Genoa, Italy; fouad.sakr@edu.unige.it (F.S.); riccardo.berta@unige.it (R.B.); alessio.capello@edu.unige.it (A.C.); ali.dabbous@unige.it (A.D.); francesco.bellotti@unige.it (F.B.)

[2] School of Electronic Engineering and Computer Science, Queen Mary University of London, London E1 4NS, UK; j.doyle@qmul.ac.uk

* Correspondence: luca.lazzaroni@edu.unige.it

Abstract: Binarization is an extreme quantization technique that is attracting research in the Internet of Things (IoT) field, as it radically reduces the memory footprint of deep neural networks without a correspondingly significant accuracy drop. To support the effective deployment of Binarized Neural Networks (BNNs), we propose CBin-NN, a library of layer operators that allows the building of simple yet flexible convolutional neural networks (CNNs) with binary weights and activations. CBin-NN is platform-independent and is thus portable to virtually any software-programmable device. Experimental analysis on the CIFAR-10 dataset shows that our library, compared to a set of state-of-the-art inference engines, speeds up inference by 3.6 times and reduces the memory required to store model weights and activations by 7.5 times and 28 times, respectively, at the cost of slightly lower accuracy (2.5%). An ablation study stresses the importance of a Quantized Input Quantized Kernel Convolution layer to improve accuracy and reduce latency at the cost of a slight increase in model size.

Keywords: Binary Neural Network; inference engine; TinyML; Cortex-M microcontrollers; Internet of Things; edge computing

Citation: Sakr, F.; Berta, R.; Doyle, J.; Capello, A.; Dabbous, A.; Lazzaroni, L.; Bellotti, F. CBin-NN: An Inference Engine for Binarized Neural Networks. *Electronics* **2024**, *13*, 1624. https://doi.org/10.3390/electronics13091624

Academic Editor: Mehdi Sookhak

Received: 1 February 2024
Revised: 9 April 2024
Accepted: 23 April 2024
Published: 24 April 2024

1. Introduction

Shifting deep learning (DL)-based computer vision from cloud data centers to Internet of Things (IoT) devices is expected to offer benefits, including lower latency, reduced network requirements, and fewer privacy issues [1,2]. Currently, the most widespread edge devices are microcontrollers, which are typically characterized by energy efficiency and low costs [3]. The main challenges associated with these deployments are related to their computational and memory limitations. To cope with such limitations, techniques have been studied and solutions deployed aimed at reducing the computational and memory requirements of DL models. A common paradigm is quantization, which reduces the number of bits used to represent the weights and the activations (e.g., [4,5]). Binarized Neural Networks (BNNs) are an extreme type of DL model quantization with binary values of typically -1 and 1 [6]. This leads to a significant reduction in memory requirements (i.e., up to 32$\times$ [7]) and enables highly efficient inference with XNOR and PopCount operations for binary multiplication and accumulation (i.e., up to 58$\times$ faster inference [8]) at the price of a drop in accuracy, which can be accepted, at least in some target applications [7,8].

As the resource limitation of typical edge devices clashes with the large model size and huge computational overhead of DL models [9], specialized solutions and development tools have been developed for what is now called TinyML (i.e., machine learning on embedded IoT devices). TinyML concerns specific model architectures, training, and inference. Example tools include Google TensorFlow Lite Micro [10] and Qkeras [11], Larq [12], ARM CMSIS-NN [13], MicroTVM [14], MIT TinyEngine [15], and the STMicroelectronics STM32Cube.AI [16].

The goal of our research is to develop and study the performance of an inference engine specifically devoted to BNNs to favor the development of embedded AI applications on the edge and enable deployment on extremely constrained devices, such as mainstream 32-bit ARM Cortex-M microcontroller units (MCUs). The proposed library is written in platform-independent C [17] and can run BNN models on bare-metal devices, so it is intended to be seamlessly portable to most software-programmable edge devices.

The remainder of the paper is outlined in the following. Section 2 details related works in the literature, while Sections 3 and 4 present BNNs and the CBin-NN inference engine, respectively. Section 5 is devoted to the experimental setting and Section 6 to the obtained results. Finally, Section 7 draws conclusions and outlines possible directions for future research.

2. Related Works

In this section, we first describe the state-of-the-art inference engines for TinyML systems; then, we focus on solutions for BNN inference, and finally, we report on applications addressed through BNNs. Thorough, recent reviews of the state of the art on BNN research can be found in [18–21], which highlight the relevance of some now well-established engines that we introduce in the following.

2.1. TinyML Inference Engines

TensorFlow Lite Micro (TF-Lite Micro) [10] is an open-source framework developed by Google for running deep learning models on edge devices. ARM, on the other hand, has developed an open-source library for Cortex-M processors, namely CMSIS-NN [13], which maximizes performance and minimizes memory requirements of neural network applications.

MicroTVM [14] is a recent development that allows the TVM [22] compiler and deployment framework to be ported to Cortex-M7 and other MCUs. The MCUNet framework [23] combines a neural architecture search algorithm (TinyNAS) that optimizes the search space with a lightweight inference engine (TinyEngine) that controls resource management in a similar way to an operating system. In other words, the TinyEngine provides the essential code to run the customized neural network of the TinyNAS.

STMicroelectronics' X-Cube-AI is a free (closed-source) expansion package targeted at STM32 devices. It accepts pre-trained models from various frameworks, such as TensorFlow Lite, Keras, and PyTorch, and generates efficient code that can run on the STM32 series [16]. This toolkit also supports quantized models from the Keras framework to further optimize the inference phase in terms of latency and memory requirements. FANN-on-MCU [3] is an open-source toolkit that facilitates the deployment of efficient artificial neural networks built using the FANN open-source library [24]. The framework supports both ARM Cortex-M processors and RISC-V Parallel Ultra-Low Power processors (PULPs). Such libraries are tailored to quantized networks (e.g., 8-bit, 16-bit) and do not support binary data types.

2.2. BNN Inference Engines

The BMXNet-v2 library [25] is an open-source BNN library based on MXNet. The developed BNN layers can be seamlessly integrated with other standard library C components. The layers can also be used on both GPUs and CPUs. DaBNN [26] is an optimized inference engine that implements BNNs on mobile platforms. It was written in C++ and 64-bit ARM assembly.

To improve inference efficiency, several acceleration and memory refinement strategies have been developed, including bit packing, binarized convolution, and memory layout. Larq Compute Engine (LCE) [27] is an open-source inference framework developed by Plumerai for BNNs. LCE comes with hand-tuned binarized operators for the TensorFlow Lite runtime as well as a converter for TensorFlow model graphs. All current Android phones and tablets, as well as the Raspberry Pi 3 and 4, are among the 64-bit ARM devices

that LCE primarily targets. Although these frameworks are specifically designed for BNNs, they do not support inference on resource-constrained devices, such as ARM Cortex-M.

2.3. BNN Applications

TinyML has proven successful in a variety of fields, including computer vision, natural language processing (NLP), industry, and robotics. Some of these applications have been implemented on edge devices using BNNs. In the field of computer vision, Fasfous et al. [28] presented a BNN classifier, namely BinaryCoP, which was implemented on an embedded FPGA accelerator for detecting correct face mask wearing and positioning. Cerutti et al. [29] adopted an NLP application and implemented a BNN on the GAP8 microcontroller for sound event detection. Lo et al. [30] proposed an FPGA-based Binarized Convolutional Neural Network for cloud detection on a satellite payload, working on 8-bit images captured by a near-infrared camera sensor. Chung et al. [31] propose a real-time CNC machinery fault detection solution using a binary weight CNN working with vibrational signals. Pau et al. [32] compared two industry frameworks used to train BNNs, namely Larq and Qkeras. Two different models were implemented, one for human activity detection (computer vision) and the other for anomaly detection (industry). They reported inference time on ARM Cortex-A and Cortex-M CPUs using custom C functions designed for binary layer execution. Dabbous et al. [33] proposed a Spiking Neural Network (SNN) architecture for real-time tactile object shape recognition implemented on Raspberry Pi. Younes et al. [34] proposed a hybrid fixed-point CNN (H-CNN) implemented on an FPGA for robot touch modality classification.

The presented applications show the importance of BNNs in various fields, but their realization typically involves a custom hardware implementation (e.g., an FPGA design), and no open-source solutions are provided, which would support a much wider development and deployment by third parties. The state-of-the-art TinyML engines offer powerful quantization solutions that allow achieving ML performance not far from their original, unquantized counterparts [4]. Still, they lack facilities for binary-specific optimizations, particularly in terms of memory footprint. This was a major motivation for our exploration and overall work.

3. Binary Neural Networks

State-of-the-art Convolutional Neural Networks (CNNs) are often unsuitable for embedded systems with limited resources because of their huge model size and high computational cost. Quantization is an optimization technique useful for deploying CNNs on resource-constrained devices. Binarization is the extreme case of quantization. By restricting both activations and weights to $\{-1, +1\}$, a 32-time memory saving is achieved compared with 32-bit floating-point networks. The Sign function shown in Equation (1) is normally used to binarize the activations and weights in the forward propagation as follows:

$$\text{Sign}(x) = \begin{cases} +1, & if\ x \geq 1 \\ -1, & otherwise \end{cases} \tag{1}$$

Not only does binarization reduce memory requirements but it also simplifies computational logic. While CNNs use expensive multiplication and accumulation (MAC) operations for convolutional layers, binary convolution can be implemented using much simpler XNOR and PopCount (counts the number of 1s in a word) operations, as shown in Figure 1. X_R, W_R, X_B, and W_B are the real and binary values of activations and weights, respectively. To avoid the use of two bits, -1 is encoded as 0.

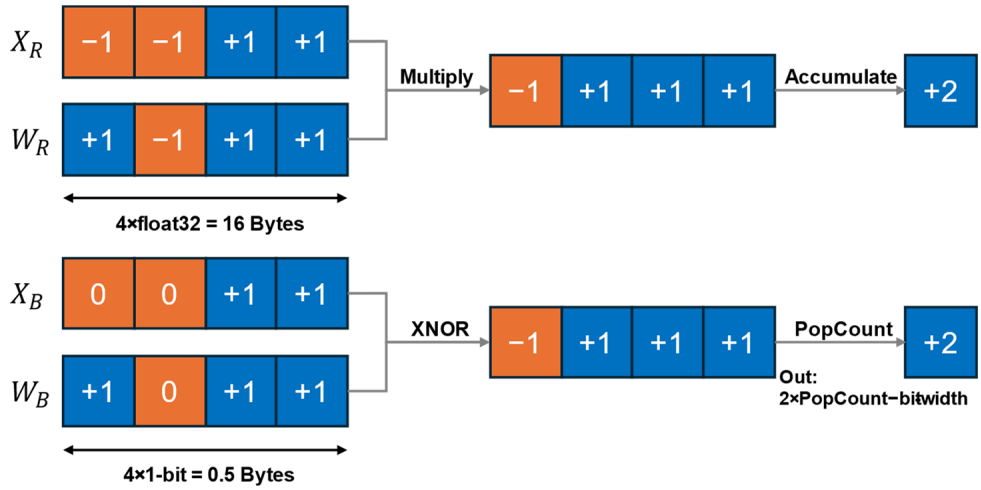

Figure 1. A convolution implemented as a MAC operation in float vs. binary XNOR and PopCount.

The binary weights of a BNN must first be learned through backpropagation. Training BNNs using the traditional gradient descent algorithm is not possible since the derivative of the sign function is equal to 0, where defined. The straight-through estimator (STE) was used to solve this issue [7], as shown in Figure 2. STE can be expressed as a clipped identity function as follows:

$$\frac{\partial X_B}{\partial X_R} = 1_{|X_R| \leq 1} \tag{2}$$

where X_R and X_B are the real-valued input and binarized output of the sign function, respectively, and $1_{|X_R| \leq 1}$ takes a value of 1 if $|X_R| \leq 1$ and is 0 otherwise (see Figure 2). The gradient of the cost function C with respect to the real-valued weights using the chain rule can be written as follows:

$$\frac{\partial C}{\partial X_R} = \frac{\partial C}{\partial X_B}\frac{\partial X_B}{\partial X_R} = \frac{\partial C}{\partial X_B} * 1_{|X_R \leq 1|} \tag{3}$$

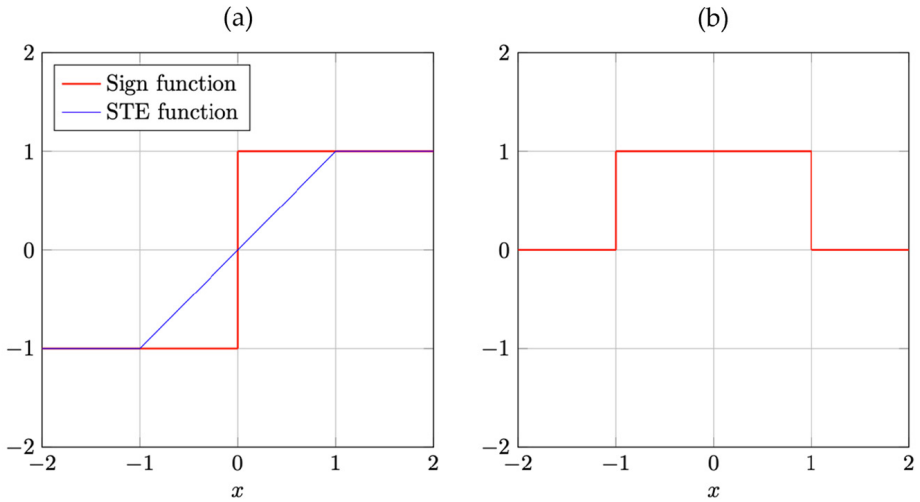

Figure 2. The sign and the STE function (**a**,**b**) its derivative that favors gradient descent [35].

To prevent the latent weights (i.e., the full precision weights) from getting out of control without affecting the binary weights, a clip function is also added as follows:

$$\text{clip}(W_R, -1, 1) = \max(-1, \min(1, W_R)) \tag{4}$$

Therefore, BNNs can be trained with the same gradient descent algorithms as real-valued CNNs using the techniques described above. Numerous optimization techniques have been proposed in recent years and have shown to be effective [36]. However, BNNs still suffer from accuracy degradation due to the severe information loss caused by parameter binarization, and the gap with their real-valued counterparts is still significant in several cases [37].

4. CBin-NN Inference Engine

We present CBin-NN, an open-source BNN inference engine for constrained devices. CBin-NN provides an optimized binarized implementation of all the basic CNN layer operators. The library is written in platform-independent C language to ensure seamless portability to most software-programmable edge devices. The project was built using the GCC compiler provided within STM32CubeIDE [38]. This GCC compiler is pre-configured for cross-compilation targeting the ARM Cortex-M architecture used by STM32 microcontrollers. During the development of CBin-NN, we did not limit the STM32CubeIDE environment but we also tested CBin-NN using GCC with MinGW in Windows. We verified that the produced assembly code was identical to that of the STM32CubeIDE framework. We also utilized the ARM GNU toolchain in Windows for cross-compilation, specifically targeting Cortex-M microcontrollers. These steps ensure that not only is the proposed library optimized for STM32 devices but it is also compatible with a wider range of development environments and target architectures.

The following subsections introduce the techniques we used to convert a real-value model for the inference phase, the operators implemented to execute the inference on the edge, and the optimizations made to increase inference efficiency.

4.1. Conversion to Inference Model

The smallest data type in the C language occupies 1 byte (8 bits). In the case of BNNs, each parameter occupies 1 bit owing to binarization. If BNN parameters were allocated as a single variable, 1 byte would be allocated for each of them, which would cancel out the memory savings compared to 8-bit quantized models. Bit packing is a common procedure for BNNs in which N elements are binarized into 1-bit each and then packed into an N-bit vector. In this way, XNOR can be performed directly between binarized vectors. CBin-NN uses bit packing so that the actual in-memory implementation achieves the ideal gain of an $8\times$ improvement over quantized 8-bit networks and a $32\times$ memory saving over the full-precision ones. To store the model parameters, bit packing is performed offline using a Python script that extracts the network parameters and transforms them into a suitable format for on-device inference. For the sake of compatibility with other frameworks that support BNN training (e.g., PyTorch), the script accepts an h5 model (trained with the Larq framework [39], see Section 5.2). As mentioned earlier, Larq BNN weights are also restricted to $W \in \{-1, 1\}$. To avoid using two bits, -1 is encoded as 0. The weights are then bit packed to a multiple of 32 across the input channels or padded if less than 32, to achieve the best memory access patterns on MCUs. Common BNNs already have a multiple of 32 channels in all layers, so no padding is performed in practice. During inference computation, CBin-NN binarizes the activations by extracting the sign bit, which is then packed into a multiple of 32 over the input channels to make optimal use of memory. Padding is applied when the size of the input is not a multiple of 32. Thus, the whole network operates on multiples of 32 input channels. The above-described workflow is depicted in Figure 3.

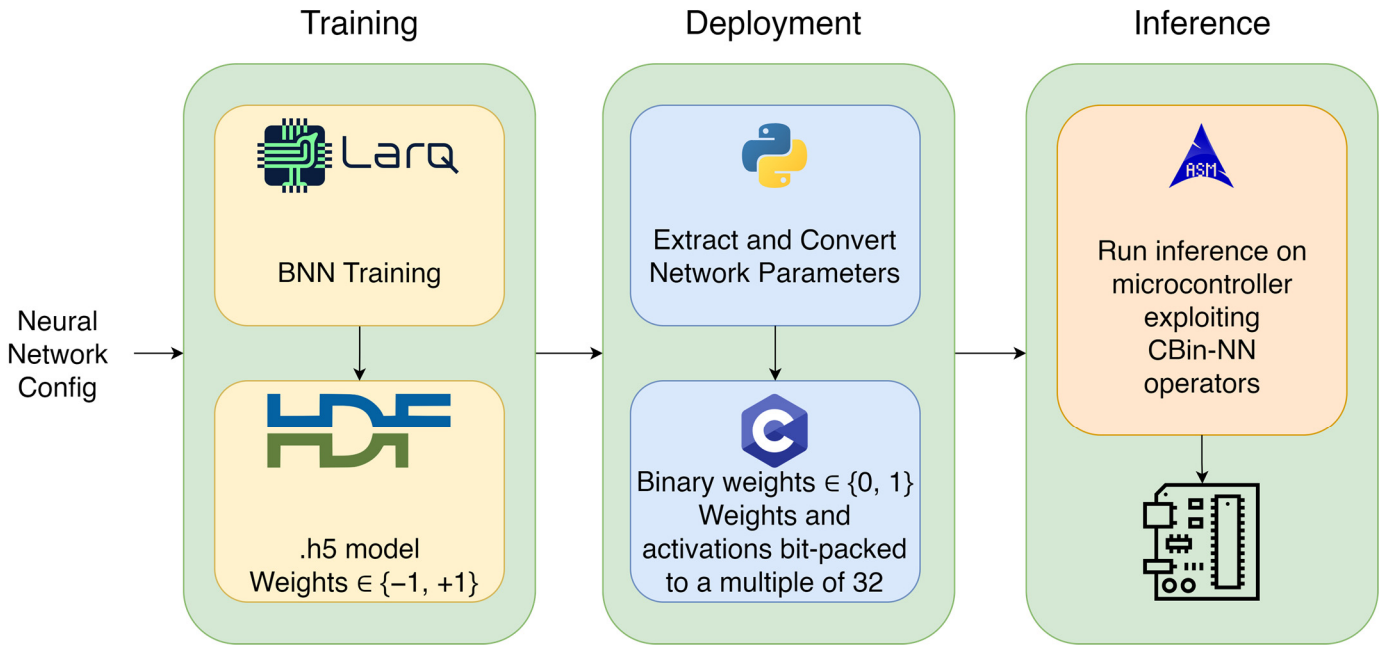

Figure 3. Workflow of BNN training, deployment, and inference on a microcontroller using CBin-NN.

4.2. CBin-NN Operators

The CBin-NN library provides a binarized implementation of the fundamental CNN layer operators, as described in the following (Table 1 provides an overview).

Table 1. CBin-NN operators.

Operator	Input	Weights	Output	Notes
QBConv2D [3]	8-bit quantized	32-bit packed	32-bit packed	Comparator + BN [1] Fusion
QBConv2D Optimized [3]	8-bit quantized	32-bit packed	32-bit packed	Comparator + LU [2] + BN Fusion
QBConv2D Optimized PReLU [3]	8-bit quantized	32-bit packed	32-bit packed	Comparator + LU + BN and PReLU Fusion
QQConv2d [3]	8-bit quantized	8-bit quantized	32-bit packed	Comparator + BN Fusion
QQConv2D Optimized [3]	8-bit quantized	8-bit quantized	32-bit packed	Comparator + LU + BN Fusion
QQConv2D Optimized PReLU [1]	8-bit quantized	8-bit quantized	32-bit packed	Comparator + LU + BN and PReLU Fusion
BBConv2D	32-bit packed	32-bit packed	32-bit packed	XOR and PopCount + BN Fusion
BBConv2D Optimized	32-bit packed	32-bit packed	32-bit packed	XOR and PopCount + LU + BN Fusion
BBConv2D Optimized PReLU	32-bit packed	32-bit packed	32-bit packed	XOR and PopCount + LU + BN and PReLU Fusion
BBPointwiseConv2D	32-bit packed	32-bit packed	32-bit packed	XOR and PopCount + BN Fusion
BBPointwiseConv2D Optimized	32-bit packed	32-bit packed	32-bit packed	XOR and PopCount + LU + BN Fusion
BBPointwiseConv2D Optimized PReLU	32-bit packed	32-bit packed	32-bit packed	XOR and PopCount + LU + BN & PReLU Fusion
bMaxPool2D	32-bit packed	-	32-bit packed	Bit-wise OR

Table 1. *Cont.*

Operator	Input	Weights	Output	Notes
bMaxPool2D Optimized	32-bit packed	-	32-bit packed	32-bit Simultaneous OR
BBFC	32-bit packed	32-bit packed	32-bit packed	Comparator + BN Fusion
BBFC Optimized	32-bit packed	32-bit packed	32-bit packed	XOR and PopCount + BN Fusion
BBFC Optimized PReLU	32-bit packed	32-bit packed	32-bit packed	XOR and PopCount + BN & PReLU Fusion
BBQFC [4]	32-bit packed	32-bit packed	Quantized [5] 8-, 16-, 32-bit)	Comparator + BN Fusion
BBQFC Optimized [4]	32-bit packed	32-bit packed	Quantized [5] (8-, 16-, 32-bit)	XOR and PopCount + BN Fusion
BBQFC Optimized PReLU [4]	32-bit packed	32-bit packed	Quantized [5] (8-, 16-, 32-bit)	XOR and PopCount + BN & PReLU Fusion

[1] BN stands for batch normalization. [2] LU stands for loop unrolling. [3] Usable as the first layer of a network. [4] Usable as the last layer of a network for classification. [5] The cumulative output for the classification layer. It is an integer that can be 8, 16, or 32 bits long.

A common practice in BNNs, for higher accuracy reasons, is to keep the first and last layers in full precision [40]. Since the inputs to the network are not binary, they must be treated by a specific operator. Thus, the first two operators that will be presented (i.e., QBConv2D and QQConv2D) are designed to be the first network layer, and it is up to the user to choose among them, depending on requirements and results.

4.2.1. QBConv2D

Quantized Input Binarized Kernel Convolution. QBConv2D receives quantized inputs and binary kernels and writes the corresponding bit-packed outputs. The convolution operation in this function is computed using a comparator. A weight of -1 (encoded as 0) leads to a negative accumulation of the inputs, while a weight of 1 leads to a positive accumulation of the inputs. In addition, QBConv2D supports layer fusion for batch normalization (BN) according to Equation (5) as follows:

$$y_i = \gamma\left(\frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}\right) + \beta \tag{5}$$

The parameters γ, β, μ, σ, and ϵ can be obtained after training, and x_i and y_i are the inputs and outputs of the BN layer. BN is applied after convolution and has been shown to be essential to train deep NNs [41]. We can rewrite Equation (5) as follows:

$$\begin{aligned} y_i &= \alpha_{2i}(x_i - \alpha_{1i}) \\ \alpha_{i1} &= \left(\mu - \frac{\sqrt{\sigma^2+\epsilon}}{\gamma}\beta\right) \\ \alpha_{i2} &= \frac{\gamma}{\sqrt{\sigma^2+\epsilon}} \end{aligned} \tag{6}$$

This fusion is applied to the accumulator values before storing them in memory. In this way, additional reads and writes are avoided, which would occur if the BN layer was treated separately. The resulting output activations are then binarized using the sign function, and finally, the binary activations are bit packed as described in Section 4.1.

4.2.2. QQConv2D

Quantized Input Quantized Kernel Convolution. This operator receives quantized inputs and quantized weights. It writes bit-packed outputs. Unlike QBCon2D, the weights are not binary but 8-bit quantized to increase accuracy. The weights are stored using the

"int8_t" data type, which occupies 1 byte of memory, which is the smallest allocatable memory in C. The convolution operation performed by QQConv2D harnesses processor-optimized MAC functionalities, indirectly accessed through higher-level C constructs for enhanced computational efficiency. This approach is more efficient than the comparator used in QBConv2D in terms of inference latency but it comes at the cost of a slight increase in model size. Likewise, BN fusion is supported in this function and follows the same steps as in QBConv2D. Finally, the final outputs are binarized and bit packed.

4.2.3. BBConv2D

Binary Input Binary Kernel Convolution. This operator accepts bit-packed inputs and weights. It writes bit-packed outputs. Binary convolution is implemented with XNOR and PopCount operations. Since Cortex-M processors do not support the XNOR operator, the XOR operator is used instead, and the result of PopCount is inverted as in Equation (7) as follows:

$$y_i = \sum_{i=0}^{\frac{c_{in}}{32}} N - 2 * \text{PopCount}(x_i \text{XOR} w_i) \tag{7}$$

where x_i, y_i, and w_i are the inputs, outputs, and weights for each convolutional step over input channels. In other words, a convolutional step for a $5 \times 5 \times c_{in}$ kernel is the first column and row of the kernel across channels $1 \times 1 \times c_{in}$ convolved over a $1 \times 1 \times c_{in}$ receptive field simultaneously. N is the bit width and c_{in} is the number of input channels. N is equal to 32 because the weights and activations are packed into multiples of 32 across the input channels. If the number of input channels is less than 32, N would correspond to the number of input channels before padding. Finally, this operator would perform up to thirty-two MACs with just four instructions, namely, XOR, PopCount, Multiply, and Subtract.

In the case of padded convolutions, which occur very frequently, the padded pixels are skipped in the calculation because they distort the results. This is because a padding value, which is usually 0, would be treated as -1 according to the bit packing specifications. Similarly, BBConv2d supports BN fusion. The BN layer can be approximated by adding an integer bias W′, which can be calculated after training according to [42]. The resulting activations are then binarized using the sign function and bit packed.

In the following, we detail the implementation of this binary convolution operator, which also serves as an example for the others. The C snippet below calculates the PopCount of the result obtained by XOR-ing (indicated by a '^') a weight and an input value. A weight is a 32-bit integer formed by packing 32 bits together across channels. "__builtin_popcount" is a compiler intrinsic function that counts the number of set bits (i.e., 1s) in a word.

C Code
sum = __builtin_popcount(weight[weight_idx] ^ input[input_idx]);

The corresponding assembly instructions generated by the GCC compiler load the weight value, perform the XOR operation, and call the PopCount function.

Assembly Code		
ldr.w	r5, [r3, r1, lsl #2]	; Load weight value
eors	r0, r5	; XOR operation between weight and input
bl	0x110 <__popcountsi2>	; Call pop count function

The pop_count variable is then incremented by the difference between the doubled cum and N (i.e., the word size in bits, which is 32 in our case) (cf. Figure 1, PopCount is inverted since we used XOR instead of XNOR).

C Code
pop_count += N—(sum << 1);

Assembly Code		
sub.w	r0, r9, r0, lsl #1	; Calculate N—(sum[idx] << 1)
add	r3, r0	; Add the result to pop_count[idx]

The C and assembly code snippets below compute the final output tensor value of a filter. If the value of conv_out (which is pop_count plus the filter's batch normalization factor) is greater than or equal to zero, SET_BIT is called on a specific bit in the out tensor. Otherwise, CLEAR_BIT is called. Every 32 conv_out values (i.e., activations in the resulting feature map), a 32-bit packing operation is performed to optimize storage.

C code
conv_out >= 0 ? SET_BIT(out[out_idx >> 5], out_idx & 31): CLEAR_BIT(out[out_idx >> 5], out_idx & 31);

Assembly Code
cmp.w r11, #0 ; compares the value in register r11 with zero bne.n 0xb0c <BBConv2D_Optimized_PReLU+1788>; Branch to address 0xb0c (set or clear bit macros)

The branch instruction redirects the program's execution to the C code segment, calling the macros for clearing or setting bits in the output tensor, which are implemented as follows.

C code
#define SET_BIT(var,pos) ((var) | = (1<<(pos))) #define CLEAR_BIT(var,pos) ((var) & = ~(1<<(pos)))

4.2.4. BBPointwiseConv2D

Binary Input Binary Kernel Pointwise Convolution. This operator has the same specifications as BBConv2D, with one minor change. The loops for kernel height and width are removed since they are equal to 1. This reduces the latency that would result from unnecessary loop branching.

4.2.5. BMaxPool2D

Binary Max Pooling. This function is applied to bit-packed inputs and simply computes a bitwise OR to calculate the binary max pool efficiently. A new feature compared to the previous version, which is found in [43], is that the operator OR is executed over the input channels rather than bitwise. This further accelerates the computation by running a 32-bit vector OR 32-bit vector instead of 1-bit OR 1-bit.

4.2.6. BBFC

Binary Input Binary Weights Fully Connected. This operator expects bit-packed inputs and weights. It writes bit-packed outputs. We optimize the previous version [43], which uses a comparator to implement the vector-matrix multiplication. The comparator is now replaced with the XOR and PopCount operators and performs 32 MACs at once. This is possible because the inputs to this layer are multiples of 32, as are the weights. This approach is more efficient than using a comparator that must check every bit in the input vector and speed up performance. BBFC supports BN fusion as in BBConv2D by adding an integer bias W', followed by binarization and bit packing.

4.2.7. BBQFC

Binary Input Binary Weights Quantized Output Fully Connected. Similar to BBFC, this operator expects bit-packed inputs and weights but writes quantized outputs for the classification layer. This layer is also optimized compared to the previous version [43] using the same approach as BBFC. The comparator is replaced by XOR and PopCount to improve performance. Similarly, this operator merges the BN layer by adding W' to the accumulator. The accumulated activations are then written to memory without binarization.

4.3. Operator Optimization

Loop unrolling is an optimization technique that aims at increasing instruction-level parallelism by reducing the number of iterations in a loop and increasing the amount of work performed per iteration. This approach aims to strike a balance between the increased instruction count and the reduced loop overhead for optimized performance in the convolutional operations discussed previously [44].

For the convolution operators, such as QBConv2D, QQConv2D, BBConv2D, and BBPointwiseConv2D, the first loop of the convolution representing the number of filters (i.e., output channels) is unrolled by a factor of 32. We unrolled this loop by this factor because network architectures commonly use multiples of 32 filters. This allows 32 filters to be processed simultaneously instead of processing each filter individually. Moreover, we have completely unrolled the loop that iterates over the input channels in QBConv2D and QQConv2D operators. It is possible to eliminate this loop since these operators correspond to the first network layer. Thus, the number of iterations is set before the execution and is equal to the number of channels in the input images. This is not possible with the other intermediate convolution operators (BBConv2D and BBPointwiseConv2D) because the number of input channels is variable. Moreover, the bit packing approach already reduces the computation of this loop by a multiple of 32, as a 32-bit binary weight vector is simultaneously convolved over a 32-bit binary receptive field.

5. Experimental Settings

5.1. Dataset

We tested the CBin-NN inference engine in a computer vision application, employing the CIFAR-10 [45] dataset. It consists of 60,000 color images 32 $\times$ 32 in size that are divided into 10 classes with 6000 images per class. A total of 50,000 images were used for training and 10,000 for testing. The dataset is smaller than others dedicated to computer vision (e.g., [46]) in terms of the number of both samples and classes, and we argue it is more representative of the TinyML embedded vision domain [47,48].

5.2. BNN Design and Training

As a reference architecture for the tests, we used SmallCifar (see Figure 4), a small network used in the literature [13,49] for the CIFAR dataset. It takes an image 32 $\times$ 32 in size that is passed through three convolutional layers with a kernel size of 5 $\times$ 5 as the input followed by a max pooling layer. As mentioned earlier, the BN layer is essential for deep learning training. Therefore, a BN layer is added after each convolutional layer. The output channels (i.e., number of filters) are 32, 32, and 64, respectively. The final feature maps are flattened and passed through a linear layer with weights of 1024 $\times$ 10 to obtain the class. The model is quite small and well-suited to embedded devices. To model and train the BNN, we used the Larq framework [39], an open-source Python library for training extremely low-precision networks. The network is trained for 500 epochs and a batch size of 50. To achieve high performance, we found it useful to resort to two main solutions concerning the binary optimizer and the activation functions, as detailed in the following.

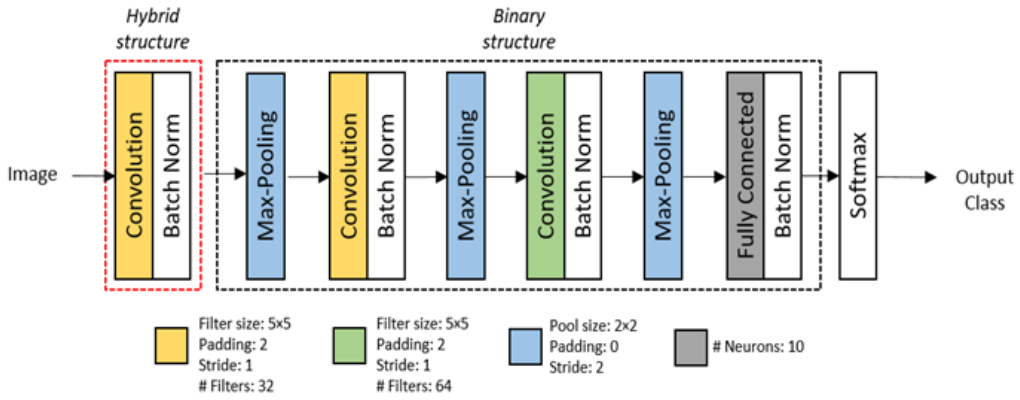

Figure 4. SmallCifar topology.

5.2.1. Binary Optimizer

For BNN training, we used the latent-free binary optimizer (Bop) introduced by [50]. Bop is specifically designed for BNNs and binary weight networks (BWNs). Bop has only one action available, which is to flip weights by changing the sign. It maintains an exponential moving average of gradients controlled by the adaptive learning rate γ. When this average exceeds a threshold τ, a weight is flipped. We set γ to 1×10^{-4} and τ to 1×10^{-8} during training.

5.2.2. Additional Activation Function (PReLU)

Kim et al. [51] argue that an unbalanced distribution of binary activations can improve the accuracy of BNNs. They showed that using an additional activation function (between the binary convolution layer and the following BN layer) makes the activation distribution unbalanced and thus improves accuracy. For this reason, we used an additional activation function (namely a Parametric Rectified Linear Unit (PReLU) as in [52]) after each convolutional and fully connected layer in the SmallCifar architecture. Results are discussed in the following subsection. We should emphasize that the operators described in Section 4.2 also support layer fusion for the PReLU activation function according to Equation (8) as follows:

$$\mathrm{PReLU}(x_i) = \begin{cases} a_i x_i, & if x \leq 0 \\ x_i, & if x > 0 \end{cases} \tag{8}$$

where x_i is the output activation and a_i is a learnable array with the same shape as x_i. To avoid excessive memory consumption for the convolutional layers, we have shared the parameters over the entire space, so that each filter has only one parameter [53].

5.3. Model Deployment

For model deployment and library CBin-NN testing, we employed the STM32F746 high-end commercial microcontroller [54], which is housed on a NUCLEO-144 board [55]. The MCU is an ARM Cortex-M7 core running at 216 MHz with 320 KB SRAM and 1 MB flash memory.

6. Results

This section presents the test results comparing our inference framework with other existing frameworks. The comparison is in terms of accuracy, latency, and memory requirements.

6.1. Accuracy

Table 2 shows the accuracy results of different implementations of the SmallCifar architecture. All available libraries are tailored to quantized models (typically 8-bit quantization), whereas CBin-NN is specialized for BNNs. The same 8-bit quantized model is deployed using state-of-the-art libraries ([10,13–15], TF-Lite Micro, CMSIS-NN, MicroTVM,

and TinyEngine, respectively), reaching a 79.90% accuracy. The basic CBin-NN implementation has lower accuracy because of binarization (12.6% less). Replacing the Adam optimizer with the Bop yields an accuracy of 71.90%. Adding the PReLU activation function after the convolutional and fully connected layers leads to 72.53%. Another performance improvement is obtained by maintaining the weights of the first network layer in a quantized 8-bit representation instead of binarization (76.12%). Combining all these optimizations, we achieve a 77.42% accuracy, which is only 2.5% less than the 8-bit quantized model.

Table 2. SmallCifar accuracy using different inference engines.

Engine	Accuracy	Notes
TF-lite Micro [10]	79.90%	-
CMSIS-NN [13]	79.90%	-
MicroTVM [14]	79.90%	-
TinyEngine [15]	79.90%	-
CBin-NN	67.30%	Adam Optimizer [1]
CBin-NN	71.90%	Bop Optimizer [1]
CBin-NN	72.53%	Bop + PReLU [1]
CBin-NN	76.12%	Bop + QQConv2D [2]
CBin-NN	77.42%	Bop + PReLU + QQConv2D [2]

[1] First layer weights are binarized and QBConv2D is used. [2] First layer weights are 8-bit quantized and QQConv2D is used.

6.2. Latency

The bar chart in Figure 5 shows that CBin-NN achieves higher inference efficiency than the other inference engines. Our library is 3.6× and 1.4× faster than TF-Lite Micro and MicroTVM, respectively. Compared to the CMSIS-NN library and TinyEngine, CBin-NN provided 20% and 15% lower inference latency, respectively. This latency result was obtained with the highest accuracy CBin-NN configuration (last row in Table 2). To calculate latency, we used the HAL_GetTick function in the STM32 HAL library.

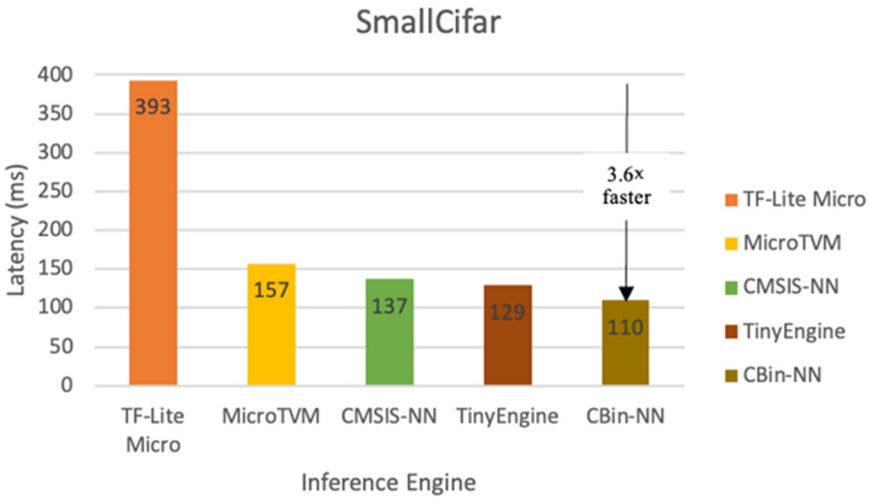

Figure 5. SmallCifar latency using various inference engines.

Table 3 analyzes the effects on latency and accuracy of the single optimization approaches. The inference time of the initial model, where the weights in the first layer are binarized, is 128 ms. In this case, the QBConv2D operator is used to compute the convolution. This operator uses a comparator, which increases the time complexity owing to its if-else instructions. The addition of the PReLU activation function comes at the expense of higher latency (2 ms slower) owing to the additional computations in Equation (8). On the other hand, when the weights of the first layer are 8-bit quantized rather than binary, the

QQConv2D operator is used during inference, which is more efficient than the QBConv2D operator since the classical MAC instructions are faster than comparators. This approach improves both accuracy and latency (76.12% and 107 ms, respectively) with a small increase in model size (see below). Similarly, the inference latency rises to 110 ms when a PReLU activation function is added, which is 3 ms slower than without PReLU.

Table 3. SmallCifar performance with different optimizations.

Optimization	Accuracy	Latency
Bop [1]	71.90%	128 ms
Bop + PReLU [1]	72.53%	130 ms
Bop + QQConv2D [2]	76.12%	107 ms
Bop + PReLU + QQConv2D [2]	77.42%	110 ms

[1] First layer weights are binarized and QBConv2D is used. [2] First layer weights are 8-bit quantized and QQConv2D is used.

6.3. Memory Footprint

Comparisons on model size are reported in Table 4. Our initial implementation is 7.5× smaller than our comparison frameworks (11.6 KB vs. 87.3 KB). This is mainly due to binarization, where each parameter occupies only 1 bit compared to 8 bits in the quantized representation. Adding the PReLU activation slightly increases the model size, as it requires storing additional parameters. To improve performance, the weights of the first layer are quantized instead of binarized (see Table 2). This increases the size of the model, as each parameter in the first layer now occupies 1 byte instead of 1 bit. This approach does not significantly affect the model size because the first layer usually has only three channels corresponding to the number of channels in the input images. Thus, the size of the largest model (quantized first layer and PReLU) increases to 14.2 KB. This, on the other hand, improves accuracy by approximately 5%.

Table 4. SmallCifar size using different frameworks.

Engine	Accuracy	Size	Notes
TF-Lite Micro	79.90%	87.34 KB	-
MicroTVM	79.90%	87.34 KB	-
CMSIS-NN	79.90%	87.34 KB	-
TinyEngine	79.90%	87.34 KB	-
CBin-NN	71.90%	11.58 KB	Bop [1]
CBin-NN	72.53%	12.12 KB	Bop + PReLU [1]
CBin-NN	76.12%	13.63 KB	Bop + QQConv2D [2]
CBin-NN	77.42%	14.17 KB	Bop + PReLU + QQConv2D [2]

[1] First layer weights are binarized and QBConv2D is used. [2] First layer weights are 8-bit quantized and QQConv2D is used.

In terms of peak memory (input and output activations [23] for the peak layer/block), CBin-NN reduces memory requirements by up to 28.8 times. This is mainly due to binarization, where activations are 32-bit packed. Figure 6 shows the difference between the available inference engines in terms of peak memory usage. CBin-NN is 12.8× and 28.8× more memory efficient compared to the TF-Lite Micro and MicroTVM libraries, respectively. Moreover, our proposed library requires 13.4× and 9.2× less memory than the CMSIS-NN and TinyEngine frameworks.

It is important to note that the achieved memory reductions enable the use of some NN models on extremely constrained devices. In the case of SmallCifar, we have tested the most accurate model (77.42%) on three additional microcontrollers (STM32F091RC, STM32F401RE, and STM32H743Z12, which represent the entry-level, mainstream, and high-performance families, respectively), with ROM ranging from 256 KB to 2000 KB and RAM ranging from 32 KB to 1000 KB. Table 5 shows the latency on each one of the mentioned

microcontrollers, which may also meet real-time constraints for some applications. Porting the models on the different devices is straightforward since the models are stored in .h5 files and our code is platform-independent C.

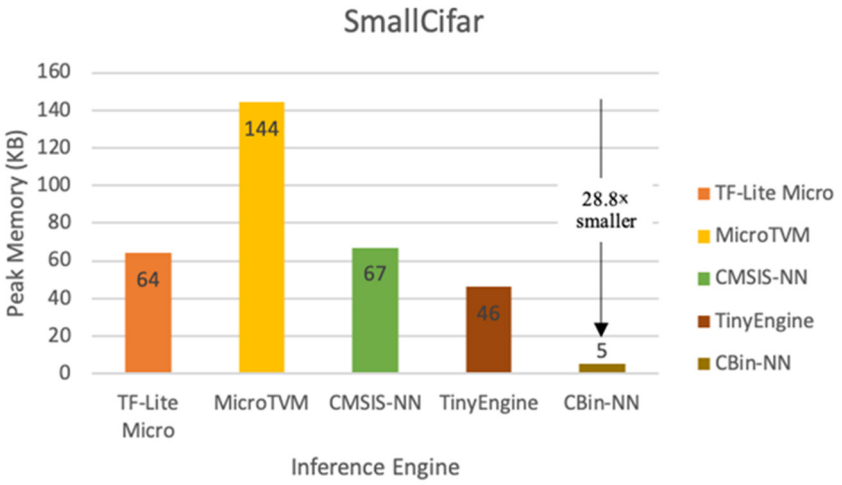

Figure 6. SmallCifar memory footprint using various inference engines.

Table 5. Microcontroller performance.

Optimization	Accuracy	Latency (ms)		
		F0	F4	H7
Bop + PReLU + QQConv2D	77.42%	925	450	58

7. Conclusions and Future Work

BNNs radically reduce the memory footprint compared to 8-bit quantized models and full precision models without a huge accuracy drop. They also reduce the computational cost thanks to simple XNOR and PopCount operations. To support the effective deployment of BNNs, we proposed a library of layer operators that facilitates simple yet flexible CNNs with binary weights and activations. CBin-NN has been developed in platform-independent C, supporting seamless porting to any software-programmable device, including affordable but extremely memory-limited devices, such as Cortex-M0. Experimental analysis on the STM32F746 MCU using the CIFAR-10 dataset shows that our library, employing some specific optimizations (e.g., channel-wise max pooling, Bop optimizer, PReLU additional activations, differentiated quantization for input layers), speeds up inference by 3.6 times and reduces the memory required to store model weights and activations by 7.5 times and 28 times, respectively, at the cost of slightly lower accuracy (2.5%). An ablation study assessing the impact of (i) a binary optimizer [50], (ii) an unbalanced distribution of binary activations obtained through an additional activation function [51], and (iii) a Quantized Input Quantized Kernel Convolution (QQConv2D) layer (with lower inference latency but slightly increased model size) stresses the importance of last factor to improve BNN accuracy.

CBin-NN is an open-source project within the ELM framework [56], available on GitHub at https://edge-learning-machine.github.io/CBin-NN/, accessed on 8 April 2024. More information can be found in [57]. We hope it can become a useful versatile toolkit for the IoT and TinyML R&D community to deploy binarized models.

As limitations, we cite two aspects that should be addressed in the next research steps. Our analysis does not focus on very large datasets, for which the extreme quantization brought by binarization typically involves a significant accuracy loss [36]. Also, we have not considered architectures specifically designed for mobile and embedded devices, such as MobileNet, because they tend to suffer from performance degradation during binarization due to their use of depth-wise convolutional layers [58]. Having demonstrated

the advantages of the proposed binarization technique in relatively simple convolutional neural layers, the approach could now be ported to more complex operations as well.

We believe that a major goal for future research is the design of a dedicated optimizer for BNNs to mitigate the performance degradation due to the gradient mismatch problem [59]. We expect that this should allow an architecture designed specifically for mobile/edge devices to achieve comparable accuracy to its 8-bit and full-precision counterparts. In addition, we plan to further enhance the efficiency of inference by optimizing the available operators. Specifically, we intend to employ Single Instruction Multiple Data (SIMD) in QQConv2D, leveraging the 8-bit quantization representation of both inputs and weights to streamline computation. Also, we plan to introduce an 8-bit packing technique to accommodate models with a number of channels divisible by eight. To evaluate CBin-NN's performance thoroughly, additional datasets should be used and a comprehensive hyperparameter study conducted to characterize the trade-off between accuracy, computational efficiency, and memory usage. Additionally, model interpretability techniques could be explored to provide a more detailed understanding of the inference engine's decision-making process.

Author Contributions: Conceptualization, F.S., R.B., J.D. and F.B.; methodology, F.S., R.B., J.D. and F.B.; software, F.S.; validation, A.C., A.D. and L.L.; data curation, F.S.; writing—original draft preparation, F.S. and F.B.; writing—review and editing, A.C., A.D., L.L. and F.B.; supervision, R.B., J.D. and F.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: CBin-NN is an open-source project available on GitHub at https://edge-learning-machine.github.io/CBin-NN/, accessed on 8 April 2024.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [CrossRef]
2. Branco, S.; Ferreira, A.G.; Cabral, J. Machine Learning in Resource-Scarce Embedded Systems, FPGAs, and End-Devices: A Survey. *Electronics* **2019**, *8*, 1289. [CrossRef]
3. Wang, X.; Magno, M.; Cavigelli, L.; Benini, L. FANN-on-MCU: An Open-Source Toolkit for Energy-Efficient Neural Network Inference at the Edge of the Internet of Things. *IEEE Internet Things J.* **2020**, *7*, 4403–4417. [CrossRef]
4. Rokh, B.; Azarpeyvand, A.; Khanteymoori, A. A Comprehensive Survey on Model Quantization for Deep Neural Networks in Image Classification. *ACM Trans. Intell. Syst. Technol.* **2023**, *14*, 97:1–97:50. [CrossRef]
5. Gholami, A.; Kim, S.; Dong, Z.; Yao, Z.; Mahoney, M.W.; Keutzer, K. *A Survey of Quantization Methods for Efficient Neural Network Inference*; CRC: Boca Raton, FL, USA, 2021.
6. Courbariaux, M.; Hubara, I.; Soudry, D.; El-Yaniv, R.; Bengio, Y. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or −1. *arXiv* **2016**, arXiv:1602.02830.
7. Courbariaux, M.; Bengio, Y.; David, J.-P. BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations. *Adv. Neural Inf. Process. Syst.* **2015**, *28*, 3123–3131. [CrossRef]
8. Rastegari, M.; Ordonez, V.; Redmon, J.; Farhadi, A. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*; Springer International Publishing: Cham, Switzerland, 2016. [CrossRef]
9. Li, Y.; Bao, Y.; Chen, W. Fixed-Sign Binary Neural Network: An Efficient Design of Neural Network for Internet-of-Things Devices. *IEEE Access* **2020**, *8*, 164858–164863. [CrossRef]
10. David, R.; Duke, J.; Jain, A.; Reddi, V.J.; Jeffries, N.; Li, J.; Kreeger, N.; Nappier, I.; Natraj, M.; Regev, S.; et al. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *Proc. Mach. Learn. Syst.* **2020**, *3*, 800–811. [CrossRef]
11. Coelho, C.N., Jr.; Kuusela, A.; Li, S.; Zhuang, H.; Aarrestad, T.; Loncar, V.; Ngadiuba, J.; Pierini, M.; Pol, A.A.; Summers, S. Automatic Heterogeneous Quantization of Deep Neural Networks for Low-Latency Inference on the Edge for Particle Detectors. *Nat. Mach. Intell.* **2021**, *3*, 675–686. [CrossRef]
12. Larq | Binarized Neural Network Development. Available online: https://larq.dev/ (accessed on 31 January 2024).
13. Lai, L.; Suda, N.; Chandra, V. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *arXiv* **2018**, arXiv:1801.06601. [CrossRef]
14. microTVM: TVM on Bare-Metal—Tvm 0.16.Dev0 Documentation. Available online: https://tvm.apache.org/docs/topic/microtvm/index.html (accessed on 19 March 2024).

15. Lin, J.; Chen, W.-M.; Lin, Y.; Cohn, J.; Gan, C.; Han, S. MCUNet: Tiny Deep Learning on IoT Devices. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 11711–11722. [CrossRef]
16. X-CUBE-AI–AI Expansion Pack for STM32CubeMX–STMicroelectronics. Available online: https://www.st.com/en/embedded-software/x-cube-ai.html (accessed on 29 January 2024).
17. Standards (Using the GNU Compiler Collection (GCC)). Available online: https://gcc.gnu.org/onlinedocs/gcc/Standards.html (accessed on 29 January 2024).
18. Sayed, R.; Azmi, H.; Shawkey, H.; Khalil, A.H.; Refky, M. A Systematic Literature Review on Binary Neural Networks. *IEEE Access* **2023**, *11*, 27546–27578. [CrossRef]
19. Yuan, C.; Agaian, S.S. A Comprehensive Review of Binary Neural Network. *Artif. Intell. Rev.* **2023**, *56*, 12949–13013. [CrossRef]
20. Zhao, W.; Ma, T.; Gong, X.; Zhang, B.; Doermann, D. A Review of Recent Advances of Binary Neural Networks for Edge Computing. *IEEE J. Miniat. Air Space Syst.* **2021**, *2*, 25–35. [CrossRef]
21. Qin, H.; Gong, R.; Liu, X.; Bai, X.; Song, J.; Sebe, N. Binary Neural Networks: A Survey. *Pattern Recognit.* **2020**, *105*, 107281. [CrossRef]
22. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Cowan, M.; Shen, H.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018. [CrossRef]
23. Lin, J.; Chen, W.-M.; Cai, H.; Gan, C.; Han, S. MCUNetV2: Memory-Efficient Patch-Based Inference for Tiny Deep Learning. *arXiv* **2021**, arXiv:2110.15352. [CrossRef]
24. Magno, M.; Cavigelli, L.; Mayer, P.; Hagen, F.V.; Benini, L. FANNCortexM: An Open Source Toolkit for Deployment of Multi-Layer Neural Networks on ARM Cortex-M Family Microcontrollers: Performance Analysis with Stress Detection. In Proceedings of the 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), Limerick, Ireland, 15–18 April 2019; pp. 793–798.
25. Bethge, J.; Bartz, C.; Yang, H.; Meinel, C. BMXNet 2: An Open Source Framework for Low-Bit Networks–Reproducing, Understanding, Designing and Showcasing. In Proceedings of the 28th ACM International Conference on Multimedia, Seattle, WA, USA, 12 October 2020; pp. 4469–4472.
26. Zhang, J.; Pan, Y.; Yao, T.; Zhao, H.; Mei, T. daBNN: A Super Fast Inference Framework for Binary Neural Networks on ARM Devices. In Proceedings of the 27th ACM International Conference on Multimedia, Nice, France, 21–25 October 2019. [CrossRef]
27. Bannink, T.; Bakhtiari, A.; Hillier, A.; Geiger, L.; de Bruin, T.; Overweel, L.; Neeven, J.; Helwegen, K. Larq Compute Engine: Design, Benchmark, and Deploy State-of-the-Art Binarized Neural Networks. *Proc. Mach. Learn. Syst.* **2021**, *3*, 680–695. [CrossRef]
28. Fasfous, N.; Vemparala, M.-R.; Frickenstein, A.; Frickenstein, L.; Stechele, W. BinaryCoP: Binary Neural Network-Based COVID-19 Face-Mask Wear and Positioning Predictor on Edge Devices. In Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Portland, OR, USA, 17–21 June 2021. [CrossRef]
29. Cerutti, G.; Andri, R.; Cavigelli, L.; Magno, M.; Farella, E.; Benini, L. Sound Event Detection with Binary Neural Networks on Tightly Power-Constrained IoT Devices. In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, Virtual, 26–28 July 2021. [CrossRef]
30. Lo, C.-Y.; Lee, P.-J.; Bui, T.-A. H-BNN: FPGA-Based Binarized Convolutional Neural Network for Cloud Detection on Satellite Payload. In Proceedings of the 2023 International Conference on System Science and Engineering (ICSSE), Ho Chi Minh, Vietnam, 27 July 2023; pp. 190–193.
31. Chung, C.-C.; Liang, Y.-P.; Chang, Y.-C.; Chang, C.-M. A Binary Weight Convolutional Neural Network Hardware Accelerator for Analysis Faults of the CNC Machinery on FPGA. In Proceedings of the 2023 International VLSI Symposium on Technology, Systems and Applications (VLSI-TSA/VLSI-DAT), HsinChu, Taiwan, 17 April 2023; pp. 1–4.
32. Pau, D.; Lattuada, M.; Loro, F.; De Vita, A.; Domenico Licciardo, G. Comparing Industry Frameworks with Deeply Quantized Neural Networks on Microcontrollers. In Proceedings of the 2021 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 10 January 2021; pp. 1–6.
33. Dabbous, A.; Ibrahim, A.; Alameh, M.; Valle, M.; Bartolozzi, C. Object Contact Shape Classification Using Neuromorphic Spiking Neural Network with STDP Learning. In Proceedings of the 2022 IEEE International Symposium on Circuits and Systems (ISCAS), Austin, TX, USA, 27 May–1 June 2022; pp. 1052–1056.
34. Younes, H.; Ibrahim, A.; Rizk, M.; Valle, M. Hybrid Fixed-Point/Binary Convolutional Neural Network Accelerator for Real-Time Tactile Processing. In Proceedings of the 2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS), Dubai, United Arab Emirates, 28 November 2021; pp. 1–5.
35. de Putter, F.; Corporaal, H. How to Train Accurate BNNs for Embedded Systems? *arXiv* **2022**, arXiv:2206.12322. [CrossRef]
36. Xu, S.; Li, Y.; Wang, T.; Ma, T.; Zhang, B.; Gao, P.; Qiao, Y.; Lv, J.; Guo, G. Recurrent Bilinear Optimization for Binary Neural Networks. *arXiv* **2022**, arXiv:2209.01542. [CrossRef]
37. Su, Y.; Seng, K.P.; Ang, L.M.; Smith, J. Binary Neural Networks in FPGAs: Architectures, Tool Flows and Hardware Comparisons. *Sensors* **2023**, *23*, 9254. [CrossRef]
38. STM32CubeIDE–Integrated Development Environment for STM32–STMicroelectronics. Available online: https://www.st.com/en/development-tools/stm32cubeide.html (accessed on 13 February 2024).
39. Plumerai Getting Started. Available online: https://docs.larq.dev/larq/ (accessed on 29 January 2024).
40. Bulat, A.; Martinez, B.; Tzimiropoulos, G. BATS: Binary ArchitecTure Search. *arXiv* **2020**, arXiv:2003.01711. [CrossRef]

41. Bjorck, N.; Gomes, C.P.; Selman, B.; Weinberger, K.Q. Understanding Batch Normalization. In *Proceedings of the Advances in Neural Information Processing Systems*; Curran Associates, Inc.: New York, NY, USA, 2018; Volume 31.
42. Yonekawa, H.; Nakahara, H. On-Chip Memory Based Binarized Convolutional Deep Neural Network Applying Batch Normalization Free Technique on an FPGA. In Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Orlando/Buena Vista, FL, USA, 29 May–2 June 2017; pp. 98–105.
43. Sakr, F.; Berta, R.; Doyle, J.; Younes, H.; De Gloria, A.; Bellotti, F. Memory Efficient Binary Convolutional Neural Networks on Microcontrollers. In Proceedings of the 2022 IEEE International Conference on Edge Computing and Communications (EDGE), Barcelona, Spain, 10–16 July 2022; pp. 169–177.
44. Cong, J.; Yu, C.H. Impact of Loop Transformations on Software Reliability. In Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, 2–6 November 2015; pp. 278–285.
45. Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. 2009. Available online: https://www.google.com.hk/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf&ved=2ahUKEwjvtf6K09eFAxUgplYBHd82Cx4QFnoECBUQAQ&usg=AOvVaw3mtyV-hQ1QzJ5miMbeD6T8 (accessed on 23 April 2024).
46. Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; pp. 248–255.
47. Alajlan, N.N.; Ibrahim, D.M. TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications. *Micromachines* **2022**, *13*, 851. [CrossRef] [PubMed]
48. Kavi, B.O.; Zeebaree, S.R.M.; Ahmed, O.M. Deep Learning Models Based on Image Classification: A Review. *Int. J. Sci. Bus.* **2020**, *4*, 75–81. [CrossRef]
49. Weber, L.; Reusch, A. TinyML–How TVM Is Taming Tiny. Available online: https://tvm.apache.org/2020/06/04/tinyml-how-tvm-is-taming-tiny (accessed on 29 January 2024).
50. Helwegen, K.; Widdicombe, J.; Geiger, L.; Liu, Z.; Cheng, K.-T.; Nusselder, R. Latent Weights Do Not Exist: Rethinking Binarized Neural Network Optimization. *arXiv* **2019**, arXiv:1906.02107. [CrossRef]
51. Kim, H.; Park, J.; Lee, C.; Kim, J.-J. Improving Accuracy of Binary Neural Networks Using Unbalanced Activation Distribution. *arXiv* **2020**, arXiv:2012.00938. [CrossRef]
52. Tang, W.; Hua, G.; Wang, L. How to Train a Compact Binary Neural Network with High Accuracy? *AAAI* **2017**, *31*, 10862. [CrossRef]
53. Team, K. Keras Documentation: PReLU Layer. Available online: https://keras.io/api/layers/activation_layers/prelu/ (accessed on 29 January 2024).
54. STM32F746NG–High-Performance and DSP with FPU, Arm Cortex-M7 MCU with 1 Mbyte of Flash Memory, 216 MHz CPU, Art Accelerator, L1 Cache, SDRAM, TFT–STMicroelectronics. Available online: https://www.st.com/en/microcontrollers-microprocessors/stm32f746ng.html (accessed on 18 March 2024).
55. NUCLEO-F429ZI–STM32 Nucleo-144 Development Board with STM32F429ZI MCU, Supports Arduino, ST Zio and Morpho Connectivity–STMicroelectronics. Available online: https://www.st.com/en/evaluation-tools/nucleo-f429zi.html (accessed on 18 March 2024).
56. Sakr, F.; Bellotti, F.; Berta, R.; De Gloria, A. Machine Learning on Mainstream Microcontrollers. *Sensors* **2020**, *20*, 2638. [CrossRef] [PubMed]
57. Sakr, F. *Tiny Machine Learning Environment: Enabling Intelligence on Constrained Devices*; Queen Mary University of London: London, UK, 2023.
58. Phan, H.; Huynh, D.; He, Y.; Savvides, M.; Shen, Z. MoBiNet: A Mobile Binary Network for Image Classification. *arXiv* **2019**, arXiv:1907.12629. [CrossRef]
59. Lin, D.D.; Talathi, S.S. Overcoming Challenges in Fixed Point Training of Deep Convolutional Networks. *arXiv* **2016**, arXiv:1607.02241. [CrossRef]