Deploying Machine Learning in Resource-Constrained Devices for Human Activity Recognition

Rafael Schild Reusch, Leonardo Rezende Juracy, Fernando Gehm Moraes
PUCRS – School of Technology, Porto Alegre, Brazil
{rafael.reusch, leonardo.juracy}@edu.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—Machine Learning (ML) has proven to be highly effective in solving complex tasks such as human activity and speech recognition. However, the introduction of accuracy-driven ML models has brought new challenges regarding their applicability in resource-constrained systems. In Human Activity Recognition (HAR), current state-of-the-art approaches often rely on complex multilayer LSTM (Long Short Term Memory) networks once they are well suited to handle temporal series data, a crucial aspect of HAR, but presenting a high computational cost associated with running the inference phase. In HAR, low-power IoT devices, such as wearable sensor arrays, are frequently used as data-gathering devices. However, we observed a limited effort to deploy ML technology directly on these devices, most commonly using edge or cloud computing services, which can be unavailable in some situations. This work aims to provide a Convolutional Neural Network (CNN) tuned for resource-constrained embedded systems. After tuning the CNN model in the Pytorch framework, we present an equivalent C model and employ optimization techniques. The results show that, compared to the reference CNN, the optimized model reduced the CNN model 2.34 times, does not require floating-point units (FPUs), and improved accuracy from 74.9% to 85.2%. These results show that running the proposed CNN on resource-constrained devices is possible.**

***Index Terms*—Machine Learning, 1D CNN, Human Activity Recognition, Embedded Systems, Constrained Devices**

I. Introduction

Human Activity Recognition (HAR) is a research area that has received significant attention in recent years, thanks to the proliferation of mobile phones and IoT sensors equipped with complex sensor arrays such as accelerometers and magnetometers. Embedded devices have been identified as crucial components in HAR. HAR data can be gathered through wearable sensors such as smartwatches and bracelets. With the ubiquity of smartphones, even within the elderly demographic, recent studies have used these devices to record and categorize human activity [1].

Machine Learning (ML) has effectively solved various tasks in industries, especially areas with complex correlations, such as speech recognition and medical diagnosis. Recent studies proposed different neural network models applied in HAR tasks, including complex architectures with low real-life applicability due to the required computational costs [2]. This is an issue in various ML tasks. More complex tasks, such as speech recognition for virtual assistants, are mainly cloud-based [3], where local devices only function as input data for the neural network located off-site on servers dedicated to ML.

Speech recognition is similar to HAR since both contain high temporal correlation, where past processed data affect the current output. Long-short-term memory (LSTM) neural networks have been widely used in these tasks and currently represent state-of-the-art accuracy in speech recognition and HAR. LSTM contains complex dependencies that require resources scarce in most embedded systems. Recent approaches using battery-powered devices suggest transferring the task of processing the inference to devices with more power availability, using embedded devices only for data gathering [1]. These proposals neglect the situation where edge devices are unavailable, creating gaps in the processed data array and increasing the probability that the system does not detect emergencies in real-time.

The *objective* of this work is to provide a CNN tuned for resource-constrained embedded systems, using optimization and efficient neural network modeling techniques. Specific goals include:

1) Develop a C model for a reference CNN, once C language brings advantages like a small memory footprint, and compact and efficient code generation;
2) Optimize the CNN C model for power-constrained embedded devices;
3) Optimize the CNN C model in terms of memory footprint.

This paper is organized as follows. Section II presents the state-of-the-art of ML in HAR tasks, positioning this work w.r.t. the literature. Section III presents the author's previous work, presented in [4], which includes the CNN reference model and optimizations made using the Pytorch framework. Section IV presents the C model and the evaluation of using integer representation. Section V presents the method adopted to compress the CNN parameters, weights, and bias to reduce the CNN model size. Section VI concludes this paper, pointing out directions for future work.

II. Related Work

Table I summarizes the related work, where each row color corresponds to an ML method: LSTM in red, CNN in green, and hybrid approaches of CNN and LSTM in blue. The last row presents our work compared to the literature. We observe

This work was financed in part by CAPES (Finance Code 001), CNPq (grants 309605/2020-2 and 407829/2022-9), and FAPERGS (grant 21/2551-0002047-4).

TABLE I
RELATED WORK ON MACHINE LEARNING APPLIED FOR HAR.

Work	Model	Modeling	Goals
LSTM Networks Using Smartphone Data for Sensor-Based Human Activity Recognition in Smart Homes [5]	LSTM	TensorFlow	Compare different LSTM approaches
Human activity recognition from inertial sensor time-series using batch normalized deep LSTM recurrent networks [6]	LSTM	Keras Python	LSTM accuracy for temporal correlation analysis
Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors [1]	CNN	not defined by the Authors	Novel CNN approach to achieve state-of-the-art accuracy in HAR problem
A Fully Onchip Binarized Convolutional Neural Network FPGA Implementation with Accurate Inference [7]	Binarized CNN	Pytorch	State-of-the-art accuracy in a FPGA using low-power and low-area techniques
Human Activity Recognition Using Cascaded Dual Attention CNN and Bi-Directional GRU Framework [8]	CNN + GRU	TensorFlow	Human activity recognition in video streams
Inception inspired CNN-GRU hybrid network for human activity recognition [9]	CNN + GRU	TensorFlow	Develop a HAR model that is reasonably accurate and less complex so that it can be later deployed in embedded devices
Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition [10]	CNN + LSTM	Lasagne	Improve accuracy of LSTM approach using CNN as Feature Extractor
Towards effective detection of elderly falls with CNN-LSTM neural networks [11]	CNN + LSTM	not defined by the Authors	Accurate fall detection using wearable sensors in the elderly population
MultiCNN-FilterLSTM: Recognition of human activity based on sensor-based resources in IoT applications [2]	CNN + LSTM	Pytorch	Support IoT systems that require a resource-efficient model
A novel vision-based fall detection scheme using keypoints of human skeleton with long short-term memory network [12]	CNN + LSTM	not defined by the Authors	Solution for fall detection using vision-based approaches
This work	1D CNN	Pytorch and C models	Integer representation with reduced model size targing processors with limited resources

a lack of proposals using IoT devices as the target device for ML inference. Optimization techniques were proposed, for example, by Gholami et al. [13], minimizing accuracy loss without power and memory footprint analyses, key parameters for embedded systems. Mekruksavanich et al. [5] mention using smartphones as a data capture source, but little effort is made to use them as the target for the inference phase.

Many of the proposed models [2, 6, 10, 11, 12, 14] consist of complex, stacked, LSTM structures, which require the use of resources that are scarce in embedded systems, such as memory area. On the one hand, these works achieved state-of-the-art accuracy for many datasets. The same occurs for Ullah et al. [8] and Dua et al. [9], but using a combination of CNN and GRU. However, they do not apply to battery-powered devices that require an edge or cloud device to process this information. In practice, edge or cloud devices are not available everywhere or always, creating the risk of not detecting an emergency, such as a hard fall event.

III. OPTIMIZED 1D CNN MODEL

This section presents the author's previous work [4], presented in SBESC 2022, which includes the CNN reference model and optimizations made using the Pytorch framework.

Figure 1 presents the 1D CNN baseline model [15]. It contains three convolutional layers and two fully connected (FC) layers. This model adopts the rectified linear activation function (ReLU) after each convolution [16]. It is a relatively simple architecture by today's standards. Its goal is not to achieve state-of-the-art accuracy but to demonstrate the effectiveness of simple CNN models in HAR tasks. This CNN uses a public dataset, similar to the MobiAct [17] dataset, containing 20,000 sensor readings from 6 people, each performing five different actions.

The major disadvantage of this 1D CNN model is the lack of temporal analysis. Human activities are highly related to previous and future movements. Although temporal characteristics are embedded in sensor data, it is insufficient to detect complex human activities accurately. Recent approaches [1, 11, 14] process data in time frames extracted from raw

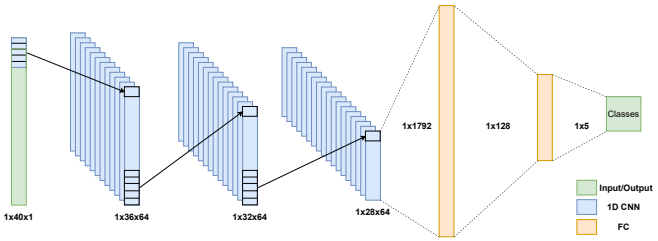


Fig. 1. 1D CNN Reference Model [4].

sensor data with a sliding-window method. These approaches increase accuracy by reducing the effect of unintended data (noise or random human motions) on the readings.

Our previous work [4] explored the design space of the CNN model, including the FC size, kernel size, number of convolutional layers, and the inclusion of temporal awareness. The main contribution of [4] was the addition of the temporal awareness to the reference model, keeping its reduced complexity, resulting in the *Extended 1D model – **Ext1D***.

The ***Ext1D*** model modified the convolutional layer to include multiple time steps simultaneously. The original model takes a single time step as input (vector with 40 elements), resulting in a 1x28 feature map (3rd convolutional layer). Combined with the number of filters (64), the flatten layer transforms these multiple vectors into a single 1x1,792 vector (Figure 1). In the ***Ext1D*** model, the new time frame has 120 elements (3 temporal time steps). Using the ***Ext1D*** model, the size of the resulting feature map is 1x108, which contains approximately 3.8 times more information than the reference model, improving accuracy. The fully connected (FC) input size is now 1x6,912 to accommodate this larger feature map. Despite the increased number of weights in the FC layer, it requires fewer weights than other evaluated optimizations while achieving higher accuracy.

The baseline model achieves 74% accuracy in activities randomly selected from the dataset, having 271,621 parameters. The ***Ext1D*** model reached an accuracy of 89%, however, the number of parameters increased to 926,981. The ***Ext1D*** model was modeled in the PyTorch [18] framework.

IV. Integer C model

This section presents how we converted the PyTorch ***Ext1D*** model to a C model using integer representation, avoiding floating-point (FP) representation. FP units (FPUs) are expensive cores in processors and should be avoided in devices that are energy and area restricted [19].

The PyTorch framework is responsible for: (i) generating C header files with input features, weights, and bias; (ii) providing the reference accuracy (gold model). The model described in C language (*C model*) executes only the inference phase using header files created by "ConvertTensor.py". The goal is to have a code in C language that runs only the CNN model without preprocessing files or performing normalization computations. The *C model* generates for each inference the tuple {*correct class, inferred class*}. An auxiliary program reads the generated data, computes the accuracy, and generates the confusion matrix, as presented in Listing 1.

Listing 1. Accuracy evaluation and confusion matrix generated after the inference phase for 3,923 samples, with a batch size equal to 32.

```
TOTAL SAMPLES: 3923
-----------------------
Accuracy: 88.5
-----------------------
Correct predictions: 615
Wrong predictions: 80
-----------------------
Confusion Matrix:
      0       1       2       3       4
0:    123     1       1       0       0
1:    17    123       6       0       6
2:    0       5     123       2       0
3:    0       0      33     123       2
4:    0       0       3       4     123
```

PyTorch considers the batch size to compute the accuracy value, which contains 32 samples. Thus, for this example, we obtain 123 sets ($\lceil \frac{3923}{32} \rceil = 123$). As a result, the maximum number displayed in the confusion matrix is 123 because, out of the 123 evaluated sets, there was at least one correct prediction for a given class in one of the 32 samples. For example, in Listing 1, the horizontal row for class 1 shows the detection of 29 sets ($17 + 6 + 6$) of a class different than 1, but in all sets, there was at least one correct detection in this class. PyTorch computes the accuracy using the formula ($\frac{correct\ predictions}{correct\ pred.\ +\ wron\ pred.}$), resulting in this example in an accuracy of 88.5% ($\frac{615}{615+80}$).

A. Description of CNN Layers

We implemented the *C model* without using standard libraries or function calls. The goal is to improve the performance of the *C model*. The training phase uses layers not required during inference, such as the Dropout Layer. Therefore, we did not include these layers in the *C model*.

Two convolution models were implemented. The first model receives a unidimensional vector corresponding to the raw data extracted from the dataset as input. This raw data input has a shape of 1×120. Considering that the input vector has 120 entries, `KERNEL_SIZE`=5, and the number of filters equals to 64, this first layer executes 37,120 MAC (multiply-accumulate) operations ($64 \times 116 \times 5$).

The 2nd and 3rd convolutional layers use the second convolution model, detailed in Listing 2. This model uses a bidimensional input shape of 64×116, and 64×112 in the 2nd and 3rd layers. The first dimension, 64, corresponds to the number of filters. The second parameter corresponds to the input features size minus ($kernelsize - 1$). The external loops (lines 1 and 2) iterate through the output filters (`filterToGenerate`) and output vectors (`inputOffset`). Line 3 initializes the counter used to generate one result. The two inner loops (lines 4 and 5) execute the kernel operation. The number of MAC operations is equal to 2,293,760 ($64 \times 112 \times 64 \times 5$) and 2,211,840 ($64 \times 108 \times 64 \times 5$) for the 2nd and 3rd layers, respectively.

Listing 2. 1D convolution layer, using a bidimensional input shape as input.

```
1 for (int filterToGenerate=0; filterToGenerate<NUM_FILTERS; filterToGenerate++ ){
2    for (int inputOffset = 0; inputOffset < CONV3_INPUT_SIZE-4; inputOffset++){
3      conv3_totalSum = 0;
4      for (int filterIn = 0; filterIn < NUM_FILTERS; filterIn++){
5        for (int kernelIndex = 0; kernelIndex < KERNEL_SIZE  kernelIndex++){
6          int weightIndex = kernelIndex + (filterIn * KERNEL_SIZE) +
   ↪  (filterToGenerate * NUM_FILTERS * KERNEL_SIZE) ;
7          int indexIn = kernelIndex + inputOffset;
8          conv3_totalSum += conv0_featureMap[filterIn][indexIn] *
   ↪  conv3_weights[weightIndex];
9        }
10     }
11     conv3_totalSum += conv3_bias[filterToGenerate];
12     conv3_featureMap[filterToGenerate][inputOffset] = conv3_totalSum;
13   }
14 }
```

The reference model uses the ReLU as the activation layer function [16]. It is a function used in many types of neural networks due to its simplicity of implementation.

Listing 3 presents the implementation of the FC layer. The external loop, starting at line 1, iterates according to the FC size (`FC1_OUTPUT_SIZE`=128 and `FC2_OUTPUT_SIZE`=5). The inner loop iterates through the input vector, performing MAC operations.

Listing 3. Fully Connected C implementation.

```
1 for (int outputIndex = 0; outputIndex < FC_OUTPUT_SIZE; outputIndex++){
2   totalValue = 0;
3   for (int i = 0; i < fc1_inputSize; i++)
4     totalValue += flatten1_vector[i] * fc1_weights[(fc1_inputSize*outputIndex)+i];
5   fc1_out_vector[outputIndex] = totalValue + fc1_bias[outputIndex];
6 }
```

Table II summarizes the CNN main parameters and the number of MAC operations per inference. It is necessary to execute 5,428,096 MAC operations per inference.

TABLE II
CNN main parameters MAC operations per inference.

	Input Size	Output Size	# of filters	kernel size	MAC operations
CONV 1	120	116	64	5	37,120
CONV 2	116	112	64	5	2,293,760
CONV 3	112	108	64	5	2,211,840
FC 1	6,192	128	128	–	884,736
FC 2	128	5	5	–	640
				Total:	**5,428,096**

B. Quantization Process for 32-bit Integer

The Pytorch framework provides quantization methods (https://pytorch.org/docs/stable/quantization.html). We evaluated the "Post-Training Dynamic Quantization" and "Quantization Aware Training for Static Quantization" methods, but they reduced the accuracy or required significant changes in

the model. *Thus, we decide to implement quantization in the C model, not in the Pytorch framework.*

Using the PyTorch results as the gold model, we first validated the *C model* using the FP representation. After validating this model, we used a Python script to convert the FP header files to integer values by multiplying the values by a constant. Although simple, the method requires defining the multiplication value M to avoid overflow in the convolution or FC layers. We evaluated different M values, observing that larger values, such as 10,000, reduce accuracy due to overflow in the FC layers. Thus, empirically, we define M as 1,000. This multiplier factor can be easily modified, as it is an input for the Python script.

Using a multiplication factor requires attention when connecting one layer's output to the next. Given that the weights are multiplied by the input data, both quantized by M, the outputs are multiplied by M^2. So, before starting a new layer, we must divide all results by M.

The *C model* is the same with FP and integer representation, with the same functions such as convolution, ReLU, and FC. However, two changes are needed in the code. The first is to change the data type from *float* to *int*. The second change is to include the header files with integers.

C. C Model Results

Table III presents the **model size** and **accuracy** for the CNN models, using the testing dataset (3,923 samples). The reference model showed a slight increase in accuracy (0.9%). This occurs because the testing dataset used by PyTorch (20%) and C models may differ. The accuracy values in PyTorch correspond to an average of several executions of the inference phase. The ***Ext1D*** FP model increased the accuracy by 13.6% ($88.5 - 74.9$) due to the reduction in wrong predictions, from 207 to 80. The ***Ext1D*** INT model slightly reduced accuracy, 1.3%, due to an increase in wrong predictions, from 80 to 90.

TABLE III
MODEL SIZE (NUMBER OF PARAMETERS) AND ACCURACY EVALUATING.

Model	**Model Size**	**Accuracy (%)**
Pytorch - Reference	271,621	74.0
Pytorch - Ext1D	926,981	**89.0**
Reference Model (C)	271,621	74.9
Ext1D FP (C)	926,981	88.5
Ext1D INT (C)	926,981	**87.2**

The C models were **profiled** using the OVP simulator [20], with a RISCV32IMF processor [21], by partitioning the instruction set into classes, instrumenting the OVP simulator through callbacks that evaluate each instruction fetched from memory. The compilation parameters use the OVP cross-compiler with O2 optimization. The RISCV32IMF processor has FP instructions that implement MAC operations, called "single precision fused multiply addition". These instructions reduce the number of instructions when executing the FP model. The instruction profiling enables to: (i) evaluate the number of memory accesses (LOAD/STORE), which implies energy consumption and processing time due to the memory access latency; (ii) quantify the number of executed arithmetic operations.

Table IV(a) presents the profiling for the reference, *Ext1D* FP and *Ex1D* INT models. This Table shows that:

1) Memory Access (`Load` and `Store`): increased by 3.7 times. An expected increase since the *Ext1D* model increases the model size (refer to Table III) by 3.4 times. The increase in model size also implies an increase in the total number of executed instructions by 3.7 times.
2) Arithmetic Operations: the number of multiplications (FP and INT) is equal to that obtained in Table II (5,428,096), showing that the C implementation did not perform unnecessary multiplications.
3) The FP implementation does not require division operations, while INT does, given the need for divisions in the quantization adjustments between the layers.
4) Integer addition/subtraction operations. In the INT model, we observe an increase of 5,507,305 operations. This is due to the MAC instructions in the FP instruction set, which reduces the number of executed instructions.

TABLE IV
C MODEL RESULTS.

(a) Instruction profiling of the CNN C models – RISC-V processor

Instruction Class	**FP Reference Model**	**Ext1D FP**	**Ext1D INT**
LOAD	**2.945.844**	**10.871.684**	**10.893.314**
STORE	12.179	41.946	63.668
IMM	90	53	46
ADDSUB	4.944.252	18.196.854	23.704.159
MULT	0	0	**5.428.096**
DIV	0	0	21.637
BRANCH	1.737.527	6.401.926	6.423.755
JUMP	242.376	887.496	887.496
FP_OP	**16.406**	**57.358**	0
FP_MUL	**1.470.336**	**5.428.096**	0
FP_DIV	0	0	0
FP_AddSub	**6.277**	**21.637**	0
Total nb of exec. inst.	**11.375.287**	**41.907.050**	**47.422.171**

(b) Execution time to execute the models (Apple M2 processor).

Total time (sec)	7.885	28.568	4.175
Time per inference (ms)	2.010	7.282	1.064

(c) Estimation of the energy consumption to execute the models (20pj INT - 131pJ FP).

Energy (pJ)	393	1,449	948

We evaluated the **execution time** on an Apple M2 processor, using a clang 14.0.3 compiler with O2 optimization. Table IV(b) presents the time to execute the 3,923 inferences and the time per inference (average result over five executions of each model). The time per inference increased by 3.6 times due to the increased model size (*Ext1D* FP). The *Ext1D* INT model is 6.84 times faster than the FP model, and despite the increase in model size, the *Ext1D* INT model is 1.9 times faster than the reference model. This result demonstrates the significant latency of FP operations. Despite accounting for instructions on the RISC-V processor and execution time on another processor, Ness et al. [22] shows that the latency is 2 to 8 times higher for FP instructions than for integer instructions, which is consistent with the results.

The **energy consumption** evaluation considers data from the literature. In [23], the average cost for executing integer operations is 20pJ@1.7GHz, at 22nm. BOOM-2w [24] and Shakti [25], both with a RISCV-IMAFD, the energy cost per FP operation is 133pJ and 122 pJ, respectively. From [22], it is possible to infer 137.6pJ for the FP operations for 22 nm@1.7GHz, a value close to that obtained in the literature.

Thus, we use 20pJ for integer instructions and 131pJ (average of 133, 120, 137.6) for FP instructions. Table IV(c) presents the estimated energy, using the instruction profiling. Consistently, the energy consumed by the FP model concerning the reference model increases by 3.7 times due to the model size. However, due to the lower energy cost of INT operations than the cost of FP operations, the energy consumed by the integer model is only 2.41 times higher.

The ***Ext1D INT*** successfully achieved its main goal: improve accuracy (12.3% better) without using FPUs. However, one challenge remains, the model size. The memory footprint is a critical parameter in embedded systems with limited resources.

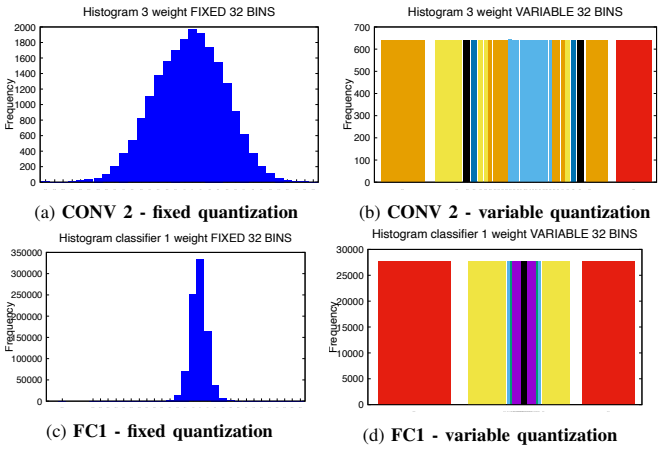
V. Memory Optimized C model

This section presents a Lookup Table (LUTs) quantization method to reduce the model size. In this method, the original values of the weights and biases are grouped into bins. An LUT with bins n contains the model values, replacing the original values with indexes pointing to a position in the LUT [26]. This work evaluated two LUT quantization methods:

- *Fixed quantization*. Employs bins of fixed size. To find the bin widths, the maximum and minimum values of the input values are calculated and then divided by the number of bins. Once the intervals are defined, each parameter receives the index of the interval into which it falls. Fixed bin sizes provide a consistent and easy-to-interpret data representation. However, it may not be optimal for datasets with varying densities or when important features are masked by bin size selection.
- *Variable quantization*. Seeks to have an equal number of elements in all bins. Equation $n\_elem = \frac{\#parameters}{nbins}$ gives the number of elements in each bin. To execute a variable quantization, the input parameters are sorted, inserting n_elem parameters into each bin. This approach offers flexibility in capturing distribution details and is useful for parameters with varying densities or outliers.

We adopt histograms to visualize the quantization methods. Figure 2 presents the weights' distributions for the 2nd convolutional and FC1 layers. We observe that the layers have distributions varying considerably. While the second convolutional layer presents a normal distribution, the FC1 layer presents most weights with a high percentage of values in the center of the histogram. Observe the histograms with variable bin widths. The width of the bin is inversely proportional to the number of values in the original distribution. The tails with fewer parameters present wider bins, whereas regions with more parameters have thinner bins. The most representative example of this behavior is observed in the weights of FC1, concentrating most parameters in a small area.

Table V displays the accuracy data, varying: (1) bin size, (2) data representation, and (3) the type of quantization. Note that the loss of accuracy is minimal with 16-bin LUTs. This is a surprising result considering that there is a layer with 884,736 parameters (FC1), being possible to represent it with just 16 values, with minimal loss in accuracy. Another unexpected data is the equivalence between fixed and variable quantization methods up to 16 bins. This result suggests that the number



(a) **CONV 2 - fixed quantization** (b) **CONV 2 - variable quantization**

(c) **FC1 - fixed quantization** (d) **FC1 - variable quantization**

Fig. 2. Quantization of the weights for the 2nd convolutional and FC1 layers. Y-axes: number of elements inside the bin. X-axes: value used in the bin (center of the interval).

TABLE V
Accuracy per bin size, varying the quantization method.

Accuracy	Float representation %		Integer representation %	
Nb. of bins	**Fixed**	**Variable**	**Fixed**	**Variable**
64	88.10	87.10	87.1	86.7
32	88.20	86.70	86.5	86.7
16	88.00	86.90	86.0	**87.1**
8	47.30	**85.80**	38.9	**85.5**
4	28.50	75.70	28.2	75.7
2	20.00	47.90	22.3	41.6

of bins equal to 16, for this CNN model, is the value to be adopted for the remainder of this work.

It is also worth noting the limits these quantization methods have. For fixed quantization in this CNN, 16 bins is the minimum value. However, with *variable quantization,* we can have an aggressive quantization, with only *eight bins*, with little loss of accuracy. Therefore, when adopting quantization methods, it is recommended to adopt variable quantization, especially if the designer adopts an aggressive strategy to reduce model size.

The implementation of the LUT-based quantization model adopts a memory "compression" method. The compressed method "packs" the quantized weight and bias values in one integer value. Using 16-bin LUTS, we need 4-bit indexes. Thus, each integer value (32 bits) in the C header files contains eight indexes, *reducing the model size by eight times.*

Table VI summarizes the results. The "compressed" model added complexity to the C model. The increased complexity of the code increased shift operations, logical operands such as `AND/OR` and `BRANCH` instructions. When comparing the number of instructions executed, the "compressed" model doubled the number of instructions compared to the *Ext1D Int* model. Despite the increase in the number of instructions, memory accesses (LOAD) increased only by 6%.

Note that besides the larger number of instructions, the compressed model is still faster than the float model. On the other hand, the estimated energy presents higher values among the evaluated models. An important future work involves

TABLE VI
SUMMARY OF THE RESULTS.

	Ref. 1D	Ext1D FP	Ext1D Int	**Compressed**
Accuracy %	74.9	88.5	87.2	87.1
Total inst. (10^6)	11.38	41.91	47.42	97. 51
LOAD instructions (10^6)	2.95	10.87	10.89	11.58
Time per inference (ms)	2.01	7.28	1.06	4.77
Energy (pJ)	393	1,449	948	1,950
Model size (#param)	271,621	926,981	926, 981	**115,953**

evaluating execution time and energy on a processor described at the RTL level, allowing accurate performance (number of clock cycles), power, and energy estimates. Another relevant future work is to use hardware acceleration, such as the extensions available in the RISC-V ISA.

The most significant result is presented in the last row of Table VI: the memory requirements to store the model (number of parameters). Since the index vectors are reduced by a factor of 8, a reduction in memory usage of this proportion is expected. According to the table, the reduction is 7.99, given the need to store the LUTs. Note that the compressed model requires less memory than the reference model, which has an accuracy of 74.9%, compared to 87.1% for the proposed model. Thus, we have achieved the second goal for resource-constrained devices: reduce the memory footprint for storing the CNN model.

VI. Conclusion and Future Work

The standard 1D CNN model used as a reference in this research lacks temporal awareness in its implementation, a crucial aspect for accurately detecting patterns in sensor data, especially for HAR. Despite its simplicity and modest hardware requirements in terms of computation, the model has trade-offs. It only achieves a 74% accuracy level in the testing dataset, making it an inefficient model for practical applications. To increase accuracy, we added temporal awareness to this model.

We deployed the reference CNN in a C model equivalent to the Pytorch model. The goal of the C model is to avoid FPUs and reduce the memory footprint. This integer-only model incurred a 1.3% accuracy loss compared to its FP counterpart. Next, we proposed the "Compressed Model" to effectively reduce the size of the model by packing the indices of 8 parameters (weight and bias) into a single integer word. This model reduced the size of the model almost eight times, keeping the accuracy of the integer model. However, this model increased the number of executed instructions, increasing execution time and energy consumption.

Summarizing, this work achieved its main goal, using a CNN model in resource-constrained embedded devices, without requiring FPUs or a large amount of memory.

This research paves the way for several future works. The following points highlight the key areas of interest for future research: (i) add pruning techniques; (ii) improve accuracy using a 2D model; (iii) evaluate hardware acceleration; (iv) evaluate the model in a processor described at the RTL level.

References

[1] M. Zeng, L. T. Nguyen, B. Yu, O. J. Mengshoel, J. Zhu, P. Wu, and J. Zhang, "Convolutional Neural Networks for human activity recognition using mobile sensors," in *MobiCASE*, 2014, pp. 197–205.

[2] H. Park *et al.*, "MultiCNN-FilterLSTM: Resource-efficient sensor-based human activity recognition in IoT applications," *Future Generation Computer Systems*, vol. 139, pp. 196–209, 2023.

[3] J. Janak, T. Tseng, A. Isaacs, and H. Schulzrinne, "An Analysis of Amazon Echo's Network Behavior," in *GLOBECOM*, 2021, pp. 1–6.

[4] R. S. Reusch, L. R. Juracy, and F. G. Moraes, "Assessment and Optimization of 1D CNN Model for Human Activity Recognition," in *SBESC*, 2022, pp. 1–7.

[5] S. Mekruksavanich and A. Jitpattanakul, "LSTM Networks Using Smartphone Data for Sensor-Based Human Activity Recognition in Smart Homes," *Sensors*, vol. 21, no. 5, pp. 1–25, 2021.

[6] T. Zebin, M. Sperrin, N. Peek, and A. J. Casson, "Human activity recognition from inertial sensor time-series using batch normalized deep LSTM recurrent networks," in *EMBC*, 2018, pp. 1–4.

[7] M. Shimoda, S. Sato, and H. Nakahara, "All binarized convolutional neural network and its implementation on an FPGA," in *FPT*, 2017, pp. 291–294.

[8] H. Ullah and A. Munir, "Human Activity Recognition Using Cascaded Dual Attention CNN and Bi-Directional GRU Framework," *Journal of Imaging*, vol. 9, no. 7, pp. 1–30, 2023.

[9] N. Dua, S. N. Singh, V. B. Semwal, and S. K. Challa, "Inception inspired CNN-GRU hybrid network for human activity recognition," *Multimedia Tools and Applications*, vol. 82, no. 4, pp. 5369–5403, 2023.

[10] F. J. Ordóñez and D. Roggen, "Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition," *Sensors*, vol. 16, no. 1, pp. 1–25, 2016.

[11] E. García, M. Villar, M. Fáñez, J. R. Villar, E. de la Cal, and S.-B. Cho, "Towards effective detection of elderly falls with CNN-LSTM neural networks," *Neurocomputing*, vol. 500, pp. 231–240, 2022.

[12] A. R. Inturi, V. Manikandan, and V. Garrapally, "A novel vision-based fall detection scheme using keypoints of human skeleton with long short-term memory network," *Arabian Journal for Science and Engineering*, vol. 48, no. 2, pp. 1143–1155, 2023.

[13] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference," *CoRR*, vol. abs/2103.13630, pp. 1–33, 2021.

[14] J. Xu, Z. He, and Y. Zhang, "CNN-LSTM Combined Network for IoT Enabled Fall Detection Applications," *Journal of Physics: Conference Series*, vol. 1267, pp. 1–6, 2019.

[15] J. Chiang, "Simple 1D CNN approach to human-activity-recognition (HAR) in PyTorch," 2022, source: https://github.com/jchiang2/Human-Activity-Recognition.

[16] Keras, "Layer activation functions," 2023, source: https://keras.io/api/layers/activations/, June 2023.

[17] G. Vavoulas, C. Chatzaki, T. Malliotakis, M. Pediaditis, and M. Tsiknakis, "The MobiAct Dataset: Recognition of Activities of Daily Living using Smartphones," in *ICT4AWE*, 2016, pp. 143–151.

[18] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *CoRR*, vol. abs/1912.01703, pp. 1–12, 2019. [Online]. Available: http://arxiv.org/abs/1912.01703

[19] Y. Yu, T. Zhi, X. Zhou, S. Liu, Y. Chen, and S. Cheng, "BSHIFT: A Low Cost Deep Neural Networks Accelerat," *International Journal of Parallel Programming*, vol. 47, pp. 360–372, 2019.

[20] S. L. Imperas, "Open Virtual Platforms the source of Fast Processor Models & Platforms," 2023, source: http://www.ovpworld.org/.

[21] R.-V. Foundation, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213," 2019, source: https://riscv.org/technical/specifications.

[22] T. V. Ness, "Low Power Floating-Point Unit for RISC-V," Master's thesis, Norwegian University of Science and Technology, 2018.

[23] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Transactions Very Large Scale Integration Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[24] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," EECS Department, University of California, Berkeley, Tech. Rep., 2015.

[25] Shakti, "Open Source Processor Development Ecosystem," 2023, source: https://shakti.org.in.

[26] L. Wang *et al.*, "Learnable Lookup Table for Neural Network Quantization," in *CVPR*, 2022, pp. 12 423–12 433.