Dependability analysis of neural networks implemented in Arduino

Joaquín Gracia-Morán[1][0000−0001−9715−8960], Juan-Carlos Ruiz[1][0000−0001−7678−3513], David de Andrés[1][0000−0002−4744−3795], Luis-J. Saiz-Adalid[1][0000−0002−4868−2050], J. Carlos Baraza-Calvo[1][0000−0001−7692−2309], Daniel Gil-Tomás[1][0000−0001−9225−1998], and Pedro J. Gil-Vicente[1][0000−0002−9364−7385]

ITACA, Universitat Politècnica de València
Camino de Vera s/n, 46022 - Valencia (Spain)
{jgracia, jcruizg, ddandres, ljsaiz, jcbaraza, dgil, pgil}@itaca.upv.es
https://gstf.blogs.upv.es/

Abstract. The use of neural networks has expanded to environments as diverse as medical systems, industrial devices and space systems. In these cases, it is essential to balance performance, power consumption, and silicon area. Furthermore, in critical environments, it is necessary to ensure high fault tolerance.
Traditionally, the parameters of neural networks have been codified using 32-bit floating-point numbers, which entails high memory consumption and greater vulnerability to failures due to the aggressive scaling of CMOS technology. An effective strategy for optimizing these systems is to reduce parameter precision, using fewer bits and thus, reducing both the amount of memory required and the processing time.
However, several questions arise when implementing these types of networks in embedded systems: Do they maintain their reliability in critical environments, or do they require fault tolerance mechanisms? Are area and latency really reduced?
This work addresses these questions by reducing the precision of a neural network and implementing it in an Arduino-based system. In addition, Error Correction Codes have been incorporated and, using the fault injection technique, their reliability has been evaluated by comparing the same neural network with parameters encoded in 8, 16 and 32 bits.

Keywords: Error Correcting Codes · Dependability · Single Faults · Multiple Faults · Optimized Neural Networks · Embedded Systems

1 Introduction

Nowadays, neural networks have expanded to such an extent that they are now commonly found in fields as diverse as the automotive industry, aerospace applications, and consumer electronics [3]. Neural networks process a large amount of

C.Fernandez-Llatas and A. Martinez-Millana (Ed.):
Workshop on Innovation on
Information and Communication Technologies
(ITACA-WIICT 2025)

ISBN 978-84-09-77296-4

information through interconnected nodes (neurons) organized in multiple layers [39], basing their operation on a series of parameters (weights, biases, etc.).

The operation of a neural network is divided into two phases. In the first one, known as training phase, the aforementioned parameters are calculated. These values will determine the network's behaviour during the second phase, known as inference phase. In this second stage, predictions are made based on new input data. Thus, the integrity of the parameters calculated in the training phase is essential for the proper functioning of the neural network, as an erroneous value can seriously compromise the accuracy of the predictions.

During the initial training stage, the different values of the neural network (weights, biases, etc.) are represented in the IEEE 754 single-precision format [1], also known as FP32. To reduce the execution time during the inference stage, weights are usually stored in memory. However, the increasing integration scale of CMOS technology has increased the rate of multiple faults in memory devices [6], which can negatively affect the neural network inference process [21][24]. For example, corruption of just 13 bits in the CNN ResNet18, within a set of approximately 93 million bits, has been shown to cause a complete collapse of the network [29]. This vulnerability has caused a growing interest in researching the reliability of neural networks [34][28]. Results obtained in these works show that it is essential to incorporate fault tolerance mechanisms when implementing neural networks in critical applications [4][33][31].

To reduce memory footprint, a current trend is the compression of the neural network models, generating optimized models [22]. This reduction in memory usage decreases the likelihood of fault occurrences, thereby increasing the network's reliability [11].

While model compression is effective and optimized neural networks maintain an adequate level of accuracy for many applications, several questions arise when attempting to use these models in embedded systems operating in critical environments: Will the optimized network be sufficiently reliable, or will it be necessary to implement additional fault-tolerance mechanisms? What is the improvement in area consumption and latency of the optimized network?

In this paper, we have developed a simple neural network with parameters codified in FP32. Then, we have optimized it, by quantizing their parameters with different data word sizes. Next, we have implemented all neural network's models into an Arduino-based system, and we have studied the reliability of these neural networks, both unprotected and protected with different Error Correcting Codes, comparing their results.

This work is organized as follows. Section 2 briefly describes the neural network model compression processes and the application of error-correcting codes to them. Section 3 summarizes the embedded system used, the implemented neural network, and the optimization methods. Section 4 presents the experimental evaluation of the system, analyzing the impact of fault injection before and after applying different types of Error Correcting Codes. Finally, Section 5 presents the conclusions obtained from the results achieved.

2 Compression and Fault Tolerance in Neural Netowrks

2.1 Compression of neural networks

As discussed in the previous section, neural network parameters are usually represented in the IEEE 754 single-precision (FP32) format. However, using parameters in this format results in high memory usage and an increase in inference time. In this way, optimising both factors has become increasingly important. Different strategies can be used to compact neural networks [7][23]. At the hardware level, this optimization can be achieved through specialized devices, such as neural network accelerators, which optimize system architecture to improve parallelism and minimize memory accesses [25][20].

At software level, it is possible to reduce network complexity by designing smaller models with acceptable accuracy [37], or by decomposing the original parameters into multiple smaller matrices or tensors, focusing on the reduction of both, the memory footprint and the number of required operations [27].

Parameter pruning and optimization processes are widely used techniques for compressing and accelerating neural network models [5]. Both techniques try to eliminate redundant parameters that do not significantly affect network performance. They are typically applied in convolutional and fully connected layers. For example, the Brain floating-point format, also known as BF16 [19], is supported by various accelerators, such as NVIDIA Ampere GPUs [10] or Intel's Nervana accelerators [16]. These devices allow for inferences using BF16 arithmetic. Although other floating-point formats exist, their use is limited by a lack of hardware support, so they will not be included in this work.

Also, quantization allows weights, represented in FP32, to be converted into 8-bit integers, also known as INT8 [19]. This transformation reduces the size of parameters in memory and simplifies arithmetic operations by decreasing numerical precision.

While approximating FP32 values to formats with fewer bits generates rounding errors that, in some cases, can affect the accuracy of certain neural networks, those based on BF16 or INT8 offer the same accuracy as their FP32-based counterparts [8].

2.2 Protecting neural networks with error correcting codes

In general, the smaller the weights of a neural network, the greater its resilience [35][30]. Even so, fault tolerance mechanisms must be added if the neural network is to be used in a critical environment. In this section, we will focus on the protection of neural network parameters using Error Correcting Codes (ECC).

Many works have studied this type of neural network protection, among which we can highlight the following ones, as they propose ECCs similar to those used in this work. For example, [18] combines single error correction (SEC) and triple repetition codes. In this case, it is a specific solution for a specific CNN, rather than a generic methodology applicable to any network.

On the other hand, in [26], and depending on the bit values (30 : 28) of the neural network weights, various types of Hamming SEC are used to protect specific bits. The approach presented in [15] is applied to optimized neural networks. In this case, a training scheme is proposed that obtains at least seven non-significant bits within eight consecutive weights (8 bytes) to implement a SEC-DED (Double Error Detection) code.

A final approach is presented in [12], where a series of ECCs with different fault-tolerance properties have been implemented to protect the weights of several neural networks with parameters in BF16 format.

3 System Description

This section summarizes the main characteristics of both the embedded system and the neural network used in this work, as well as the optimization methods.

3.1 Embedded system and neural network

The embedded system has been used in a mobile robot based on an Arduino UNO R3 board. This system incorporates an ATmega328p microcontroller, 32 KB of FLASH memory, 2 KB of SRAM, 1 KB of EEPROM, and operates at 16 MHz. The robot has two DC motors for the rear wheels, three infrared proximity sensors (one E18-D80NK in the center and two KY-032 sensors on the sides), and a servomotor for steering, in addition to the necessary hardware to power the system, as shown in Fig. 1.

(a) Side view. (b) Top view.

Fig. 1. Arduino-based system.

To drive the mobile robot, we have generated and trained a small neural network with the following characteristics (see Fig. 2):

- 3-layer neural network: input, intermediate, and output.
- Input layer: receives readings from the three proximity sensors (left, right, and center).

– Intermediate layer with 5 neurons.
– Output layer: generates outputs corresponding to speed, direction of rotation, and forward direction.
– Two sets of weights, one between the input layer and the intermediate layer (hereafter referred to as Stage 1 weights), and another one between the intermediate layer and the output layer (hereafter referred to as Stage 2 weights).

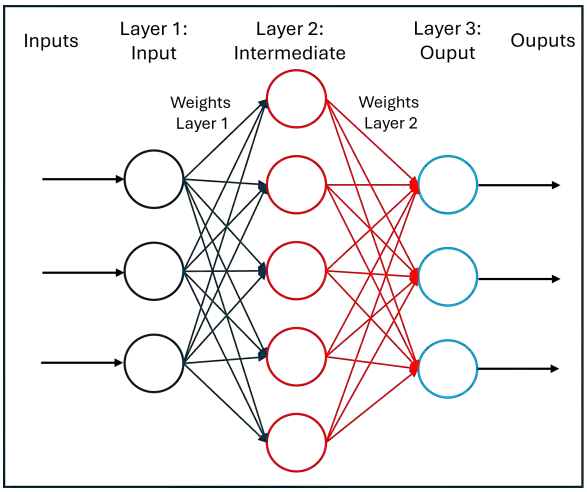

Fig. 2. Neural network scheme.

Once the weights were generated, inference was implemented in the mobile robot, such that the appropriate outputs were generated based on the proximity sensor values. Weights are originally encoded in the FP32 format. This format uses 32 bits distributed as follows (see Fig. 3): bit 31 stores the sign bit (S), bits 30 through 23 contain the exponent (E) in excess-127 format, and bits 22 through 0 represent the magnitude, with one implicit bit. Thus, the corresponding decimal value can be calculated according to Equation 1.

Fig. 3. FP32 format.

$$value = (-1)^S \times 2^{E-127} \times \left(1,0 + \sum_{i=1}^{23} b_{23-i} \times 2^{-i}\right) \quad (1)$$

3.2 Reduction of the neural network parameters' size

Despite the simplicity of the neural network used, its design allows us to evaluate the network's behavior in a real-world environment, providing a solid basis for analyzing the system's resilience to memory faults, and verifying both the feasibility of optimizing its parameters and its protection with ECC. In this way, we have carried out two types of parameter's size reduction, which we will describe next.

Summary of the BF16 format. We have converted the original FP32 parameters into the BF16 format just truncating the lowest 16 bits, as shown in Fig. 4. Thus, the optimized network weights are stored in memory in 16-bit variables. Inference is executed in real time, adjusting the robot outputs according to the sensor signals.

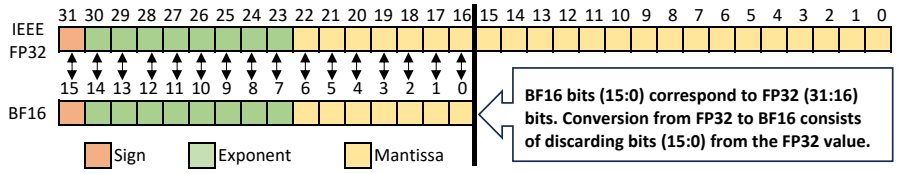

Fig. 4. FP32 vs BF16 format [32].

Although the weights in BF16 are stored in memory in 16-bit variables, the various operations related to the neural network are performed in 32-bit floating-point variables. To do this, we define a 32-bit floating-point variable, in which the weight in BF16 is stored in the 16 most significant bits, and the 16 least significant bits are set to '0'. This way, the same Arduino library can be used in both neural networks (FP32 and BF16) to perform the necessary floating-point operations, without having to implement a new library.

Summary of the quantization process: from FP32 to INT8. This section will detail the post-training quantization process carried out based on the general procedure presented in [17].

As already mentioned, quantization process reduces the computational and energy resources of a neural network. In the case of this work, this process consisted on converting the network's weights and biases, initially represented in 32-bit floating-point (FP32), into 8-bit integers (INT8).

To quantize the network, the first step is to quantize its weights and inputs. Each set of weights and the set of inputs are processed independently, since their quantized values depend only on their range of values, once the network has been trained, and on the range of values defined for quantization. Thus, each data set is traversed to locate its maximum and minimum values. Each data set is then iterated through again to adjust its values to the 8-bit integer range $[-128, 127]$ according to Equation 2.

$$\begin{aligned} scale &= \frac{(max - min)}{255} \\ zero &= -round(\frac{min}{scale}) - 128 \\ int8 &= round(\frac{float}{scale}) + zero \end{aligned} \tag{2}$$

scale allows to divide the range of real values into uniform partitions, while *zero* defines the integer value associated with the real value 0.0. Values greater than 127 or lower than -128 are truncated to those values.

Once this data have been quantized, it is necessary to determine how it affects the processing performed in each of the network's layers and what are the *scale* and *zero* values for these layers. This operation must be performed sequentially, following the propagation of information from the system's input to its output. Thus, in this case, the first stage to consider would be the intermediate layer in Fig. 2, which corresponds to a dense layer with a hyperbolic tangent as the activation function.

To do this, the dense layer is first applied to the input data set, but using the real values resulting from dequantizing the previously quantized inputs and weights, according to Equation 3.

$$float = scale \times (int8 - cero) \tag{3}$$

This allows us to monitor the operation performed by the dense layer, obtain the maximum and minimum values generated when processing all the input data, and thereby determine its *scale* and *zero*. From these values, using Equation 2, we obtain the quantized values provided by this layer when processing the quantized data.

Afterwards, it is necessary to determine the effect of quantization on the activation function. Since the hyperbolic tangent is a very expensive operation to perform, the simplest solution to fulfill it using integer arithmetic is to precalculate a lookup table for all possible values this function can have for the range of integers considered, $[-128, 127]$ in this case. As done before, it is necessary to consider the effect that the quantization of the dense layer from which the data processed by this activation function comes from. Therefore, the lookup table is calculated using Equation 4, from the dequantization of the output data of the dense layer (Equation 3).

$$
\begin{aligned}
scale &= \frac{(1.0 - (-1.0))}{255} = \frac{2.0}{255} \\
zero &= -round(\frac{-1.0}{scale}) - 128 = 0
\end{aligned}
\tag{4}
$$

$$
\begin{aligned}
&\forall\ value \in [-128, 127]\ : \\
&\quad float = scale_{Dense} \times (value - zero_{Dense}) \\
&\quad int8[value + 128] = round(\frac{tanh(float)}{scale}) + zero
\end{aligned}
$$

Thus, from this lookup table, the output values of the dense layer's activation function can be obtained after processing all the quantized values.

This same procedure must be applied successively to all the layers defined in the network. In our case, only one more dense layer with a hyperbolic tangent as the activation function remains.

Once the network is quantized, inference operations are performed using integer arithmetic, which simplifies calculations and significantly reduces latency. The convolutional and dense layers, which are the most computationally intensive, transform multiplication and accumulation operations into integer operations.

In fully connected layers, the output is calculated as the weighted sum of the inputs multiplied by the weights, plus a bias (translated as bias, shift, or threshold). When quantized, these values are scaled with a zero point, allowing floating-point operations to be transformed into integer operations with bit shifts and rounding. This transformation not only simplifies the required hardware but also allows operations to be parallelized, increasing efficiency. However, when implementing the quantized network in Arduino, we had an additional problem: to obtain maximum precision, the quantized network internally uses not only 8-bit integers, but also requires integers up to 64 bits. Specifically, there is a product of 32-bit integer values, which generates a 64-bit result. This is not supported by Arduino, since it only implements integers up to 32 bits and the arithmetic supported by them. Since the 64-bit result is subsequently shifted 32 places to the right and rounded, it was possible to solve this by using only 32-bit integers, alternating partial sums and one-bit shifts, instead of performing the entire shift at the end.

Integer arithmetic inference has the advantage of running faster and consuming less power, but it introduces small variations in the results due to the reduction in numerical precision. However, these variations are usually tolerable in many practical cases, especially when looking for lightweight and efficient systems.

4 Experimental evaluation

4.1 Results of fault injection experiments in the unprotected system

In order to evaluate the performance of the neural network when memory errors affect its weights, we have carried out different fault injection experiments [2] in the embedded system. To do this, single and multiple adjacent bit-flip faults were injected into all bits of all weights, analyzing whether these errors caused changes in one or more outputs. As can be seen in Table 1, even a single erroneous bit can cause changes in the outputs. Furthermore, the percentage of changed outputs increases as more faults are injected, an expected result.

Table 1. Percentage of outputs changed

Adjacent Faults	Layer 1	Layer 2	No Variation
FP32			
1	3,78	2,42	93,81
2	3,82	2,43	93,75
3	3,62	2,46	93,92
4	3,75	2,58	93,68
5	3,88	2,68	93,44
6	4,01	2,77	93,22
7	4,24	2,86	92,9
8	4,40	2,98	92,62
BF16			
1	6,41	3,81	89,78
2	6,51	3,83	89,67
3	5,77	3,85	90,38
4	5,80	3,87	90,33
5	5,84	3,9	90,26
6	5,89	3,93	90,18
7	6,15	3,97	89,88
8	6,25	4,02	89,73
INT8			
1	2,08	0,48	97,43
2	2,38	0,47	97,15
3	2,68	0,6	96,73
4	2,92	0,65	96,43
5	3,13	0,67	96,21
6	3,27	0,69	96,03
7	3,72	0,74	95,54
8	3,87	0,89	95,24

We can also see that the network with the INT8 neural network presents the lowest percentage of changed outputs, a common behavior in this type of

model [38]. As quantization limits the range of representable values, even in the presence of errors in the parameters, their value remains within the expected range, which limits the effect of the error. On the contrary, errors in the FP32 format can change the represented value by many orders of magnitude, with a devastating effect on the inference process.

On the other hand, BF16 network shows the highest percentage of changed outputs. This is because the BF16 format eliminates the superfluous bits that could be found in the FP32 format network, and whose modification does not affect the network's behavior. In other words, the BF16 format consists of fewer, but more significant, bits, so the variations are greater than in FP32 in proportion to the number of injections.

It can also be seen that the weights in Layer 1 are more likely to induce changes in the outputs than those in Layer 2. This is due to the complete connection between neurons in consecutive stages: an error in the intermediate stage propagates to all neurons in the output stage, while an error in the final stage only affects one neuron, which reduces the damage that can be caused.

Table 2 shows the percentage of failures that occurred with respect to the changed outputs, considering a variation in the output value greater than 5% as a failure. We can observe that the optimised neural networks (BF16 and INT8 networks) have a higher failure percentage than the FP32 network. As mentioned before, in the BF16 optimized network, all superfluous bits have been removed, leaving only the truly significant bits. On the other hand, although there is a lower percentage of changed outputs in the INT8 network (as seen in Table 1), these variations mainly provoke a change in outputs greater than 5%.

Table 2. Percentage of failures

	FP32		**BF16**		**INT8**	
Adjacent faults	**Layer 1**	**Layer 2**	**Layer 1**	**Layer 2**	**Layer 1**	**Layer 2**
1	12,56	3,91	15,79	5,21	75,00	66,07
2	11,56	5,33	15,03	7,41	71,43	70,69
3	4,93	4,89	7,14	7,14	72,22	74,07
4	5,21	4,85	8,12	7,69	73,47	67,31
5	4,93	4,86	8,33	8,33	71,43	68,89
6	4,95	4,57	9,09	8,59	78,79	70,59
7	6,49	4,07	12,90	8,33	84,00	69,57
8	6,76	3,52	14,88	8,02	92,31	76,92

It should be noted that, in more realistic environments, an optimized network is more robust, as the probability of errors decreases as fewer bits are required to store the different parameters. This can be verified theoretically with Equation 5, which shows the overall failure rate, which can be seen in Fig. 5 (considering λ equal to the failure rate -which in our case is 1 since we have performed an exhaustive fault injection in all bits of all weights-, n_cells the number of cells

occupied by the neural network weights relative to the overall number of cells, and $n\_failure$ the number of failures). In this figure, we can confirm that the optimized networks are more robust, with a higher failure rate of the FP32 network, followed by the BF16 failure rate.

$$Failure\_rate = \lambda \times n\_cells \times n\_failure \tag{5}$$

Fig. 5. FP32 vs BF16 format [32].

4.2 Error correcting codes used

As we have just checked, a variation in a single bit can cause changes in the outputs of the neural network. If we want to use it in a critical environment, the weights of the neural network must be protected. We have used a series of Error Correcting Codes (ECCs) to protect these weights.

- *Single Error Correction-Double Adjacent Error Correction (SEC-DAEC).* We have used different versions of this ECC, adapted to the word length of each data (FP32, BF16 and INT8). They are based on Hamming SEC codes and the interleaving technique [36].
- *Asymmetric ECC.* This type of ECC is able to protect some bits more than others. The idea behind applying Asymmetric ECC is to sacrifice some robustness of the neural network in exchange for improving both the size and

latency of the program, protecting some bits more than others [13]. As in the previous ECC, different versions of this ECC, adapted to the word length of each data (FP32, BF16 and INT8) have been used. Specifically, we have used:

  - UEP (40, 32) for the FP32 network. This ECC can correct single and double adjacent errors in the 16 most significant bits (the protected area); and single errors in both the protected area and the parity bits.
  - UEP (22, 16) for the BF16 network. This ECC protects the 10 highest-order bits against single or double adjacent errors. That is, if a single or double adjacent error occurs, regardless of the position, the 10 highest-order bits are guaranteed to be correct. If the error affects those bits, they are corrected. If the error affects unprotected bits, the decoding process is guaranteed to not affect the protected bits, ensuring their integrity.
  - UEP (12, 8) for the INT8 network. As in the previous case, this asymmetric ECC guarantees that the four highest-order bits will be protected against single or double adjacent errors regardless of the affected bits of the codeword.

– *FUEC-DAEC (23, 16) [14].* This ECC is capable of correcting single and double adjacent errors in 16-bit words with only 7 parity bits. Furthermore, it can also detect bursts of 3- or 4-bit errors. It has been used in the FP32 and BF16 networks.

4.3 Results of fault injection experiments in the protected system

As we have already seen, errors in the neural network weights cause failures in the outputs. To reduce these effects, we have added the ECCs just described to the different networks. Table 3 shows the failure rate of the different neural networks protected by the ECCs. With respect to the unprotected networks, the failure rate has decreased in all cases, an expected result. Specially significant are the results of single and double adjacent errors, as failure rate is 0 in almost all cases, according to the fault tolerance of the corresponding ECC. UEP ECC for the INT8 is the unique case where the failure rate is not 0 for single and double adjacent errors. This is the normal behaviour of the UEP ECC. In the cases of FP32 and BF16, the intrinsic redundancy of these networks causes a failure rate equal to 0 when using this type of ECC.

4.4 Analysis of the introduced overhead

The incorporation of fault-tolerance mechanisms in the system generates an overhead in both the software size and its latency. Table 4 presents the results obtained, where the values in parentheses indicate the percentage of memory used. It should be noted that, in the implementation of the different Error Correcting Codes, a literal translation of the design into hardware has not been made. Instead, software structures have been used that minimize the overhead introduced by the ECCs.

Table 3. Failure rate

FP32			
Adjacent Faults	**FUEC-DAEC**	**SEC-DAEC**	**UEP**
1	0,00	0,00	0,00
2	0,00	0,00	0,00
3	14,77	15,59	19,69
4	15,18	15,18	37,32
5	15,18	34,45	40,61
6	14,36	15,18	58,65
7	11,89	18,87	20,51
8	15,59	16,41	53,73
BF16			
Adjacent Faults	**FUEC-DAEC**	**SEC-DAEC**	**UEP**
1	0,00	0,00	0,00
2	0,00	0,00	0,00
3	6,56	6,56	6,56
4	14,15	14,15	14,15
5	10,87	10,87	10,87
6	7,79	7,79	7,79
7	9,02	9,02	9,02
8	8,82	8,82	8,82
INT8			
Adjacent Faults	**SEC-DAEC V1**	**SEC-DAEC V2**	**UEP**
1	0,00	0,00	2,67
2	0,00	0,00	2,46
3	8,20	7,69	8,82
4	7,49	7,49	9,02
5	6,15	6,36	6,87
6	4,61	5,13	5,02
7	2,87	2,87	3,79
8	2,26	1,13	2,26

Table 4. System overhead

FP32			
	Sketch	**Global variables**	**Execution time (us)**
Unprotected	5556 (17%)	265 (12%)	2576
FUEC-DAEC	8716 (27%)	341 (16%)	14122
SEC-DAEC	7870 (24%)	341 (16%)	13267
UEP	7908 (24%)	307 (14%)	14084
BF16			
	Sketch	**Global variables**	**Execution time (us)**
Unprotected	5452 (16%)	179 (8%)	2601
FUEC-DAEC	6592 (20%)	221 (10%)	6592
SEC-DAEC	6192 (19%)	221 (10%)	3717
UEP	6152 (19%)	221 (10%)	3596
INT8			
	Sketch	**Global variables**	**Execution time (us)**
Unprotected	5740 (17%)	658 (32%)	2916
SEC-DAEC V1	6236 (19%)	700 (34%)	3649
SEC-DAEC V2	6186 (196%)	700 (34%)	3666
UEP	6136 (19%)	700 (34%)	3552

Software size analysis. As can be seen in Table 4, memory consumption decreases significantly with the size of the parameters. If we also take into account that the ECCs used are also more compact, the final result is a clear decrease in the amount of memory used, even though the neural network is very small.

Regarding the ECCs themselves, the differences in memory consumption are minimal. One of the reasons is that all ECCs have similar error correction capabilities (they all correct single and double adjacent errors), which means that the formulas used for error correction are similar. It can be observed that the fewer the parity bits, the lower the memory consumption. In this sense, the UEP ECC requires the least memory, as it corrects the fewest errors (single and double adjacent errors in the protected area).

Regarding the FUEC-DAEC ECC, it occupies the largest memory space since it has the highest fault tolerance capabilities (it is capable of correcting single and double adjacent faults, and detecting bursts of 3 and 4 erroneous bits). It should be noted that in the experiments carried out with the FUEC-DAEC ECC, the 3- or 4-bit burst error detection offered by this ECC was not used, since we were initially only interested in error correction. However, considering the memory occupancy data and the time overhead associated with this ECC (analyzed in the next section), it could be beneficial to take advantage of this fault tolerance capability. For example, when detecting these types of errors, the calculations performed by the neural network could be repeated, that is, it is possible to implement temporal redundancy [9]. However, it is important to keep

in mind that implementing this additional functionality would imply an increase in both memory usage and program latency.

Runtime analysis. Regarding the execution time, given that the processor commands require approximately 750 ms to generate the desired effects on the mobile robot's mechanical components, there is sufficient time to perform the verification of the various ECCs without affecting the normal operation of the robot.

One notable finding is the decrease in execution time for the protected versions in the optimized neural networks. This significant reduction is due to the fact that the ECC error checking operations are performed with 8- or 16-bit data, stored as unsigned integers. This makes the operations performed on the Arduino more efficient, which directly affects execution time. Regarding the unprotected version, it can be seen that the execution time is similar in all versions of the neural network.

5 Conclusions and future work

In this work, we have analyzed the integration of different Error Correcting Codes (ECC) into a neural network implemented in Arduino with parameters in different formats (FP32, BF16 and INT8). Our main objective has been to protect the neural network weights from memory faults and to evaluate whether the generated overhead is manageable in terms of software size and execution time.

Although the neural network used is small, it is useful as a test case to study this type of protection against memory faults. To evaluate the system's behavior, we have exhaustively run several fault injection campaigns on the network's weights, both when protected and unprotected. In any case, we have observed that the inclusion of ECCs does not entail excessive overhead, making its use viable in this context. We have also seen a significant improvement in memory consumption and execution time when using reduced size parameters.

As future work, we plan to continue exploring new ECC and other fault tolerance mechanisms to evaluate their applicability in this type of embedded systems.

Acknowledgments. This work has been funded by the Spanish Government (DEFADAS project, Grant PID2020-120271RB-I00, MCIN/AEI/10.13039/501100011033) and by Universitat Politècnica de València (Convocatoria A+D, Proyectos de Innovación y Mejora Educativa, PIME/24-25/435).

References

1. IEEE standard for floating-point arithmetic. IEEE Std 754-2008 (2008)

2. Benso, A., Prinetto, P.: Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation. Springer Publishing Company, Incorporated, 1st edition (2010)
3. Brando, A., Serra, I., Mezzetti, E., Cazorla, F.J., Perez-Cerrolaza, J., Abella, J.: On neural networks redundancy and diversity for their use in safety-critical systems. Computer **56**(5), 41–50 (2023). https://doi.org/10.1109/MC.2023.3236523
4. Burel, S., Evans, A., Anghel, L.: Zero-overhead protection for cnn weights. In: 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). pp. 1–6 (2021). https://doi.org/10.1109/DFT52944.2021.9568363
5. Cheng, Y., Wang, D., Zhou, P., Zhang, T.: Model compression and acceleration for deep neural networks: The principles, progress, and challenges. IEEE Signal Processing Magazine **35**(1), 126–136 (2018). https://doi.org/10.1109/MSP.2017.2765695
6. Coelho, B.L., Dos Santos, F.F., Saveriano, M., Allen, G., Daniel, A., Guertin, S., Vartanian, S., Wyrwas, E., Frost, C., Rech, P.: Impact of radiation-induced effects on embedded gpus executing large machine learning models. IEEE Transactions on Nuclear Science, In Press pp. 1–1 (2025)
7. Deng, L., Li, G., Han, S., Shi, L., Xie, Y.: Model compression and hardware acceleration for neural networks: A comprehensive survey. Proceedings of the IEEE **108**(4), 485–532 (2020). https://doi.org/10.1109/JPROC.2020.2976475
8. Dhiraj Kalamkar, e.a.: A study of bfloat16 for deep learning training (2019), https://arxiv.org/abs/1905.12322, arXiv:1905.12322
9. Dubrova, E.: Fault-tolerant design. Springer-Verlag New York (2013)
10. Feng, B., Wang, Y., Geng, T., Li, A., Ding, Y.: Apnn-tc: accelerating arbitrary precision neural networks on ampere gpu tensor cores. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3458817.3476157
11. Goldstein, B.F., Srinivasan, S., Das, D., Banerjee, K., Santiago, L., Ferreira, V.C., Nery, A.S., Kundu, S., França, F.M.G.: Reliability evaluation of compressed deep learning models. In: 2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS). pp. 1–5 (2020). https://doi.org/10.1109/LASCAS45839.2020.9069026
12. Gracia-Moran, J., Ruiz, J.C., de Andres, D., Saiz-Adalid, L.J.: Allocating ecc parity bits into bf16-encoded cnn parameters: A practical experience report. In: Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing. p. 75–80. LADC '24, Association for Computing Machinery, New York, NY, USA (2024), https://doi.org/10.1145/3697090.3697092
13. Gracia-Morán, J., Ruiz-García, J.C., Saiz-Adalid, L.J.: Uso de códigos de corrección de errores asimétricos en un sistema empotrado. In: VII Jornadas de Computación Empotrada y Reconfigurable (JCER2023), Jornadas SARTECO,. pp. 699–706 (2023)
14. Gracia-Morán, J., Saiz-Adalid, L.J., Gil-Tomás, D., Gil-Vicente, P.J.: Improving error correction codes for multiple-cell upsets in space applications. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **26**(10), 2132–2142 (2018). https://doi.org/10.1109/TVLSI.2018.2837220
15. Guan, H., Ning, L., Lin, Z., Shen, X., Zhou, H., Lim, S.H.: In-place zero-space memory protection for CNN. Curran Associates Inc., Red Hook, NY, USA (2019)

16. Hickmann, B., Chen, J., Rotzin, M., Yang, A., Urbanski, M., Avancha, S.: Intel Nervana Neural Network Processor-T (NNP-T) Fused Floating Point Many-term Dot Product. In: 2020 IEEE 27th Symposium on Computer Arithmetic (ARITH). pp. 133–136 (2020). https://doi.org/10.1109/ARITH48897.2020.00029
17. Jacob, B., et al.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. Tech. rep. (2017), https://arxiv.org/abs/1712.05877
18. Jang, M., Hong, J.: Mate: Memory-and retraining-free error correction for convolutional neural network weights. Journal of Information and Communication Convergence Engineering **19**(1), 22–28 (Mar 2021). https://doi.org/10.6109/jicce.2021.19.1.22
19. JunKyu Lee, e.a.: Resource-efficient convolutional networks: A survey on model-, arithmetic-, and implementation-level techniques. ACM Comput. Surv. **55**(13s) (Jul 2023). https://doi.org/10.1145/3587095
20. Lee, J., Kim, C., Kang, S., Shin, D., Kim, S., Yoo, H.J.: Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision. IEEE Journal of Solid-State Circuits **54**(1), 173–185 (2019). https://doi.org/10.1109/JSSC.2018.2865489
21. Li, G., Hari, S.K.S., Sullivan, M., Tsai, T., Pattabiraman, K., Emer, J., Keckler, S.W.: Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In: SC17: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2017)
22. Li, Z., Liu, F., Yang, W., Peng, S., Zhou, J.: A survey of convolutional neural networks: Analysis, applications, and prospects. IEEE Transactions on Neural Networks and Learning Systems **33**(12), 6999–7019 (2022). https://doi.org/10.1109/TNNLS.2021.3084827
23. Marinó, G.C., Petrini, A., Malchiodi, D., Frasca, M.: Deep neural networks compression: A comparative survey and choice recommendations. Neurocomputing **520**, 152–170 (2023). https://doi.org/https://doi.org/10.1016/j.neucom.2022.11.072, https://www.sciencedirect.com/science/article/pii/S0925231222014643
24. Mittal, S.: A survey on modeling and improving reliability of dnn algorithms and accelerators. Journal of Systems Architecture **104**, 101689 (2020). https://doi.org/https://doi.org/10.1016/j.sysarc.2019.101689, https://www.sciencedirect.com/science/article/pii/S1383762119304965
25. N. P. Jouppi, e.a.: In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture. p. 1–12. ISCA '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3079856.3080246
26. Nguyen, D.T., Ho, N.M., Chang, I.J.: St-DRC: Stretchable dram refresh controller with no parity-overhead error correction scheme for energy-efficient DNNs. In: 2019 56th ACM/IEEE Design Automation Conference (DAC). pp. 1–6 (2019)
27. Novikov, A., Podoprikhin, D., Osokin, A., Vetrov, D.: Tensorizing neural networks. In: Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1. p. 442–450. NIPS'15, MIT Press, Cambridge, MA, USA (2015)
28. Qutub, S., Geissler, F., Peng, Y., Gräfe, R., Paulitsch, M., Hinz, G., Knoll, A.: Hardware faults that matter: Understanding and estimating the safety impact of hardware faults on object detection dnns. In: Trapp, M., Saglietti, F., Spisländer, M., Bitsch, F. (eds.) Computer Safety, Reliability, and Security. pp. 298–318. Springer International Publishing, Cham (2022)

29. Rakin, A.S., He, Z., Fan, D.: Bit-flip attack: Crushing neural network with progressive bit search. In: IEEE/CVF International Conference on Computer Vision. pp. 1211–1220 (2019)
30. Reagen, B., Gupta, U., Pentecost, L., Whatmough, P., Lee, S.K., Mulholland, N., Brooks, D., Wei, G.Y.: Ares: A framework for quantifying the resilience of deep neural networks. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). pp. 1–6 (2018). https://doi.org/10.1109/DAC.2018.8465834
31. Ruiz, J.C., de Andrés, D., Saiz-Adalid, L.J., Gracia-Morán, J.: In-memory zero-space floating-point-based cnn protection using non-significant and invariant bits. In: Ceccarelli, A., Trapp, M., Bondavalli, A., Bitsch, F. (eds.) Computer Safety, Reliability, and Security. pp. 3–17. Springer Nature Switzerland, Cham (2024)
32. Ruiz, J.C., de Andrés, D., Saiz-Adalid, L.J., Gracia-Morán, J.: Tolerancia a fallos múltiples en redes convolucionales en coma flotante de 16 bits utilizando códigos correctores de errores. In: VIII Jornadas de Computación Empotrada y Reconfigurable (JCER2024), Jornadas SARTECO,. pp. 823–832 (2024)
33. Ruiz, J.C., de Andrés, D., Saiz-Adalid, L.J., Gracia-Morán, J.: Zero-space in-weight and in-bias protection for floating-point-based CNNs pp. 89–96 (2024). https://doi.org/10.1109/EDCC61798.2024.00028
34. Ruospo, A., Gavarini, G., de Sio, C., Guerrero, J., Sterpone, L., Reorda, M.S., Sanchez, E., Mariani, R., Aribido, J., Athavale, J.: Assessing convolutional neural networks reliability through statistical fault injections. In: 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6 (2023). https://doi.org/10.23919/DATE56975.2023.10136998
35. Sabbagh, M., Gongye, C., Fei, Y., Wang, Y.: Evaluating fault resiliency of compressed deep neural networks. In: 2019 IEEE International Conference on Embedded Software and Systems (ICESS). pp. 1–7 (2019). https://doi.org/10.1109/ICESS.2019.8782505
36. Saiz-Adalid, L.J., Gil, P., Baraza-Calvo, J.C., Ruiz, J.C., Gil-Tomas, D., Gracia-Moran, J.: Modified Hamming Codes to Enhance Short Burst Error Detection in Semiconductor Memories (Short Paper) . In: 2014 Tenth European Dependable Computing Conference (EDCC). pp. 62–65. IEEE Computer Society, Los Alamitos, CA, USA (May 2014). https://doi.org/10.1109/EDCC.2014.25
37. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: MobileNetV2: Inverted Residuals and Linear Bottlenecks . In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 4510–4520. IEEE Computer Society, Los Alamitos, CA, USA (Jun 2018). https://doi.org/10.1109/CVPR.2018.00474
38. Syed, R.T., Ulbricht, M., Piotrowski, K., Krstic, M.: Fault resilience analysis of quantized deep neural networks. In: 2021 IEEE 32nd International Conference on Microelectronics (MIEL). pp. 275–279 (2021). https://doi.org/10.1109/MIEL52794.2021.9569094
39. Y. Bengio, Y. Lecun, G.H.: Deep learning for AI. Communications of the ACM **64**(7), 58–65 (2021). https://doi.org/10.1145/3448250