A Smart Fault Detection Scheme for Reliable Image Processing Applications

Matteo Biasielli, Cristiana Bolchini, Luca Cassano, Antonio Miele
Dip. Elettronica, Informazione e Bioingegneria – Politecnico di Milano – Italy
{first_name.last_name}@polimi.it

***Abstract*—Traditional fault detection/tolerance techniques exploit multiple instances of the nominal processing and then perform a bit-wise comparison of the outputs to detect the occurrence of faults. In specific application scenarios, e.g., image/signal processing, the elaboration has an inherent degree of fault tolerance because it is possible to use the output even in the presence of slight alterations. In these contexts, the classical bit-wise comparison may be inefficient. Indeed, it may lead to conservatively discard outputs that have been only slightly altered by the fault and that could still be usefully exploited. In this paper, we propose a smart checking scheme based on Convolutional Neural Networks that rather than distinguishing between faulty and not faulty images, discriminates between usable and not usable images according to the ability of the end user to correctly process the output. The experimental evaluation shows that this solution enables an execution time saving of about 6.35% with a 99.42% accuracy, on average.**

***Index Terms*—Convolutional Neural Networks, Fault detection, Image processing, Smart checking**

I. Introduction

There are several application environments that are inherently tolerant to a certain degree of inexactness or inaccuracy, because i) some applications are natively meant to deal with noisy inputs (e.g., sensors), or ii) the output may be a probabilistic estimate (e.g., in machine learning applications), or eventually iii) the end user is a human, whose ability to perceive inexactness is by nature limited up to a certain level of detail (e.g., image and video processing applications) [1].

At the same time, in safety- and mission-critical application environments, some degree of reliability in terms of fault detection or tolerance is required even for such inaccurate data. Indeed it is a matter of an *acceptable* approximation of the data, or its *usability* for the given functional goal. Traditionally, fault detection/tolerance is achieved by duplicating/triplicating the system and then performing a bit-wise comparison of the produced outputs. Such well-known techniques allow for detecting/tolerating any single fault occurring in the processing. Nevertheless, in the image processing application scenario, such classical fault detection/tolerance based on bit-wise comparison may lead to inefficiency due to possible unnecessary re-executions. Indeed, a bit-wise comparison discards slightly altered images that might still be usable by the end user. In such scenarios a smarter checking scheme may allow to save time, still guaranteeing the required level of reliability. It is thus important to be able to detect the occurrence of those faults that corrupt the computation making the output image *not usable* by the downstream applications/users.

In this paper we exploit Convolutional Neural Networks (CNNs) [2], [3] for fault detection purposes in the image processing scenario, implemented on a legacy hardware platform. Our idea is to enhance the classical Duplication with Comparison (DWC) scheme by building the system with two identical copies of nominal processing functionality and by substituting the Two-Rail Checker (TRC) with an ad-hoc Smart Checker (SC) module. The SC compares the output images of the two identical replicas and classifies them into:

- **usable** images, correct images and images where the fault has a negligible impact, thus meeting the requirements of the downstream applications and/or users.
- **not usable** images, disrupted by faults.

The final goal of our proposal is to avoid re-execution of the processing filters when a fault occurred but the produced output image can still be effectively used, to save execution time and improve the overall performance.

The proposed approach has been applied to a software implementation of a case study image processing application for the identification of buildings in satellite photos. The baseline reference is a software execution on a single core commercial embedded microprocessor where tasks are scheduled in a time-triggered fashion. Execution time savings range from 0.93% to 10.80% w.r.t. the traditional DWC approach, providing a fault detection capability ranging from 98.28% to 99.95% depending on the considered configuration.

Nowadays state-of-the-art reliable designs in aerospace applications either employ special-purpose radiation-hardened chips [4], [5], which are much slower than commercial ones, or rely on traditional duplication/triplication of the system and on a bit-wise comparison of the produced results [6], [7]. In all cases, the incurred performance degradation is significant. On the other hand, there is a growing demand for both high performance and reliability in modern space applications, as recently discussed in [8]. Thus, we believe that the smart checking scheme proposed in this paper can represent a step towards fast and reliable applications in aerospace. To the best of our knowledge, no similar approach has yet been proposed.

The remainder of this paper is organized as follows: Section II and Section III present CNNs, and the adopted case study and the error models, respectively. Section IV discusses the proposed fault detection scheme and the design methodology for the corresponding SC module. Section V reports the experimental campaign we carried out on the considered case study while Section VI concludes the paper.

978-3-9819263-2-3/DATE19/©2019 EDAA

II. Background: Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of deep, feed-forward Artificial Neural Networks (ANNs) extensively employed in image classification [2], [3]. Like ANNs, CNNs are comprised of neurons that self-optimize through learning. The output of a CNN is a single classification score for each one of the available classes.

CNNs are multi-layer architectures, where each layer is designed to learn progressively from the previous layers, until the last one produces probability values. Based on their architecture, CNNs can take advantage of the 2D structure of input data, and with respect to images, CNNs are employed to autonomously identify the specific features of classes of images (learning from a training set of them) to then classify new images based on such features.

A CNN is composed of a **feature learning stage** followed by a **classification stage**. From a very high-level point of view, the former consists of a chain of several instantiations of two type of layers, namely the *Convolutional layer* and the *Pooling layer*. The first layer of the chain receives the input image and produces a matrix of values associated with the learned features, called activation map. Each internal layer in the chain receives an activation map from the previous layer and produces an activation map for the subsequent one. In details they perform the following activities:

- Convolutional layer: performs a set of 2D convolutions on the input image/activation map by using different filters: the weights of the filters are determined during the training of the CNN.
- Pooling layer: downsamples the activation map in input, to gradually reduce the dimensionality of the model.

The classification layer consists of a fully-connected ANN (usually referred to as *fully-connected layer*) performing the final classification based on the last produced feature map.

III. Running Example: An Application for Buildings Identification

As a running example and case study we implemented an image processing application meant for the identification of buildings in satellite photos. The considered application takes an image and outputs a matrix the size of the input image. Each matrix entry represents the likelihood of the corresponding pixel in the input image to belong to a building. For the reader's sake, the output matrix is here represented as a single-channel image, where each likelihood value is converted into pseudo-color intensity.

A. The Structure of the Considered Application

The selected image processing application is a pipeline composed of four image filters connected as depicted in Figure 1. There is a *Sharpening* filter (a convolutional filter that exploits the sharpening kernel shown in Figure 1), applied to the input image, two classification filters *Thresholding* and *Reshape&Convolution* applied to the sharpened image, and an *Aggregation* filter that aggregates the two outputs by performing a pixel-wise product.

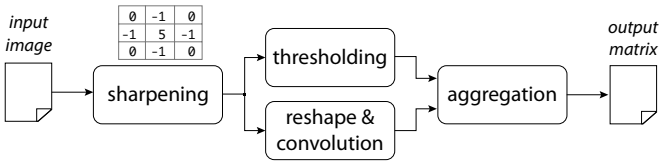

Figure 1. The structure of the considered image processing application

The two classification filters represent the core of the considered application. The Thresholding filter performs a pixel-wise and channel-wise classification, the Reshape&Convolution one performs a cross-channel analysis of groups of pixels. The rationale for having these filters working in parallel is that the subsequent Aggregation filter can alleviate errors introduced by one of the two filters.

B. The Effects of Faults in the Application Blocks

Faults affecting a microprocessor during the execution of a software may cause crashes, exceptions, timeouts or silent data corruptions. The first three categories lead the running application not to be completed with no output produced, therefore this kind of failures are typically managed by the operating system. Silent data corruptions allow the application to complete its execution in most cases, generating though an incorrect output. In this case, redundancy-based solutions are suitable approaches to achieve fault detection/tolerance.

We analyzed the effects of the faults on the final outputs, to be able to design an appropriate fault detection approach, based on the errors produced on the output matrix. To this aim, we employed the state-of-the-art fault injector for microprocessor systems, LLFI [9] and performed a fault injection campaign. LLFI allows to correlate the effects of faults occurring at the gate-level in a microprocessor with the errors potentially induced at the application-level. Faults have been injected in all stages of the application pipeline, and the following error models have been identified:

1) *image shift*, a part of the output matrix is circularly right-shifted or left-shifted;
2) *black (gray) area*, an initial or final part of the output matrix is set to all black (gray) values;
3) *black (gray) spots*, pixels or lines are set to all black (gray) values;
4) *random spots*, pixels or lines are set to random values;
5) *color change*, a constant change in the likelihood index of a part of the output matrix occurs (on an image it appears as a change in the luminosity, hence the name).

To exemplify these error models, Figure 2 reports the effects of injecting a fault in the microprocessor while executing the Thresholding filter, also classifying the output with respect to its usability. More precisely, Figure 2(a) is the input image, Figure 2(b) is the correct output matrix, when no fault occurs. Figure 2(c) and Figure 2(e) are the images at the output of the Thresholding filter, when a fault has occurred, belonging to the *color change* and *image shift* classes, respectively. The corresponding final outputs at the end of the processing flow are shown in Figure 2(d) and Figure 2(f), respectively. The

(a) Application input

(b) Golden output matrix

(c) Corrupted Thresholding output: *color change*

(d) Usable corrupted output matrix

(e) Corrupted Thresholding output: *image shift*

(f) Not usable corrupted output matrix

Figure 2. Examples of usable and not usable corrupted output matrix

identified buildings in the output matrices 2(b), 2(d) and 2(f) are highlighted by red boxes. While the first corrupted output matrix is still usable (Figure 2(d)), the second one (Figure 2(f)) is not.

C. Usable and Not Usable Image Classification

A simple pixel-wise difference between the actual and the golden output would not be effective since errors may affect parts of the matrix that are not relevant for the application or may only slightly affect relevant parts without compromising its usability. Thus, we designed a software module (dubbed *Oracle*) meant to emulate the end-user of the output who specifies whether the application is usable.

The Oracle takes in input the actual and the golden matrix and produces a boolean value. It consists of a convolutional filter and a thresholding filter. The former applies an all-ones kernel to highlight the higher values in the input matrices and

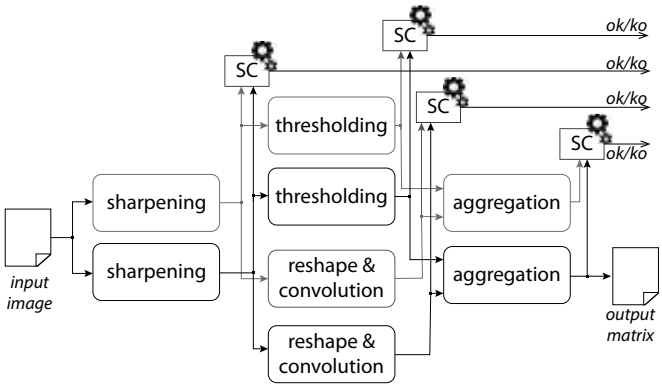

Figure 3. The proposed fault detection scheme

to suppress small errors. These two outputs are then compared pixel per pixel: if the difference of intensity value is higher than a configurable threshold th_{diff} an error is signaled. If the number of signaled errors is higher than a configurable threshold th_{err}, then the output of the pipeline is assumed to be not usable. As a result, the Oracle performs a block-wise (instead of a pixel-wise) comparison of the output w.r.t. the golden reference output.

IV. The Proposed Fault Detection Scheme and its Design Methodology

Our fault detection scheme builds on top of the classical DWC and extends it by substituting the classical TRC checker with the Smart Checker (SC) module. As shown in Figure 3, every filter is duplicated and their outputs are analyzed.

By using the classical bit-wise comparison, any single faulty pixel causes the output image to be considered faulty and triggers a re-execution, independently of the magnitude of the effect of the fault. The proposed SC module determines if the quality of the produced image is sufficient for being forwarded to the subsequent steps, assuming a meaningful final result can be achieved, even in the presence of errors. The goal is to prevent unnecessary re-executions to save execution time and limiting the impact of reliability-related overheads.

A. The Smart Checker

The SC module, shown in Figure 4, receives the global input image of the application and the output images of the two corresponding filter replicas and produces a Boolean value. It is composed of two *Pre-processing* blocks (a *Subtraction* and a *Resizing* block) and a *Classification* block, implemented by a CNN.

The underlying idea is that by comparing the outputs of the two replicated filters it is possible to identify whether a fault occurred during the processing, as well as the position and the magnitude of the error. This information, together with the input image is exploited by the CNN to infer (based on a previously performed training) whether the error would cause the final output not to be usable.

The subtraction block is meant to calculate the absolute value of the difference between the two inputs of the SC.

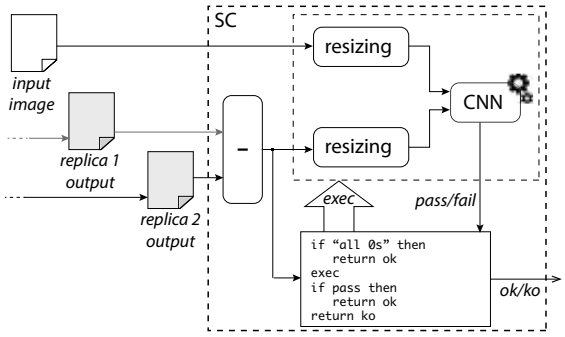

Figure 4. The proposed SC module

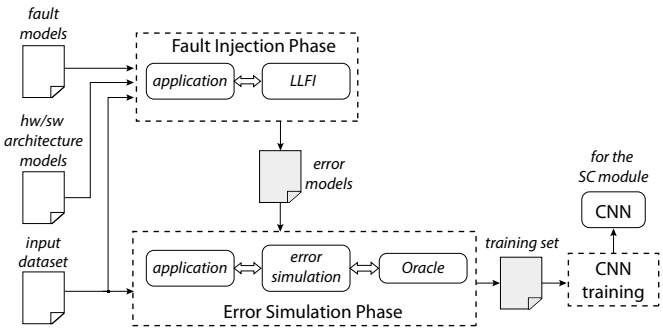

Figure 5. The training process for the proposed Smart Checker

This operation prevents training the CNN twice, a first time with faults occurring in the first replica, a second one for those occurring in the second replica. Furthermore, by computing the (absolute) difference between the replicas' outputs, the actual classification of the SC is invoked only when an error has occurred (the subtraction block does not produce all zeros), limiting the introduction of the computational overhead. The resizing block reduces the size of the two inputs of the CNN to make the classification process computationally feasible. The last block of the SC module is a CNN that performs the image classification for fault detection based on the difference between the outputs of the two filter replicas and on the application input.

B. The Smart Checker Design Methodology

The core of the proposed SC module is the CNN: it needs a careful training to be accurate and efficient at the same time and requires a relevant number of corrupted outputs. To create such a set, the *fault* injection campaign discussed in Subsection III-B has been used to extract error models. These models have then been exploited to run an *error* simulation campaign generating the set of corrupted outputs, used to train the CNN. Indeed, when performing fault injection in general, given a rich set of faults, only a limited number of corrupted outputs are produced while most of the time the fault is not activated or it is absorbed by the system itself, especially when focusing only on specific effects, such as the silent data corruption we are interested in. Hence, the collected set of corrupted data can be fruitfully used to identify *realistic error models*. These error models are then used to generate the large set of training data to feed the CNN. More precisely, it is more practical to use corrupted data created starting from the collected error models by applying them to the data itself to build a training set as large as necessary. The first fault injection campaign is fundamental to be able to create all and only errors that are realistic in the adopted working scenario, to prevent the creation of corrupted effects that cannot be the result of any real fault, thus invalidating the overall methodology. In the following, a brief discussion for each step is presented.

1) Fault Injection: As previously discussed, the aim is to identify error models on the outputs of the image processing task, when a fault occurs. Indeed, these effects depend on the employed filters and on the target platform (e.g., a software application running on a CPU or a hardware accelerator).

The input of this step is an implementation of the considered image processing application, a dataset representative of the images that the application will process at runtime, a description of the processing platform and the set of fault models that can occur. The output of this step is a set of error models that may affect the output of each filter if a fault occurs.

2) Error Simulation: The goal of this step is to generate for each filter a large training set comprising both correctly processed and corrupted images; each SC module for each filter pair in the duplicated application needs to be independently trained and thus needs its specific training set.

In accordance to the structure of the SC module, each entry of the training set contains three images and a label: the input image to be processed by the application pipeline, the filter fault-free output and the filter corrupted output. To correlate the corrupted output with the actual usability of the complete application output (i.e., the likelihood matrix) we add the boolean label that is the Oracle output when fed with the two golden/corrupted outputs.

3) Training: We employ the Adam optimizer [10] that exploits the benefits of both AdaGrad [11] and RMSProp [12] strategies, and has already proven to work better than other optimizers in many contexts. In particular, we decided not to employ the well-known gradient descend and back-propagation strategy [13] because of some problems it incurs in, such as local minima, training speed and vanishing gradient.

V. Experimental Analysis

A. Experimental Setup

We implemented the case study application and the SC module in C++, and the design flow in Python. TensorFlow implementation of CNNs [14] has been employed. Experiments have been performed on a Intel i5 machine.

We considered a software implementation of the processing pipeline running on a single-core architecture, with tasks scheduled in a time-triggered fashion. To compute performance gains, we assumed filters always to be scheduled in this order: Sharpening, Thresholding, Reshaping&Convolution and Aggregation (here abbreviated with S, T, R&C and A,

Table I
EXECUTION TIMES FOR FILTERS AND CHECKERS

Filter	Exec. Time
Sharpening	15.0ms
Thresholding	11.0ms
Reshape&Convolution	40.0ms
Aggregation	11.0ms
Proposed Smart Checker (when fault occurs)	4.4ms
Proposed Smart Checker (fault-free situation)	2.0ms
TRC	2.0ms

respectively). Execution times according to these working hypotheses are reported in Table I.

For both the training and test set we downloaded satellite images (by means of API provided by Microsoft Bing Maps [15]), acquired with the space resolution of 0.2m/pixel, and a 1,080x720 size. Each sets consists of 1,000 images of various cities, and each image is unique (no overlap between sets) and moreover the images from each city belong to one set only.

B. Results

We classify the output of the pipeline based on its usability according to the Oracle (usable **U**, and not usable **/U**), and the acceptance or rejection according to the methodology and the SC decision (discarded **D** as not usable, not discarded **/D** as accepted). The four classes we obtain are D /U, D U, /D U and /D /U, used to analyze the performance of our proposal against the traditional DWC approach.

A first analysis takes into account the effectiveness, in terms of fault detection capability, and the efficiency, in terms of time saving w.r.t. the DWC. For both strategies, we assumed that when a fault is detected, only filters whose outputs are identified as "corrupted and discarded" are re-executed (since it is not possible with this scheme to identify *which* copy is the corrupted one). For the resizing block of the SC module we considered an image resizing factor of 100x (both height and width are decreased by ten times) and a CNN structure having 2+6+4 convolutional layers (with 25 filters each) separated by max-pooling layers. Among the various CNN architectures we analyzed, the one here reported achieves the best trade-off between accuracy of the classification and increase of the execution time. We trained each SC module independently, by injecting errors only in the filters associated with the SC being considered. We injected 10 random errors (one at a time) per image in the training set, obtaining 10,000 training images. We tested each SC module independently; we injected 10 random errors (one at a time) per image in the test set, obtaining 10,000 test images. Results from this first experiment are summarized in Table II; the first column reports the filter, columns two to five report the percentage of D /U, D U, /D U and /D /U images, respectively, column six reports the average execution time when a fault occurs in the considered filter. The next three columns refer to the DWC scenario, where corrupted data is always discarded; columns seven and eight report the percentage of D U and D /U outputs, while column nine reports the execution time. Finally, column ten computes the performance improvement in terms of time saving for the proposed scheme w.r.t. DWC. In the DWC scheme the pipeline is always re-executed when a fault occurs. In the SC case the re-execution is conditioned by the output of the SC module. We do not report the times related to the fault-free cases because when no fault occurs the execution times of the proposed SC module and of the TRC are the same. Finally, in both cases, we assume a single fault affecting the pipeline execution, and the subsequent the re-execution to be fault-free. In all the experiments most of the usable outputs are not discarded: on average only 3.33% of the corrupted usable images are discarded (false positive, D U), and on average 52.33% of images are accepted even if corrupted, leading to the aimed time saving (6.35% on average, 10.80% for the R&C filter) demonstrating the advantage of our approach w.r.t. classical DWC. Such improvement is particularly relevant for filters requiring long execution times. Therefore, the proposed approach would achieve even better time savings in complex applications, employed in real world scenarios. As a final remark, the percentage of corrupted not usable images that are not discarded (false negative, /D /U – i.e. the cases when our approach fails) is always very small (1.01% on average).

To better substantiate the effectiveness of our approach we also compared against a second counterpart, implementing a less expensive approach from the computational point of view, based on the use of thresholds on the number of different pixels between the outputs of two replicas to distinguish between faulty and not faulty images. Different thresholds have been used: 1%, 5% and 10% of different pixels, and a configurable one (Avg%), where the threshold is set as the average number of different pixels found in the outputs discarded by the Oracle as corrupted during the training phase. Data related to these *naive* approaches with a focus on false positive/negative cases is reported in Table III, columns 6 to 13, compared against the proposed solution (columns 2 and 3) and the DWC (no false negatives). As it emerges, simplifying the detection scheme reduces the effectiveness of the methodology, aimed at accepting slightly corrupted images, for a more conservative but limited benefit on false negatives; as a conclusion the proposed smart checking scheme constitutes the best trade-off. It is worth noting that the execution time of the naive checkers is similar to that of the TRC, therefore we omit it.

Finally, we can consider that some errors that do not lead the output to be discarded at one stage in the pipeline, might cause a more relevant effect later, because of a subsequent filter manipulation such that a downstream SC does discard the corrupted output. The final set of experiments we performed aimed at analyzing the overall fault detection capability offered by the methodology as a whole. Table IV reports the analysis with respect to all four classes of output results, having simulated errors in any part of the application pipeline and benefiting by the combined smart checking performed by all modules. The number of false negatives is reduced, also improving the accepted corrupted images.

Table II
RESULT COMPARISON BETWEEN THE PROPOSED APPROACH AND THE DWC

	Proposed Approach					DWC			Time Saving
	D /U	D U	/D U	/D /U	Exec.Time	D U	D /U	Exec.Time	
S	40.26%	2.10%	56.87%	0.77%	101ms	43.62%	56.38%	102ms	0.93%
T	29.39%	6.36%	62.76%	1.49%	89ms	29.86%	70.14%	98ms	9.15%
R&C	40.73%	3.47%	55.75%	0.05%	113ms	44.21%	55.79%	127ms	10.80%
A	63.97%	1.38%	32.93%	1.72%	93ms	65.35%	34.65%	98ms	4.52%
Average	43.59%	3.33%	52.33%	1.01%	99ms	45.45%	54.55%	106ms	6.35%

Table III
FALSE POSITIVES (D U) AND FAULT NEGATIVES (/D /U) ANALYSIS

	Proposed Approach		DWC		Naive Approaches Based on Threshold % Faulty Pixels							
					1%		5%		10%		Avg%	
	D U	/D /U	D U	/D /U	D U	/D /U	D U	/D /U	D U	/D /U	D U	/D /U
S	2.10%	0.77%	42.36%	0.00%	6.11%	0.01%	2.54%	0.73%	1.11%	3.20%	0.18%	12.01%
T	6.36%	1.49%	29.86%	0.00%	15.50%	0.00%	11.36%	0.20%	7.01%	2.54%	3.41%	13.63%
R&C	3.47%	0.05%	44.21%	0.00%	9.93%	0.00%	6.42%	0.00%	3.85%	0.02%	0.34%	6.93%
A	1.38%	1.72%	65.35%	0.00%	14.72%	0.00%	4.94%	0.12%	1.18%	2.94%	0.78%	4.30%
Average	3.33%	1.01%	45.45%	0.00%	11.57%	0.00%	6.32%	0.26%	3.28%	2.16%	1.18%	9.22%

Table IV
FAULT DETECTION CAPABILITIES ANALYSIS

	D /U	/D U	D U	/D /U
Proposed Approach	47.10%	47.53%	4.80%	0.58%
DWC	47.68%	–	52.32%	–
1% Naive	47.68%	32.58%	19.75%	0.00%
5% Naive	47,68%	37.50%	14.83%	0.00%
10% Naive	47.68%	39.95%	12.38%	1.20%
Avg% Naive	46.48%	41.50%	10.83%	2.38%

VI. CONCLUSIONS AND FUTURE WORK

We presented a smart fault detection scheme for image processing applications, based on Convolutional Neural Networks (CNNs). The new scheme allows to overcome the classical sharp classification between corrupted/not corrupted outputs by introducing a new usable-not usable one, to provide mitigated overheads related to reliability when the output of the process is still usable. The obtained results show that the proposed scheme effectively identifies corrupted but still usable images, leading to a relevant time saving.

As future work, we plan to explore the influence of the structural parameters of the CNN on the effectiveness and efficiency of the SC module. We will we consider deeper and more complex image processing pipelines, to better tailor the benefits of the proposed solution.

REFERENCES

[1] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, 2016.

[2] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[3] F.-J. Huang and Y. LeCun, "Large-scale learning with svm and convolutional for generic object categorization," in *IEEE Conf. Computer Vision and Pattern Recognition*, 2006, pp. 284–291.

[4] J. Marshall, D. Rickard, D. Sova, H. Miller, R. Lapihuska, A. Dennis, and M. Graziano, "Heterogeneous high performance computing modules for next generation onboard processing," in *IEEE Aerospace Conf.*, 2017, pp. 1–10.

[5] J. Andersson, M. Hjorth, F. Johansson, and S. Habinc, "Leon processor devices for space missions: First 20 years of leon in space," in *Int. Conf. Space Mission Challenges for Information Technology*, 2017, pp. 136–141.

[6] M. Fayyaz and T. Vladimirova, "Fault-tolerant distributed approach to satellite on-board computer design," in *IEEE Aerospace Conf.*, 2014, pp. 1–12.

[7] L. Sterpone, M. Porrmann, and J. Hagemeyer, "A novel fault tolerant and runtime reconfigurable platform for satellite payload processing," *IEEE Trans. Computers*, vol. 62, no. 8, pp. 1508–1525, 2013.

[8] G. Lentaris, K. Maragos, I. Stratakos, L. Papadopoulos, O. Papanikolaou, D. Soudris, M. Lourakis, X. Zabulis, D. Gonzalez-Arjona, and G. Furano, "High-performance embedded computing in space: Evaluation of platforms for vision-based navigation," *Journal of Aerospace Information Systems*, vol. 15, no. 4, pp. 178–192, 2018.

[9] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *Proc. Int. Conf. Dependable Systems and Networks*, 2014, pp. 375–382.

[10] D. P. Kingma and J. L. Ba, "Adam: Amethod for stochastic optimization," in *Proc. Int. Conf. Learn. Representations*, 2014.

[11] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[12] Geoffrey Hinton, "Overview of mini-batch gradient descent," http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[13] S. Amari, "Backpropagation and stochastic gradient descent method," *Neurocomputing*, vol. 5, no. 4-5, pp. 185–196, 1993.

[14] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. [Online]. Available: www.tensorflow.org/

[15] Microsoft Corporation, "Bing Maps APIs," https://msdn.microsoft.com/en-us/library/dd877180.aspx, 2018.