ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **24**, Number 3, 2021, 299–320

Debugging FPGA projects using artificial intelligence

Alexandru DINU[1], Stefan GHEORGHE[1], Gabriel Mihail DANCIU[1], and Petre Lucian OGRUTAN[1]

[1]Transilvania University of Brasov, 29, Eroilor Avenue, Brasov, Romania
E-mail: alexandru.dinu@unitbv.ro

Abstract. Debugging digital designs implemented into FPGA devices is a challenging task. As opposite to simulation, simultaneous access to all input and output signals is not possible. The main obstacles in the debugging process are a limited number of input/output ports of FPGAs and the transfer of information from a digital device to an external data processor. However, debugging a digital design requests analysis of many combinations of inputs and outputs of a module to assess if these are well correlated and if their operation matches device specifications. The current work of the research team consisted of designing an end-to-end flow of data processing that fulfills the aim of debugging digital designs (particularly in this work, FPGA devices are considered). Firstly, a data generator based on majority voting idea was created using RTL languages. After checking its behavior using simulation, it has been downloaded into the FPGA fabric of a Spartan 3E board. The data generated from this reconfigurable device was acquired through the UART protocol, using an FT232R adapter. It was preprocessed to reconstruct the fields of each data sample and to remove transmission errors. The team analyzed the distribution of the obtained values and adjusted the data to achieve a uniform distribution. The team used the data to train both machine learning and deep learning models to create a golden reference model which accurately reflects the main functionality of the DUT: executing the majority vote operation over three pairs of numbers. Finally, the team presented how to use the resulting reference model to debug digital systems.

Key-words: data mining and analysis, FPGA, machine learning, deep learning, reference model, neural network configuration.

1. Introduction

The current industry relies on Field Programmable Gate Array (FPGA) in many domains. From molecular measurements [1] to financial information calculus [2], Field Programmable Gate Array (FPGA) devices prove their efficiency wherever intensive computing power and flexibility are needed. If there is a need for millions of specific processor-driven devices, Application Specific Integrated Circuit (ASIC) is the optimal solution. However, if only a few tens, hundreds,

or even thousands of a specific type of digital device are needed, using FPGAs is the right decision. Also, FPGA devices represent one of the best solutions when it comes to prototyping: "Sometimes, students and engineers just wanted to tinker with a design in a lab to prove a concept, or had only small production volumes in mind"[3].

Unlike already available built-in self-test (BIST) approaches which aim to check the correct operation of internal blocks in FPGA devices [4], this work assumes that the internal structure of FPGA is well operating. The focus of the current analysis is represented by determining if logic modules downloaded on FPGA fabric behave correctly.

Also, unlike ASICs, FPGA devices provide flexibility and make possible device updates and corrections. However, finding unobvious bugs in these types of custom processors is known to be tedious work. To determine where the problems of a digital design downloaded on FPGA are, the processing unit (e.g., a server used for testing) needs access to input and output signals (black box verification). Moreover, when deep analysis of FPGA internal structure is further required, access to internal signals is also needed (gray box or white box verification is applied). For parallelization of the mentioned accesses, a large number of physical probes is necessary. It makes unfeasible this way of collecting data: "to debug an FPGA prototype, probes are added directly to the Register-Transfer Level (RTL) design to make specific signals available for observation, synthesized and downloaded to the FPGA prototype platform. Large SoCs efficient debugging often requires concurrent access to hundred thousand of design signals" [5].

Another debug possibility is to take information from signals of interest through various internal buses of the FPGA board and to store them in available on-board Random Access Memory (RAM)[6]. To the best of the authors knowledge, this approach is adopted by Xilinx by using ChipScope™ Logic Analyzer [7], which employs Integrated Logic Analyzer (ILA) cores. Afterward, data can be transferred from RAM to the server. Nevertheless, implementing this possibility in regular projects assumes an increased workload for configuring information transfer between the processing unit and memory.

Moreover, in some projects, block RAM (BRAM) cells are already hosting design functionality and cannot be used anymore for debugging purposes. In this case, unused shift-register LUTs can be used for adding logic probes to signal of interest. "Once the defined trigger signal occurs, signal capture stops, and the captured values are shifted out to the JTAG port via inserted BSCAN primitives"[8]. However, it must be noted that "SRL-based trace buffers are relatively small relative to an ILA"[8].

Another study ([9]), that uses LUT memory for debugging purposes, proposes a software-like problem searching approach, where debug is performed using commands given in a console. Also, it calls for a comparison between the high-level language model of the design and its synthesized result. In contrast with this approach, the current proposed digital design is realized using RTL languages. Debugging of FPGA implemented designs can also be eased when soft GPU implementations are employed: "Overlay architectures are one of the approaches that enable fast development and debugging on FPGAs by providing software programmability" [10]. This approach configures the FPGA fabric with a compatible GPU implementation. However, this approach represents a sizeable effort for not very large projects.

FPGA debug also can be performed indirectly. The same module can be simulated using one of the available tools (for example, Questa®Advanced Simulator from Siemens EDA, SimVision™ from Cadence Design Systems, VCS®from Synopsys, etc.) and downloaded into FPGA logic. If in the "real world" the electronic circuit is misbehaving, its functionality stage (when the error occurs) will be researched deeper through simulations. During these simula-

tions, more signals values can be accurately noticed compared to the signals from the FPGA board (which are limited). If a problem is found, the design is fixed and afterward downloaded into the FPGA fabric. The downside of this method is that sometimes it is difficult to associate an error of the digital design with the event that triggered it.

Well performing debug methods of FPGA designs that embed neural networks are also developed across industry (e.g. [11, 12]). However, these are not intended for general usage, as current work aims.

The authors analyzed in [13] some means of automation of integrated circuits debugging. For current research, authors focused on the automation of debug process of digital designs implemented on FPGA devices.

The current proposed approach consists of collecting signal values from FPGA at specific periods through a distinct module implemented in the circuit fabric. These signals are transmitted further to Personal Computer (PC) through a UART interface. After data preprocessing, the obtained information is used to train machine learning and deep learning models, which will be able to classify FPGA behavior as correct or incorrect and even to tell which functionality from design is misbehaving. The usage of machine learning in FPGA-based projects is not new. Exploring particular FPGA capabilities such as reconfigurability, the work in [11] aims to create different machine learning applications.

2. Data generation

One of the most significant things in training an AI model is represented by a balanced distribution of training data across the possible range of values. Constrained-random data generation has a place among the best possibilities to achieve a balanced distribution. In the current project, the team has chosen the Linear Feedback Shift Register (LFSR) algorithm to generate random numbers.

Depending on the formula used, an LFSR circuit can generate sequences of data that have shorter or longer periodicity. To maximize the counter length of LFSR iterations, Formula (1) was applied over bits of the previous element (marked with x), obtaining the computation rule for the most significant bit (MSb) of each following element.

$$MSb = x^{12}\ xnor\ x^{6}\ xnor\ x^{4}\ xnor\ x^{1} \tag{1}$$

Thus, a stream of six data items that repeats every 3398 data items was created. A project based on this algorithm was implemented on an FPGA device embedded into the Spartan 3E board [14]. Notice the schematic of the digital project in Fig. 1

This project consists of a majority vote module having as input a stream of data coming from implemented LFSR generator. The module operates only when its input called “enable” is asserted. A human operator can toggle the input through a button located on the Spartan-3E board. To eliminate the mechanical vibrations that appeared when the button is pressed, a debouncer (a module that waits a fixed time before transmitting the signal change to the next steps in the processing chain) was employed.

LFSR module supplies data “on-demand” when the majority vote module asks for data (through the signal named “request_data_o”). For every three consecutive pairs of data received, the module computes where the biggest number of each pair was located: left (first number of pair) or right (second number of pair). In this case, the operands are 12-bits wide. Due to parameterizable code, the operands dimension could be changed, if needed, in another project. In the

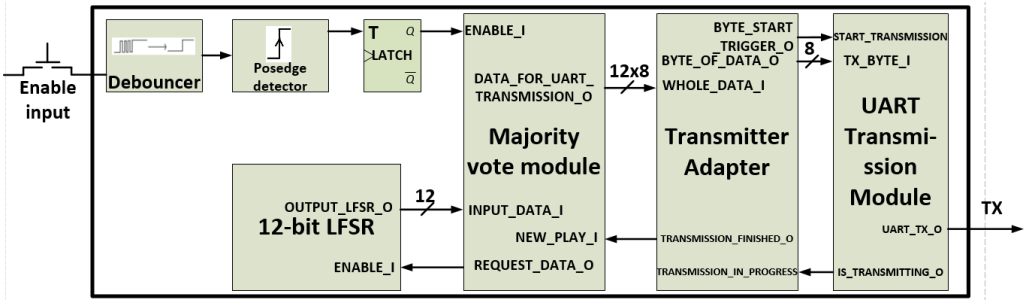


Fig. 1. Schematic of data generator implemented on FPGA.

end, it is counted (it is "voted") how many points the right side has and how many points the left side has. The possible scores are: 3-0, 2-1, 1-2, 0-3. The output of the voting module consists of each group of six numbers used as input during a voting process, along with resulted labels: "2" if "left" won and "1" when "right" won. This output is sent to the transmitter adapter that splits data into bytes and supplies it to the UART transmitter module, also implemented on FPGA. Parity bits are not used. Further, the connection between the chosen port of FPGA, configured as TX line and PC, is achieved using an FT232R adapter.

At this point, one limitation arises: the data transmitted through the UART connection must not be generated with a frequency bigger than the baud rate of serial connection. In the other scenario, where data is not sampled correctly, a part of the information items will be lost.

3. Data handling overview

PuTTY tool (which can be downloaded as Windows executable file from https://www.putty.org/) was used for receiving data from an FPGA device. Each byte received is displayed in a text editor as a symbol, according to the ASCII symbol.

Information is then transmitted to a parser developed from scratch in C++ language. The parser extracts from the received bytes the 12-bit width operands and vote result, writing on separate lines in a comma-separated values (CSV) file the afore-mentioned seven values for each voting process. By letting the LFSR algorithm wrap around about ten times, a file containing approximately 32000 data lines is generated. This file is used as input for tested machine learning classifiers.

4. Peculiarities of data preprocessing

In the PC, data is stored into a file that is further processed as follows:

Letting the parser read data until the end of the file, the team noted that when byte "1A" was read (ASCII code of SUBSTITUTE symbol), the parser considered it has reached the end of the file and ignored the remainder of its content. Therefore, all bytes containing the hexadecimal value "1A" (the ASCII code of SUBSTITUTE symbol) are manually replaced with a tilde symbol whose hexadecimal value equals "7E" (we opted for a symbol that is not likely to often occur). Further, data from all voting processes containing the "7E" symbol will be ignored because it could include inaccurate values.

Also, for the first version of hardware implementation on FPGA, carriage return (which has the hexadecimal value "D" in ASCII table) and line feed (which has the hexadecimal value A in ASCII table) symbols were transmitted as separators between data from two voting processes. The team has created a C++ program to parse the input file line by line, and one intriguing issue occurred: data contained between received bytes ASCII codes of carriage return or line feed symbols. Therefore, instead of receiving 12 bytes of data for each voting process, sometimes fewer bytes were received, as the real information was corrupted. This issue was solved by replacing CR and LF characters with 2 bytes having the hexadecimal value FF. Consequently, the C++ program parsed the input file byte by byte, and each time it met two consecutive FF bytes, it recognized that all values from a voting process were received. If only one FF byte had been received, it would have been processed as a usual data item. Also, if the second byte of FF is received, but in current measurement, less than 12 bytes were received, the parser noted that transmission was corrupted and ignored its data. Data preprocessing flow is represented in Fig. 2.

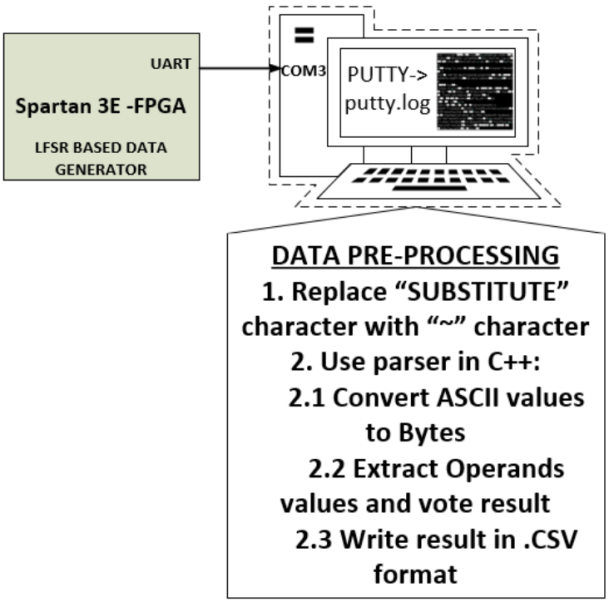
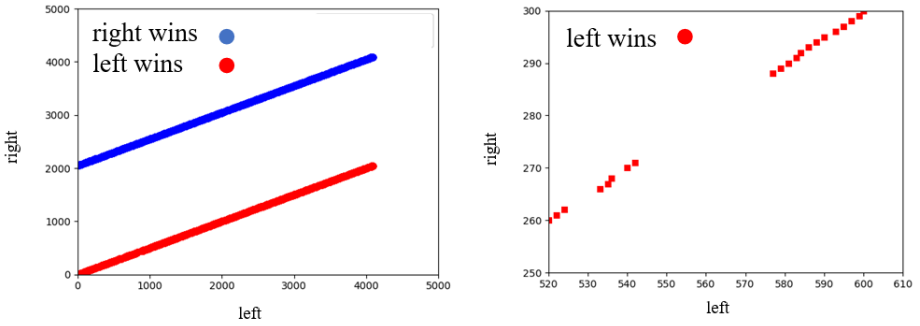


Fig. 2. Steps of data preprocessing presented in this paper.

5. Data analysis

By plotting the data of each generated pair of values, linear graphics from Fig. 3a and Fig. 3b were obtained. It demonstrates that data generated by an LFSR generator cannot be used as it is for training a machine learning model where diversity of cases is an important requirement.

Further, the team switched the right and left operands between some of the three pairs analyzed. Additionally, the left side was reversed with the right side for all the pairs created above. However, a still very organized graphic was produced, which demonstrates that data generated by



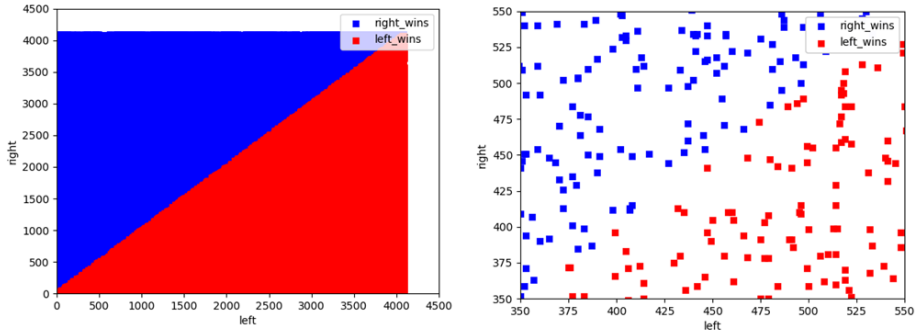
(a) Data items generated with LFSR

(b) Small section from data items generated by LFSR

Fig. 3. Data generated by LFSR before randomization of values order.

an LFSR generator cannot be used as it is for training a machine learning model where diversity of cases is an important requirement.

To assure that data will be well distributed within the generated range, a shuffling step was further required. Consequently, following instructions from [15], the elements from the right side of the pairs were shuffled, using randomly generated numbers as support. Thus, the team obtained proper data distribution, as seen in Fig. 4a and in Fig. 4b, where a part of the graphic is zoomed in.



(a) (b)

Fig. 4. Entire data generated by LFSR after randomization (a) and a section at boundary between the two situations (b).

Given the new obtained data, the confidence that trained model will have a low bias has increased. As shown in 2-features data analysis, data generated by LFSR is biased, fitting very well two lines. This aspect can be observed also in Fig. 5, where the left members of all pairs are represented. Marked in red are the events when the left part won during the voting process, and in blue the events when the right part won.

After randomizing the order of values received from the LFSR generator, the data was uniformly distributed across the values range, as shown in Fig. 6a. This figure plots all left elements in input data pairs: if the left element is lower than the right element of the pair, it is colored in blue; otherwise, it is colored in red. The same balanced mixture of data points can be better seen

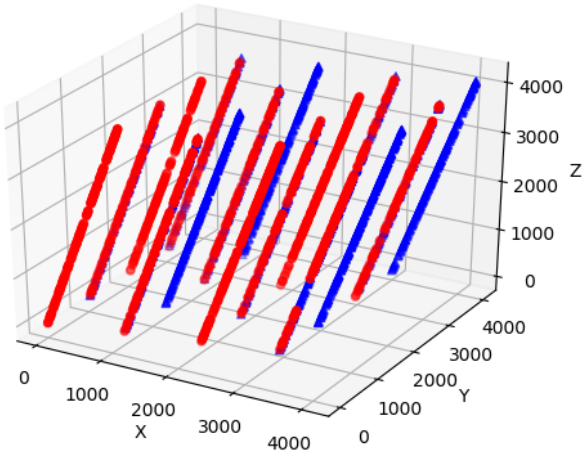
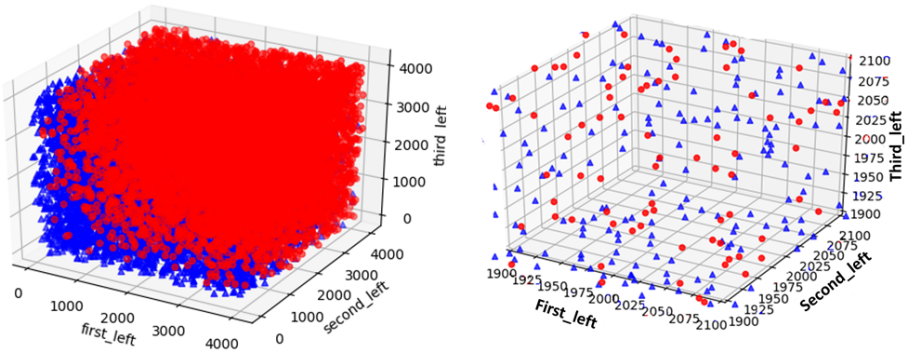


Fig. 5. Left members of data values pairs generated by LFSR algorithm.

in Fig. 6b, where the graphic is zoomed in.

Since the data distribution is now reliable, it can be used for creating a reference model which implements the DUT behavior. The current approach is to use the FPGA as a black box that provides AI algorithms with data. Further, based on a model trained with the received data, it is possible to diagnose a black box operation.

Moreover, if simple unsupervised learning algorithms are fed with data coming from FPGAs, outliers and special cases in functionality can be easily detected. An early step of debugging process is already achieved, given the fact that special cases could also represent bugs in digital design functionality. In the current research, the team explored two ways of employing artificial intelligence. One is the usage of supervised learning algorithms, and the second is the creation of neural networks with multiple hidden layers (deep learning).



(a) (b)

Fig. 6. Left members of data values pairs after randomizing the order of values seen entirely (a) and zoomed-in (b).

6. Approach and results obtained using machine learning classifiers implemented in scikit learn library

A machine learning algorithm, often called a model, is a complex mathematical function that can emulate a data processing system. The models can achieve different degrees of performance, depending on their structure and configuration. There are different approaches as well as use cases for the data analysis and representation. For this use case, the team has tested several supervised training methods which predicted the output of the algorithm implemented on FPGA based on known input. Using the dataset which contains approximatively 32800 samples - *a comma-separated values* file containing shuffled data for model training - multiple classifiers, were tested: K-Nearest Neighbor (KNN) [16], Support Vector Machine (SVM) [17], Random Forests [18], Nave Bayes (with several distributions), Logistic Regression [19]. There are different software libraries (in this work *scikit-learn* library [20] is used) that can provide each of these classifiers functionalities. Also, libraries can make it easier working with machine learning algorithms (examples of libraries for Python: pandas library [21], NumPy, matplotlib, SciPy [22]) by providing specialized functions for data handling or enabling GPU support for running algorithms (e.g. [23]). There are also tools (e.g., Jupyter Notebook or MATLAB environment) that support AI-based projects.

When data features have different ranges, it is necessary, before training a model, to scale their values to a common range. This action can be achieved in a Python environment, for example, using Expression (2), if data is stored using a *pandas.array* structure.

$$data = data.rank(pct = True) \tag{2}$$

Considering this point of view, in the current situation, since all training features have values between the same range (0 .. 4095), normalization would not have been needed. Nevertheless, there was another constraint: the SVM algorithm was unable to operate correctly with raw unnormalized values: all predicted values were "1". After data normalization, this classifier was the most successful. The other algorithm which was trained using normalized data in the current work is Multinomial Nave Bayes.

Excepting Nave Bayes, at each algorithm, a hyper-parameter was chosen to be set at different values. The best performance was obtained by using the SVM classifier (hyperparameter C= 500000); after training, its corresponding model presented 97.6% accuracy.

It is worth mentioning that accuracy was computed using *cross_val_score* function available in *scikit learn* library, with the data divided into training (80%) and testing parts (20%).

Therefore, in the current work, the diagnose system could be represented by the SVM machine learning algorithm, a supervised classifier. This classifier acts as a predictor for generated input. If the predicted output differs from the DUT output, this can indicate a potential bug in the FPGA project.

7. Approach and results obtained using deep learning techniques

If in [24], authors developed the reference models using machine learning algorithms exclusively, in this paper, the next level of implementing artificial intelligence, namely deep learning,

is accessed. Deep learning represents another step in AI domain evolution: machine learning approaches evolved into deep learning approaches, which are a more powerful and efficient way to deal with the massive amounts of data[25]. This concept is based on creating artificial neural networks (ANN) layer by layer and stating the regularization methods between each pair of consecutive layers.

In the current work, the training stage had in focus multiple network configurations, giving different values to some important parameters: number of epochs used for training the network, seed used for splitting data into training and test parts, test data percent (how much of entire data is represented by test data) and learning rate [26].

For the experiments presented below, the authors used two main architectures of neural networks (NNs): one architecture containing seven fully connected layers [27] (one input layer consisting of the six inputs used for each voting process, five hidden layers with a different number of neurons and one output layer), as can be seen in Fig. 7, and one architecture containing thirteen fully connected layers, whose layers are depicted in Fig. 8. Each picture contains on the upper part information extracted from the FPGA design. This data represents inputs of the first fully connected (FC) layer which has 6 inputs and 25 outputs. The 25 neurons at the output of the first layer are used as input for the next layer and so on. Each neuron has attached a value (weight) which during training is changed multiple times. The entire number of weights, along with biases, reflects the unique configuration of the neural network which is thus trained to give a specific output.

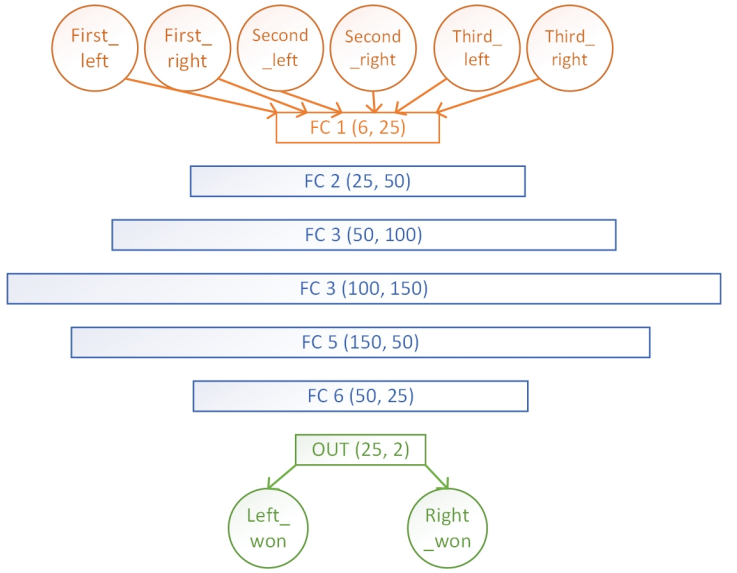


Fig. 7. Architecture of neural network with seven fully connected layers.

The output layer, marked with OUT (25, 2) for first architecture (Fig. 7) and with OUT (10, 2) for second architecture (Fig. 8) represents the last step of computing network result. There, the two output neurons computed based on the previous 25 (or 10, for second architecture) neurons, corresponding to the two possible responses of the network: 2 if the left side won or 1 if the right side won (NNs operate in the current case as binary classifiers).

Starting from these two architectures, several neural networks were built by adding and com-

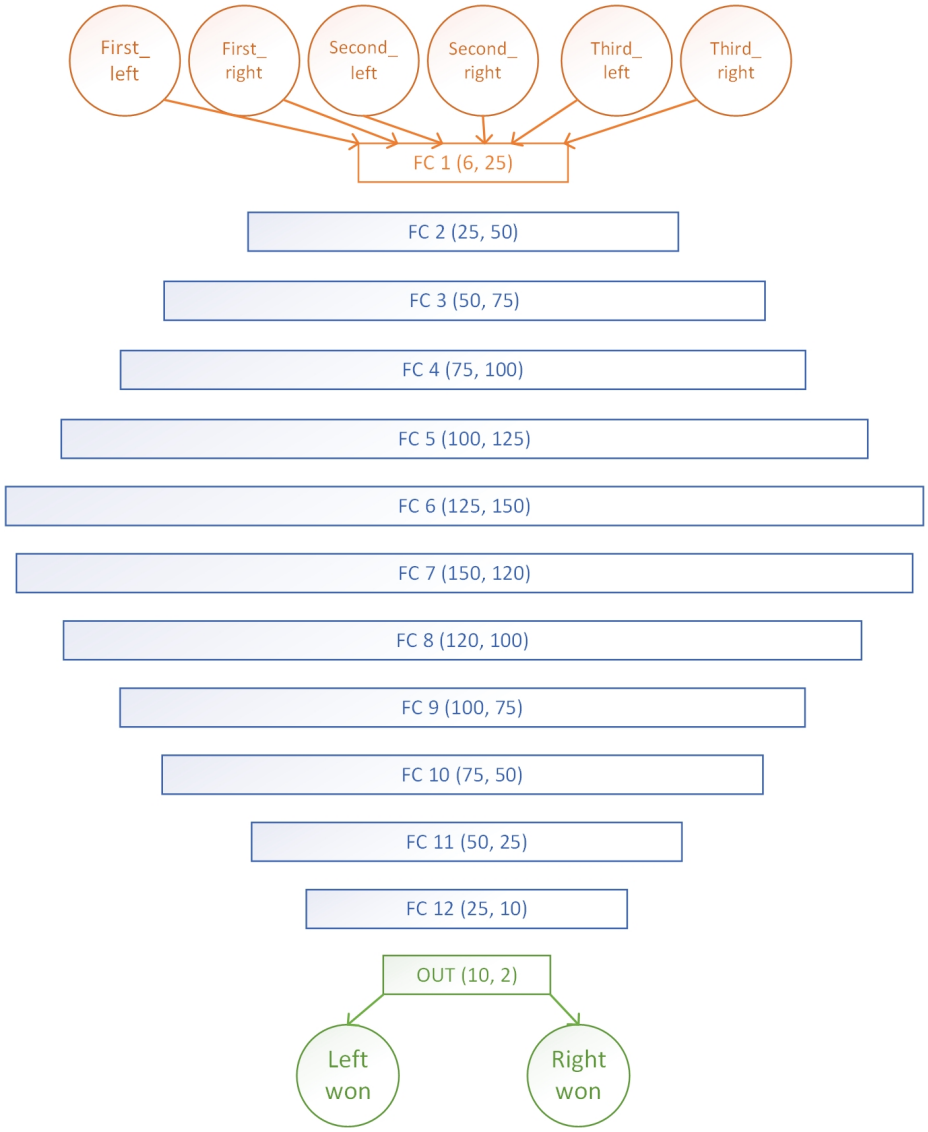


Fig. 8. Architecture of neural network with thirteen fully connected layers.

bining two regularization methods: drop-out (through this approach, some connections between neurons are removed; thus, links between layers become less complex, and the overfitting possibility is reduced; p parameter is used to establish which percentage of neurons will be drop-out) and batch normalization (neurons weights values are also normalized; successive modifications of network weights does not affect entire network; thus, each optimization step becomes more meaningful, and oscillations in loss function graphic are reduced).

In total, eleven versions of networks based on first architecture and seven versions of networks based on second architecture were created, as can be seen in Table 1. After each training

iteration, the weight and biases of neural network are updated using the Adam Optimizer. The evaluation metric for networks' performance was computed accuracy based on Adam Optimizer, over the test data set. The performance of each neural network is computed each training epoch using Cross Entropy Loss function available in PyTorch library. Also, two activation functions were employed: linear and sigmoid. Another activation function which could be used for classification is softmax.

Table 1. Configured neural networks during current work

Version	No. of layers	Input and hidden layers configuration	Output layer activation function	p (for Drop-out)
v1	7 layers	6x (FC, ReLU)	Linear	n/a
v2	7 layers	4x (FC, ReLU), 2x (FC, ReLU, Drop-out)	Linear	0.2
v3	7 layers	6x (FC, ReLU, BatchNormalization)	Linear	n/a
v4	7 layers	4x (FC, ReLU), 2x (FC, ReLU, Drop-out) (identical with v2)	Linear	0.2
v5	7 layers	4x (FC, ReLU), 2x (FC, ReLU, Drop-out)	Linear	0.2
v6	7 layers	4x (FC, ReLU, BN), 2x (FC, ReLU, BN, Drop-out)	Linear	0.2
v7	13 layers	5x (FC, ReLU, BN), 7x (FC, ReLU, BN, Drop-out)	Linear	0.2
v8	13 layers	5x (FC, ReLU, BN), 7x (FC, ReLU, BN, Drop-out)	Linear	0.2
v9	13 layers	5x (FC, ReLU, BN), 1x (FC, ReLU, BN, Drop-out), 6x (FC, Sigmoid, BN, Drop-out)	Linear	0.2
v10	13 layers	5x (FC, ReLU), 7x (FC, ReLU, Drop-out)	Linear	0.2
v11	13 layers	5x (FC, ReLU, BN), 7x (FC, ReLU, BN, Drop-out)	Linear	0.2
v12	13 layers	5x (FC, ReLU, BN), 7x (FC, ReLU, BN, Drop-out)	Linear	0.2
v13	7 layers	4x (FC, ReLU), 2x (FC, ReLU, Drop-out)	Linear	0.2
v14	7 layers	4x (FC, ReLU), 2x (FC, ReLU, Drop-out)	Linear	0.2
v15	13 layers	12x (FC, ReLU, BN, Drop-out)	Linear	0.2
v16	7 layers	6x (FC, ReLU, Drop-out)	Linear	0.2
v17	7 layers	6x (FC, ReLU, BN, Drop-out)	Linear	0.2
v18	7 layers	6x (FC, ReLU, BN, Drop-out)	Linear	0.2

The network which offered the best accuracy used the fifteenth architecture, and best it succeeded when it was configured as follows: learning rate = 0.02; data used for test = 40% of entire data; seed (used for random splitting data in train and test sets with scikitlearns *train_test_split* function) = 3245; p parameter for drop-out layer (probability for each neuron to be kept in the network) = 0.5 for the first layer, 0.3 for the second layer and 0.2 for all the other layers in the network, excepting the output layer which was a linear layer. At 600 epochs (iterations), the networks accuracy was 99.749%. Its structure can be seen in Fig. 9, where FC is the abbreviation for Fully Connected and ReLU is the abbreviation for Rectified Linear Unit

The evolution of the network performance during the training process is reflected in loss function evolution against the number of epochs (Fig. 10a). Cross entropy was employed as a loss function since it is the most commonly used loss function for training NNs used for classification [28]. For this reason, function *CrossEntropyLoss()* available in *torch.nn* library was called during network training. Common calculation rule of loss function computed using cross-entropy, in the current situation, for each data sample, when binary classification is performed, can be found in Formula (3).

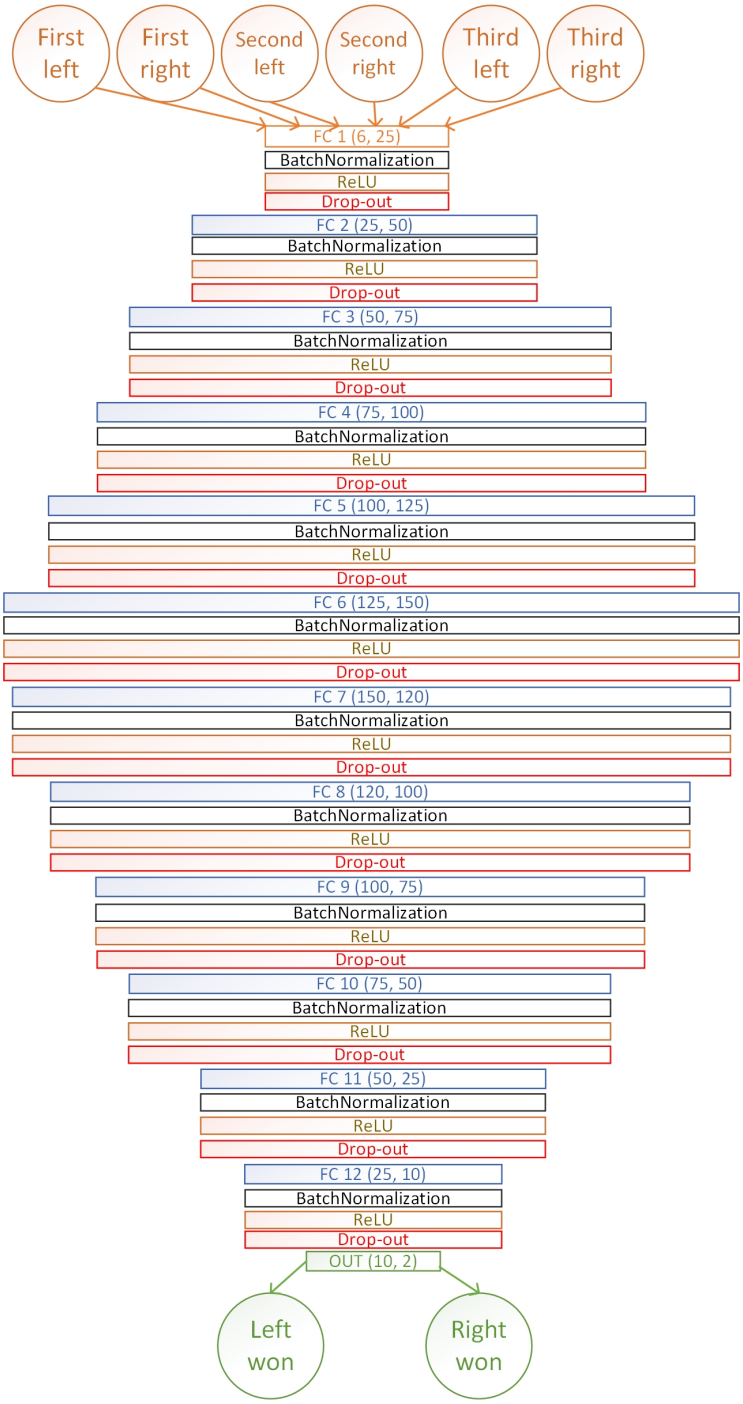
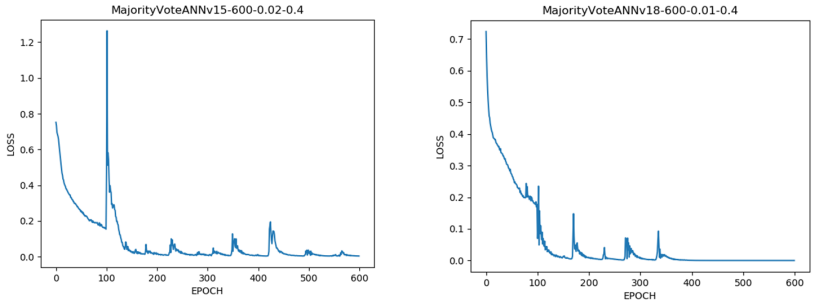


Fig. 9. Structure of best performing ANN in current work (based on second architecture).



(a) (b)

Fig. 10. Structure of best performing network based on first architecture in current work.

$$\begin{aligned} CrossEntropyLoss = -(&real\_label(classLeftWon) \times \\ &\times ln(predicted\_label(classRightWon)) + \\ &+ (1 - real\_label(classLeftWon)) \times \\ &\times ln(1 - predicted\_label(classRightWon))) \end{aligned} \tag{3}$$

In (3), *classLeftWon* represents class coded with 2 in this work, and *classRightWon* represents class coded with 1 in this work. *real_label()* function represents data given by DUT as the result of the voting process for each data sample. *predicted_label()* function represents the predicted result of the voting process returned by ANN operating as a binary classifier.

Table 2. First 10 network configurations which best performed during current works training.

Model name	Training time(s)	Epochs	Seed	Test data percent (%)	Learning rate	Accuracy (%)
MajorityVoteANN v15-600-0.02-0.4	980.4	600	3245	40	0.02	99.74916
MajorityVoteANN v18-600-0.01-0.4	349.5	600	3245	40	0.01	99.74156
MajorityVoteANN v18-600-0.01-0.3	443.0	600	3245	30	0.01	99.7365
MajorityVoteANN v18-400-0.01-0.4	229.9	400	3245	40	0.01	99.73396
MajorityVoteANN v18-500-0.01-0.4	287.5	500	3245	40	0.01	99.73396
MajorityVoteANN v15-600-0.01-0.3	967.0	600	3245	30	0.01	99.70609
MajorityVoteANN v15-500-0.01-0.3	813.8	500	3245	30	0.01	99.68582
MajorityVoteANN v18-400-0.01-0.3	313.2	400	3245	30	0.01	99.67569
MajorityVoteANN v18-300-0.005-0.4	174.5	300	3245	40	0.005	99.75795

According Table 2, very close in terms of performance to the winner model, the best-performing network based on the first presented architecture obtained an accuracy value of 99.741%.

The values of hyperparameters used for this ANN were: learning rate = 0.01; data used for test = 40% of entire data; seed = 3245; p parameter for drop-out layer = 0.5 for the first layer, 0.3 for the second layer, and 0.2 for all the other layers in the network, excepting the output layer which was a linear layer. Its structure can be seen in Fig. 11 and the evolution of the network during the training process is depicted in Fig. 10b. Also, to allow the reader to compare current obtained results for each trained neural network, Table 3 shows the best results achieved by each proposed NN model. Also, in Fig. 12, a comparison over the oscillations of Loss functions (considering epochs number) for each neural network model can be observed. For better readability, the boundaries of the Y-axis have been emphasized.

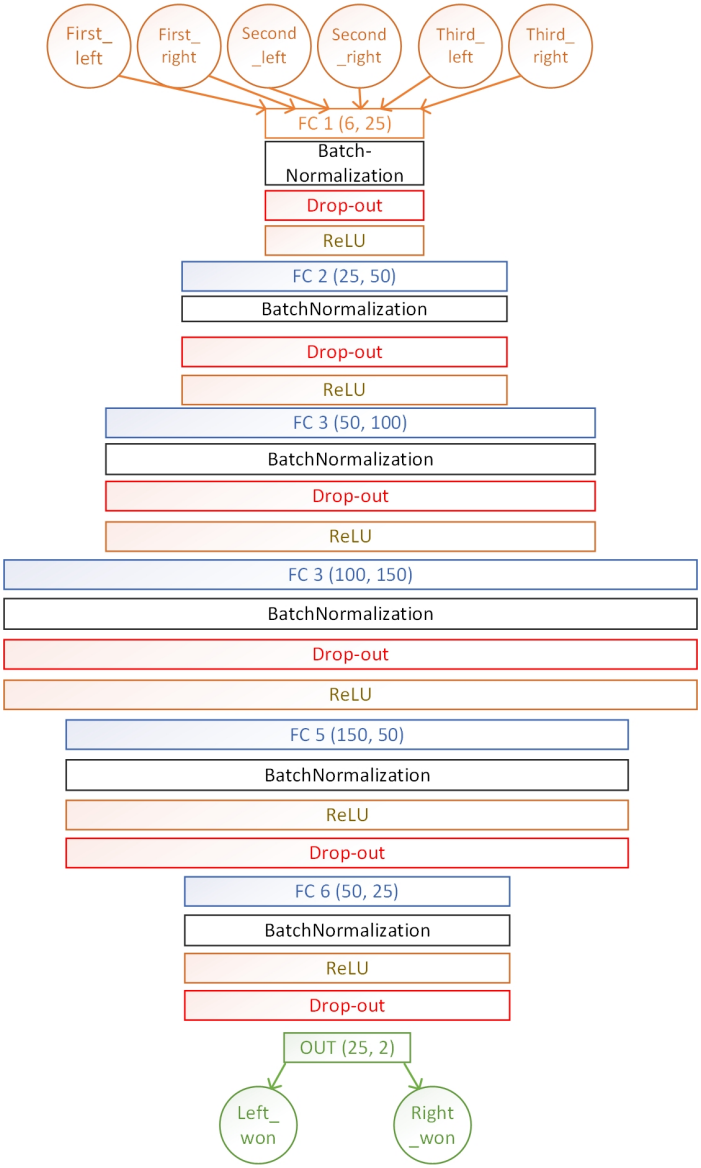


Fig. 11. Structure of best performing network based on first architecture in current work.

Table 3. Neural networks configurations are ordered considering their best obtained accuracy over test set.

Model name	Training time(s)	Epochs	Seed	Test data percent (%)	Learning rate	Accuracy (%)
MajorityVoteANN v15-600-0.02-0.4	980.4	600	3245	40	0.02	99.74916
MajorityVoteANN v18-600-0.01-0.4	349.5	600	3245	40	0.01	99.74156
MajorityVoteANN v17-500-0.01-0.3	364.5	500	3245	30	0.01	99.7365
MajorityVoteANN v16-600-0.01-0.4	145.7	600	3245	40	0.01	99.73396
MajorityVoteANN v3-500-0.02-0.4	286.7	500	3245	40	0.01	99.73396
MajorityVoteANN v12-600-0.01-0.4	887.3	600	3245	30	0.01	99.70609
MajorityVoteANN v5-496-0.005-0.3	404.0	495	543	30	0.01	99.68582
MajorityVoteANN v2-600-0.02-0.3	132.3	600	3245	30	0.01	99.67569
MajorityVoteANN v4-600-0.02-0.3	118.1	600	3245	40	0.005	99.75795
MajorityVoteANN v11-600-0.02-0.3	1099.6	600	1073	40	0.005	99.75795
MajorityVoteANN v13-600-0.02-0.4	135.1	600	3245	40	0.005	99.75795
MajorityVoteANN v1-600-0.02-0.4	119.1	600	3245	40	0.005	99.75795
MajorityVoteANN v14-300-0.01-0.3	223.1	300	3245	40	0.005	99.75795
MajorityVoteANN v6-500-0.02-0.4	312.2	499	1573	40	0.005	99.75795
MajorityVoteANN v8-400-0.01-0.4	561.7	399	1573	40	0.005	99.75795
MajorityVoteANN v7-500-0.01-0.3	789.1	499	1573	40	0.005	99.75795
MajorityVoteANN v10-400-0.005-0.3	311.6	399	1573	40	0.005	99.75795
MajorityVoteANN v9-400-0.02-0.3	646.6	399	1573	40	0.005	99.75795

Since for the majority voting problem, there was no need for networks with hundreds of layers (which were likely to model more closely the training data, thus being prone to overfit), the chosen neural network architectures had not more than thirteen layers. Thus, the resources of the computer used in this research (based on Intel®Core™ i7-6498DU CPU @ 2.50GHz processor, with two physical cores and four logic cores) were enough to fit the training process requirements.

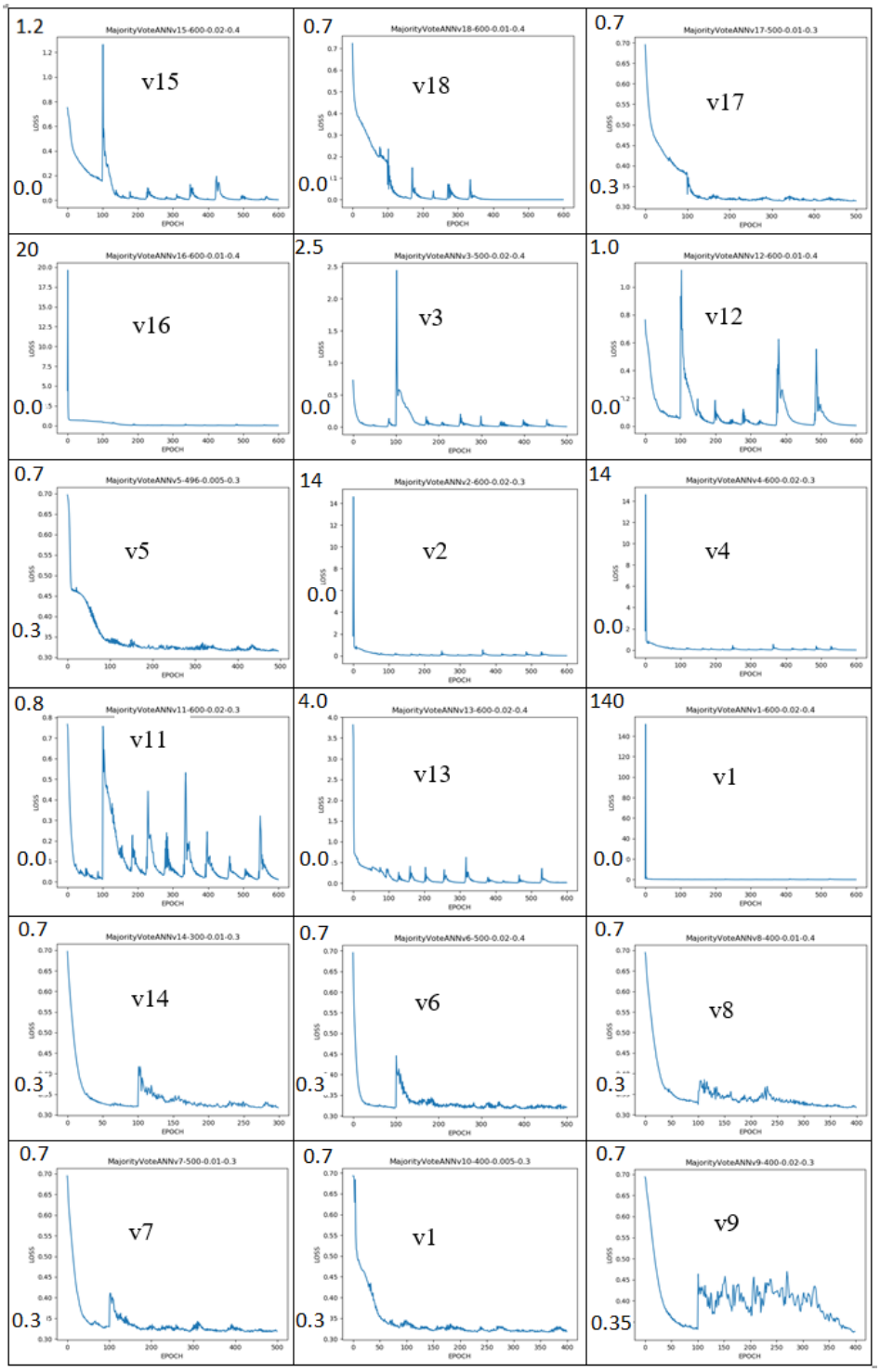


Fig. 12. Loss function graphics of each best performing configuration of neural network model.

The same training processes run with the help of a GPU or FPGA would be significatively faster. Also, to ensure that networks did not overfit during the training process, the testing data set was established to be at least 30% of the entire data measurements).

By training an ANN and tuning it to achieve good accuracy, the researchers have obtained a golden reference model for DUT behavior. From now on, the DUT further operation can be checked whenever needed to point out any anomalies. In an industrial project, continuous checking can be implemented by attaching a processor to the monitored device (DUT).

Thus, DUTs input and output data are read, and the processing unit verifies if output data match numbers predicted by the reference model. This original information processing flow, designed in the current paper, is represented in Fig. 13.

Another aspect becomes obvious at this point: data used for training the ANN must be error-free. The common garbage in, garbage out principle applies to ML, meaning that the quality of the predictions made by the model will never exceed the quality of the training set used to build the model[29]. In this case, the use of unsupervised learning methods could be considered too, for automation of data checking, for removing the outliers, etc., as this artificial intelligence technique already gives good results in the domain [30].

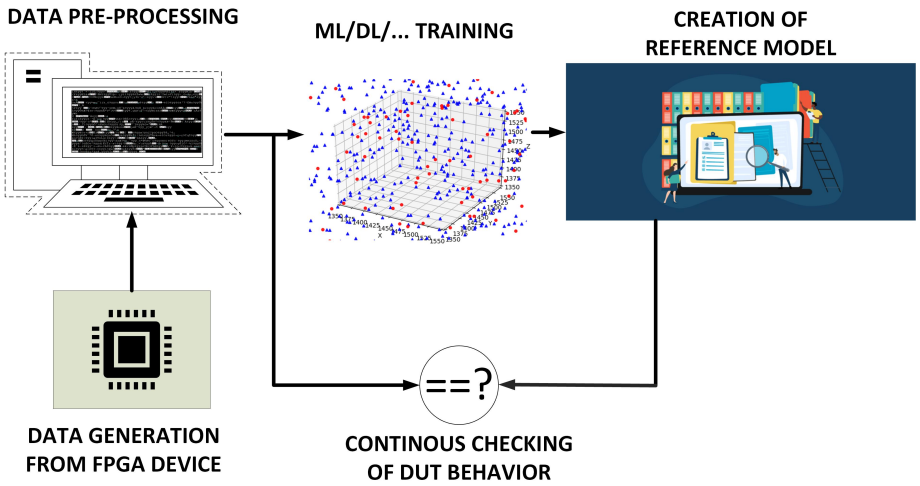


Fig. 13. FPGA complete debugging flow described in this paper.

8. Discussion

Although classic machine learning algorithms tested in this paper obtained good results in creating reference models, designing artificial neural networks through a deep learning approach exceeded their performance. The best performing machine learning algorithm (SVM classifier, C parameter=500000) was trained for 1134.64 seconds and obtained an accuracy of 97,6%. Using the same CPU, the best performing deep learning designed ANN was trained for 980.4 seconds and has obtained an accuracy of 99,749%.

Therefore, it is worth analyzing how artificial neural networks were configured and trained to achieve high performance. After obtaining a table with more than 1300 lines comprising training results over different neural networks configurations (the top of the database can be seen in Table

2), this research revealed and proved some properties of neural networks. Their configuration differences were clearly emphasized in Table 1.

Comparing v1 and v2 model versions, the drop-out regularization method was added for layer 5 and layer 6 of the v2 network. This aspect has highly contributed to increasing the accuracy of the v2 neural network from a maximum of 99.03% accuracy to a maximum of 99.27%. This observation was proven again by creating the v16 neural network model where, additionally, each layer, but last, was regularized using the drop-out method. Its best performing training instance obtained 99.45% accuracy, thus having the #4 rank in the results database. ANNs using this method outperformed the other similar networks which did not use it: according to Table 3, v18 and v16 ANN obtained higher performance than v3 and v1, respectively. Therefore, the drop-out regularization method was used in almost all designed network models. The current work also proved the statement that the drop-out regularization offers a very computationally cheap and remarkably effective regularization method to reduce overfitting and improve generalization error in deep neural networks of all kinds [31].

Learning rate is considered the most relevant parameter when training a deep neural network [32], as its influence has a major impact over network performance and loss function evolution. In the current work, four values of learning rates (0.001, 0.005, 0.01, 0.02) were initially used, but it was observed that for the 0.001 value, the network models obtained low performance in terms of accuracy. Therefore, the other three values were kept for the training of most networks. The learning rate is primarily used to set the step size of the modifications over neural network weights after each training iteration. Thus, a higher learning rate leads to significant consecutive alterations of model accuracy, too. This aspect was seen in current work during training of several NNs (the link between learning rate parameter and loss function evolution can be seen in Fig. 14). However, other training processes did not stick to this principle in several cases. One obvious reason for that is the fact that accuracy is not linked exclusively to the learning rate.

Also, in this paper, the authors used the rectified linear activation function for all layers, except the output layer. They followed state-of-the-art approaches. For modern deep learning neural networks, the default activation function is the rectified linear activation function[34]; Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid activation function or the hyperbolic tangent activation function[33]). As written in [35], attaching the sigmoid activation function to output layer of a neural network improves the models performance. However, it is specified that the sigmoid and hyperbolic tangent activation functions cannot be used in networks with many layers due to the vanishing gradient problem[34]. In the current work, the four networks models of thirteen layers whose output layer contained sigmoid neurons (v7, v8, v9 and v10) obtained the lowest performance (see Table 3). Contrary to this, when the sigmoid activation function was used for the last layer of networks based on seven layers architecture, it did not crucially influence network performances, given the fact that the v17 network model was ranked as third best performing neural network and v17 ANN performed worse than v18, his sibling network which had the default linear activation function for the output layer. On the other hand, v5 ANN which used sigmoid performed better than v4 (which is identical with v2) ANN, which used a linear default activation function for the last layer. Also, v9 ANN which contains six layers using the Sigmoid activation function obtained the lowest performance. In the current work, the authors tested the usage of the sigmoid activation function. They reached the same conclusion from [34] stating that sigmoid activation function is not proper for networks having many layers.

On the other hand, batch normalization (BN) regularization method provided ANNs with an

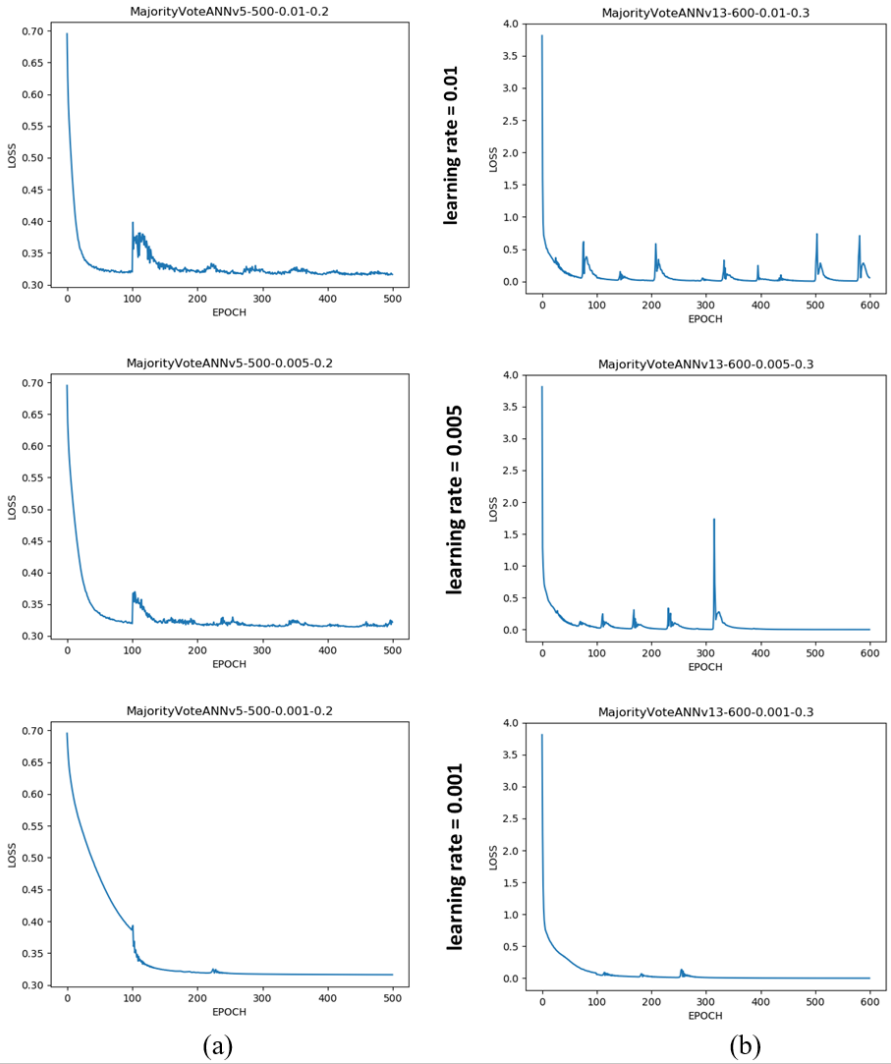


Fig. 14. Effect of learning rate over the loss function evolution when v5 (a) and v13 (b) network models were trained.

increased accuracy. Considering Table 1 and Table 3, four of the five best performing networks used BN. Also, v18 and v3 ANNs using BN outperformed similar BN-less networks v16, v1 respectively. Thus, even if the networks inputs belong to the same range (12 bits wide), the weights of neurons of different hidden layers can take highly different values, and normalization is still necessary. BN acts to standardize only the mean and variance of each unit in order to stabilize learning, but allows the relationships between units and the nonlinear statistics of a single unit to change[33].

Also, as a general statement, during current work, it was observed that NN training time increased when the network had multiple layers. This idea is very well reflected in the first two entries of Table 3 where best performing 7-layers NN was trained almost three times faster than best-performing 13-layers NN, given the same number of epochs.

The development of a golden reference module with the method presented in this paper can also be useful in the testing process of FPGA or integrated circuits (ICs). After a reference model which checks different functionalities of a DUT was created, it can be used to check all devices built to implement the project functionality. If functionality is implemented in hardware, integrated circuits and FPGAs are in focus; for software implementation, (micro)processors can be referred; if the functionality of the digital design is implemented both in hardware and software, System-on-Chip projects [36] can be given as examples. In this way, for example, tests for integrated circuits which were just produced in factories can be fast implemented (as software test benches with physical access at ICs pins through a general setup of probes), and the items containing manufacturing defects can be early removed from next production steps.

Moreover, considering that created reference model can continuously point out functional bugs of digital design implemented in FPGA, it can represent a safety and security utility. Being implemented into a monitoring device, the DUT functionality can be checked continuously, and functional mismatches can be signaled on time and with no effort.

If at some point during monitoring, erroneous behavior of digital design is detected, the digital project must be fixed and redownloaded in FPGA firmware. Moreover, if the DUT needs to be upgraded to fit the new functional requirements, then the reference model needs to be upgraded, too. Due to the highly automated data processing flow which uses data processing scripts, this step will become much more straightforward. In this case, the principal requirement for data scientists remains to find out which is the new best performing ANN, to select it as the reference model, or to tune a new network that could outperform the already created NNs for outdated DUT.

9. Conclusion

The data parser presented in this paper was implemented in an original way, where the information collected from hardware is prepared to be processed in software. A very important requirement for proposed flow is that data scientists or engineers must make sure the digital design was well operating when they collected data used further for training a model using the supervised learning paradigm. The main limitation in using the presented approach is the transmission of data to the processing unit (e.g. PC). If data is generated with a frequency that is larger than the UART transmission baud rate, some data samples will be lost. Therefore, engineers must pay attention to this aspect when they collect data from FPGA-based boards.

With the creation of the reference model, which can be used for real-time checking, we proposed an original workflow for debugging the designs implemented on FPGAs. Considering [32], this paper was also enriched using a deep learning approach for building reference models. Collection of data from FPGA through UART connection, data reconstruction and storage in a ready-to-supply-information table, usage of artificial intelligence techniques to create a golden reference model and checking of DUT behavior against high-accuracy built model are arguments to conclude that we achieved good progress towards obtaining a complete automated debugging system for designs implemented in FPGA fabrics and for other digital systems.

The obtained reference model can be used to regularly check if digital logic implemented in FPGA continues to operate well. If problems are found during DUT monitoring, the highly automated data processing flow presented in the current paper combined with the flexibility of FPGA devices can facilitate the fast rehabilitation of the erroneous component. Thus, systems robustness increases and the changes in production flow can be easier accommodated, too.

References

[1] ALAM S.R., AGARWAL P.K., SMITH M.C., VETTER J.S., CALIGA D., *Using FPGA Devices to Accelerate Biomolecular Simulations*, Computer **40**(3), pp. 66-73, 2007.

[2] INDECK R.S., CYTRON R.K., FRANKLIN M.A., CHAMBERLAIN R.D., *Method and apparatus for processing financial information at hardware speeds using FPGA devices*, Google Patents, 2011.

[3] NENNI D., DINGEE D., *Prototypical: The Emergence of FPGA-Based Prototyping for SoC Design*. S2C Inc, 2016.

[4] ABRAMOVICI M., STROUD C.E., *BIST-based test and diagnosis of FPGA logic blocks*, IEEE Transactions on Very Large-Scale Integration (VLSI) Systems **9**(1), pp. 159-172, 2001.

[5] EL MANDOUH E., WASSAL A.G., *Accelerating the debugging of fv traces using k-means clustering techniques*, Proc. of 11th International Design & Test Symposium (IDT), pp. 278-283, 2016.

[6] HALE R., HUTCHINGS B., *Distributed-Memory Based FPGA Debug: Design Timing Impact*, Proc. of 2018 International Conference on Field-Programmable Technology (FPT 2018), pp. 353-356, 2018, doi: 10.1109/FPT.2018.00071.

[7] ARSHAK K., JAFER E., IBALA C., *Testing FPGA based digital system using XILINX ChipScope logic analyzer*, Proc. of 29th International Spring Seminar on Electronics Technology, 2006: IEEE, pp. 355-360, 2006.

[8] HALE R., HUTCHINGS B., *Enabling Low Impact, Rapid Debug for Highly Utilized FPGA Designs*, Proc. of 28th International Conference on Field Programmable Logic and Applications (FPL), pp. 81-813, 2018.

[9] ISKANDER Y., PATTERSON C., CRAVEN S., *High-level abstractions and modular debugging for FPGA design validation*, ACM Transactions on Reconfigurable Technology and Systems (TRETS) **7**(1), pp. 1-22, 2014.

[10] MA R., HSU A., Tan T., NURVITADHI E., SHEFFIELD D., DASU A., PELT R., LANGHAMMER M., SIM J., CHIOU D., *Specializing FGPU for Persistent Deep Learning*, Proc. of 29th International Conference on Field Programmable Logic and Applications (FPL), pp. 326-333, 2019.

[11] NORONHA D.H., ZHAO R., GOEDERS J., LUK W., WILTON S., *On-chip FPGA debug instrumentation for Machine Learning Applications*, Proc. of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 110-115, 2019.

[12] NORONHA D.H., ZHAO R., QUE Z., GOEDERS J., LUK W., WILTON S., *An Overlay for Rapid FPGA Debug of Machine Learning Applications*, Proc. of 2019 International Conference on Field-Programmable Technology (ICFPT), pp. 135-143, 2019.

[13] DINU A., OGRUTAN P. L., *Opportunities of using artificial intelligence in hardware verification*, Proc. of IEEE 25th International Symposium for Design and Technology in Electronic Packaging (SIITME), pp. 224-227, October 23-26, 2019, doi: 10.1109/SIITME47687.2019.8990751.

[14] XILINX, *Spartan-3E FPGA Starter Kit Board User Guide*, [Online], Available: https://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf (accessed July,2020).

[15] CHEUSHEVA S., *Random sort in Excel: shuffle cells, rows and columns*, [Online]. Available: https://www.ablebits.com/office-addins-blog/2018/01/24/excel-randomize-list-random-sort/ (accessed August 16, 2020).

[16] GUO G., WANG H., BELL D., BI Y., GREER K., *KNN model-based approach in classification*, in OTM Confederated International Conferences "On the Move to Meaningful Internet Systems", pp. 986-996, 2003: Springer.

[17] EVGENIOU T., PONTIL M., *Support vector machines: Theory and applications*, in Advanced Course on Artificial Intelligence, pp. 249-257, 1999: Springer.

[18] LIAW A., WIENER M., *Classification and regression by randomForest*, R news **2**(3), pp. 18-22, 2002.

[19] ALLISON P. D., *Logistic regression using SAS: Theory and application*, SAS institute, 2012.

[20] PEDREGOSA F, *et al., Scikit-learn: Machine learning in Python*, Journal of Machine Learning Research **12**, pp. 2825-2830, 2011.

[21] MCKINNEY W., *pandas: a foundational Python Library for Data Analysis and Statistics*, Python for High Performance and Scientific Computing **14**(9), 2011.

[22] MCKINNEY W., *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*, O'Reilly Media, Inc., 2012.

[23] ABADI M., *et al., Tensorflow: A system for large-scale machine learning*, Proc. of 12th USENIX Symposium on Sperating Systems Design and Implementation (OSDI '16), pp. 265-283, 2016.

[24] DINU A., DANCIU G. M., OGRUTAN P. L., *Efficient analysis of digital systems' supplied data*, Proc. of 2020 International Symposium on Electronics and Telecommunications (ISETC),pp. 1-4, 2020: IEEE.

[25] ZHANG L., TAN J., HAN D., ZHU H., *From machine learning to deep learning: progress in machine intelligence for rational drug discovery*, Drug discovery today **22**(11), pp. 1680-1685, 2017.

[26] BROWNLEE J., *Understand the Impact of Learning Rate on Neural Network Performance*, [Online]. Available: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/ (accessed December 21, 2020).

[27] MA W., LU J., *An equivalence of fully connected layer and convolutional layer*, arXiv preprint arXiv:1712.01252, 2017.

[28] ZHANG Z., SABUNCU M., *Generalized cross entropy loss for training deep neural networks with noisy labels*, Advances in neural information processing systems **31**, pp. 8778-8788, 2018.

[29] WILLIAMS A. M., LIU Y., REGNER K. R., JOTTERAND F., LIU P., LIANG M., *Artificial intelligence, physiological genomics, and precision medicine*, Physiological genomics **50**(4), pp. 237-243, 2018.

[30] TRUONG A., HELLSTROM D., DUQUE H., VIKLUND L., *Clustering and Classification of UVM Test Failures Using Machine Learning Techniques*, Proc. of the Design and Verification Conference and Exhibition Europe (DVCon Europe), Mnchen, Germany, 2018.

[31] BROWNLEE J., *A Gentle Introduction to Dropout for Regularizing Deep Neural Networks*, [Online]. Available: https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/ (accessed January 5, 2021).

[32] DINU A., DANCIU G. M., OGRUTAN P. L., *Debug FPGA projects using machine learning*, Proc. of 2020 International Semiconductor Conference (CAS). IEEE, pp. 173-176, 2020.

[33] GOODFELLOW I., BENGIO Y., COURVILLE A., *Deep learning (no. 2)*, MIT Press, Cambridge, MA, USA, 2016, ISBN: 978-0262035613.

[34] BROWNLEE J., *A Gentle Introduction to the Rectified Linear Unit (ReLU)*, [Online]. Available: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ (accessed January 1, 2021).

[35] BROWNLEE J., *Loss and Loss Functions for Training Deep Learning Neural Networks*, [Online]. Available: https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/ (accessed January 5, 2021).

[36] DINU, A., CRACIUN A., ALEXANDRU M., *Hardware reconfiguration of a SOC*, Review of the Air Force Academy **1**, pp. 55-64, 20.