BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FAULT TOLERANT FIELD PROGRAMMABLE NEURAL NETWORKS

FIELD PROGRAMMABLE NEURAL NETWORKS ODOLNÁ PROTI PORUCHÁM

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR — Ing. MARTIN KRČMA
AUTOR PRÁCE

SUPERVISOR — doc. Ing. VLADIMÍR DRÁBEK, CSc.
ŠKOLITEL

BRNO 2022

Abstract

This thesis focuses on the Field Programmable Neural Networks concept intended to make implementation of neural networks in FPGAs less resource demanding. The thesis introduces and discusses several types of Field Programmable Neural Networks which provide different trad-offs between the resource consumption and the accuracy of the implemented neural network approximation. This thesis also introduces and discusses methods of hardening the Field Programmable Neural Networks against faults with and without redundancy.

Abstrakt

Tato práce se zaměřuje na koncept Field Programmable Neural Networks jehož cílem je učinit implementaci umělých neuronových sítí v hradlových polích méně náročnou jejich prostředky. Za tímto účelem práce konept rozvíjí a představuje několik jeho různých typů jež se vyznačují různými poměry mezi spotřebou zdrojů hradlových polí a přesností s jakou aproximují původní neuronovou síť jíž implementují. Teze díle rozšiřuje koncept o metody zabezpečení proti poruchám s využitím redundance a také bez ní.

Keywords

Field Programmable Neural Networks, fault tolerance, neural networks, FPGAs

Klíčová slova

Field Programmable Neural Networks, odolnost proti poruchám, neuronové sítě, FPGA

Reference

KRČMA, Martin. *Fault tolerant Field Programmable Neural Networks*. Brno, 2022. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Vladimír Drábek, CSc.

Fault tolerant Field Programmable Neural Networks

Declaration

I hereby declare that this Ph.D. thesis was prepared as an original work by the author under the supervision of Mr. doc. Ing. Zdeněk Kotásek CSc. and Mr. doc. Ing. Vladimír Drábek CSc. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

....
Martin Krčma
August 31, 2022

Acknowledgements

I would like to thank Mr. doc. Ing. Zdeněk Kotásek CSc. for his leadership and many years of professional support. Moreover, I would like to thank him for the wisdom and the life experiences he shared with me. I would also like to thank Mr. doc. Ing. Vladimír Drábek CSc. for his kind willingness to step in and take the role of my supervisor during a difficult time and to provide me the support I needed to finish this thesis.

Contents

1 Introcution **3**

2 Neural networks **4**
2.1 Neurons . . . 4
2.2 Activation functions . . . 4
2.3 Threshold . . . 5
2.4 Basic topology . . . 6
2.5 Learning of neural networks . . . 7
2.5.1 Supervised learning . . . 8
2.5.2 Unsupervised learning . . . 8
2.5.3 Backpropagation . . . 8

3 Fault tolerance of neural networks **11**
3.1 Possible faults in neural networks . . . 11
3.2 Hardening neural networks using learning . . . 14
3.2.1 Methods based on faults injection . . . 14
3.2.2 Methods based on restricting the weights . . . 16
3.2.3 Methods based on activation and basis functions modifications . . . 17
3.2.4 Methods using relearning . . . 18
3.2.5 Other learning-based methods . . . 19
3.3 Methods based on redundancy . . . 19
3.3.1 Triple modular redundancy . . . 19
3.3.2 Inserting new neurons . . . 21
3.3.3 Temporal redundancy . . . 21

4 Field Programmable Neural Arrays **23**
4.1 Field Programmable Neural Network . . . 24
4.1.1 The computation . . . 25

5 Research progress **31**
5.1 Approximation capabilities . . . 32
5.1.1 FPNNs with a single operator per link . . . 33
5.1.2 Reduced and Full FPNNs . . . 35
5.2 Fault tolerance . . . 36
5.2.1 Identity operators and mapping . . . 36
5.2.2 Triple modular redundancy . . . 38
5.2.3 Detecting hard synapses fault . . . 39
5.2.4 The FPNNs robustness . . . 40

5.2.5 Recovery using partial dynamic reconfiguration 42
5.3 List of Publications Related to the Thesis 43
5.3.1 Author's contributions to papers related to The Thesis 45
5.4 List of Other Publications, unrelated to the Thesis 45
5.5 Research Projects and Grants 47

6 Conclusions 49
6.1 Contributions 51
6.2 Possibilities of Future Research 52

Bibliography 53

A Mapping trained neural networks to FPNNs 62

B Comparison of FPNNs models approximation capabilities and FPGA resources utilization 67

C Comparison of FPNNs Approximation Capabilities 76

D Detecting hard synapses faults in artificial neural networks 79

E Fault tolerant Field Programmable Neural Networks 86

F Triple modular redundancy used in field programmable neural networks 91

G Fault tolerant Field Programmable Neural Networks 98

H Implementation of fault tolerant techniques into FPNNs 105

I Fault tolerance of different Field Programmable Neural Networks types 109

Chapter 1

Introcution

It was the year 1943 when Warren McCulloch and Walter Pitts introduced the first mathematical description of a neuron in their *A Logical Calculus of Ideas Immanent in Nervous Activity* [40] paper. Their neuron was behaving as a logic switch, and they proved that an interconnected network composed of such neurons is able to calculate any operation of propositional logic. Donald Hebb followed their ideas and introduced the first learning algorithms for such neural networks in his book *The Organization of Behavior* [24] in 1949. Eight years later, Frank Rosenblat introduced *perceptron* [64], a generalized model of a neuron that worked with real numbers. He developed a learning algorithm for neural networks based on his model. The algorithm was able to calculate the desired configuration of the network in finite time and independently from the initial state of the network. With this algorithm in hand, Rosenblat constructed the very first neuro-computer, which he named *Mark Perceptron I*. The computer was able to recognize characters and its successful presentation attracted first serious attention and interest in neural networks.

The beginning of the seventies came with the first model of a *binary associative neural network* developed by Karl Steinbuch [70]. At the end of that decade, Marvin Minsky and Seymour Papert pointed out in their *Perceptrons* [43] book that the logical exclusive disjunction operation is impossible with only a single perceptron. The authors admitted that it was possible to realize the said operation with a network of three perceptrons organized into two layers. Unfortunately, no known algorithm could guide such a network to learn the operation at that time. From that, they incorrectly concluded that no such algorithm could exist. This unfortunate conclusion, together with a lack of new fresh ideas, led to a significant drop in interest in neural networks and caused cuts in funding for the research. Despite that, the research quietly continued and got the attention back when John Hopfield presented a new model of an associative neural network that worked as a memory in 1982 [25]. The same year brought another important model of neural networks - the *Kohonen's* networks [32].

David Rumelhart, Geo Rey Hinton, and James McClelland published one of the most used and essential learning algorithms - the *backpropagation* algorithm [65]. The algorithm was based on the iterative improvement of the network based on propagating the value of the network output error back through the network while modifying its weights. The first significant conference focused solely on neural networks, the *IEEE International Conference on Neural Networks*, was held in San Diego in 1987 and the neural networks have remained in the academic, research and software engineering communities' interest ever since.

Chapter 2

Neural networks

Neural networks generally are abstract mathematical structures inspired by the human brain even though artificial neural networks are massively simplified and more specifically focused compared to their original archetype. Just like the brain, artificial neural networks are composed of neurons. Similar to their biological counterparts, artificial neurons are interconnected by synaptic interconnections or for short, synapses. Synapses represent channels through which information flows between neurons while being modified by the synapses' parameters. Those parameters are generally called *weights*, and they represent the strength of the connection between neurons. It is the values of these weights that hold the knowledge that the particular neural network gained during its learning process.

2.1 Neurons

If we refer to Equation 2.1 representing a general model of an artificial neuron we can see that the neuron n has $x_1, ..., x_n$ inputs representing all the incoming synaptic interconnections equipped with weights $w_1, ..., w_n$ accordingly. The data modified by the weights then enter the function f which computes the output value y of the neuron that would be send through outgoing synapses to connected neurons (2.2). This function is called an *activation function*. The input of the activation function is generated by a function called *basis function* which transforms the set of input data $x_1, ..., x_n$ and corresponding weights $w_1, ..., w_n$ into a value called the neuron's *potential* or *net*. This value then serves as the input to the activation function. One of the most common basis functions is a weighted sum (2.1).

$$potential = net = \sum_{i=1}^{n} x_i w_i \tag{2.1}$$

$$y = f(net) \tag{2.2}$$

2.2 Activation functions

In Section 2.1 we said it is the *activation function* of a neuron that generates its output which then serves as an input to other neurons or as the output data of the neural network itself. The activation function is, therefore, a core element of a neuron. Different types

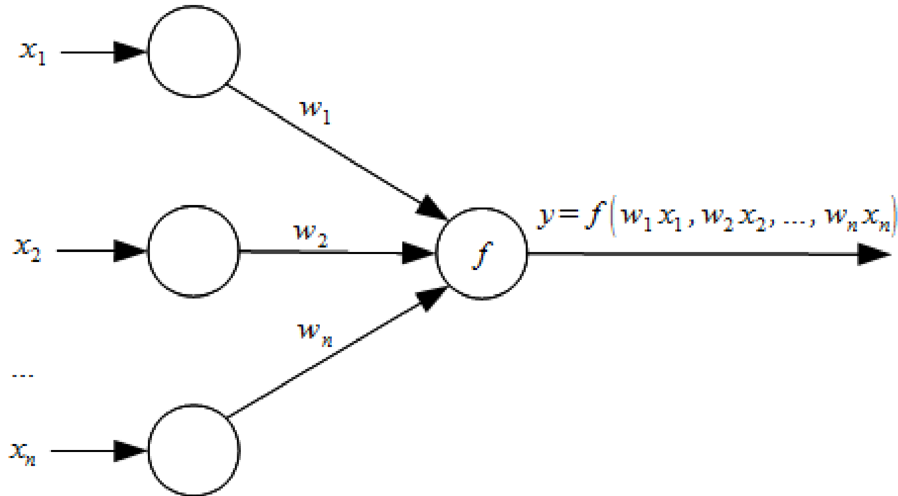

Figure 2.1: General inside structure of a neuron

and models of neural networks utilize different activation functions or even combinations of different functions. An activation function is often an increasing continuous and differentiable function. Discontinuous functions can also be used, but their downside is that neural networks utilizing such a function cannot be learned with a learning algorithm based on differentials such as the *Backpropagation algorithm* described in section 2.5.3. The the most frequently used activation functions are for instance the sigmoid function (2.3), the uni-polar step function (2.5), bi-polar step function (2.6), hyperbolic tangent (2.4) which are defined as follows:

$$f(x) = \text{sigmoid}(x) = \frac{1}{1+\epsilon^{-\theta x}} \tag{2.3}$$

$$f(x) = \tanh(x) = \frac{2}{1+\epsilon^{-2x}} - 1 \tag{2.4}$$

$$f(x) = \text{unipolar_step}(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases} \tag{2.5}$$

$$f(x) = \text{bipolar_step}(x) = \begin{cases} -1 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases} \tag{2.6}$$

Fig. 2.2 illustrates the graphs of the said functions.

2.3 Threshold

The threshold is a way to improve neural network capabilities by affecting a particular neuron's activation function independently. Originally the *term* threshold referred to an actual threshold that guarded the neuron output. It was a minimum limit that the neuron potential had to reach for the neuron to „fire,“ i.e., to produce an output. The neuron output would remain null if the potential did not cross this threshold. Even though this specific behavior can be achieved with a threshold as it is used in artificial neural networks, it is another variable that enters the computation of the neuron's potential rather than a limit imposed on it.

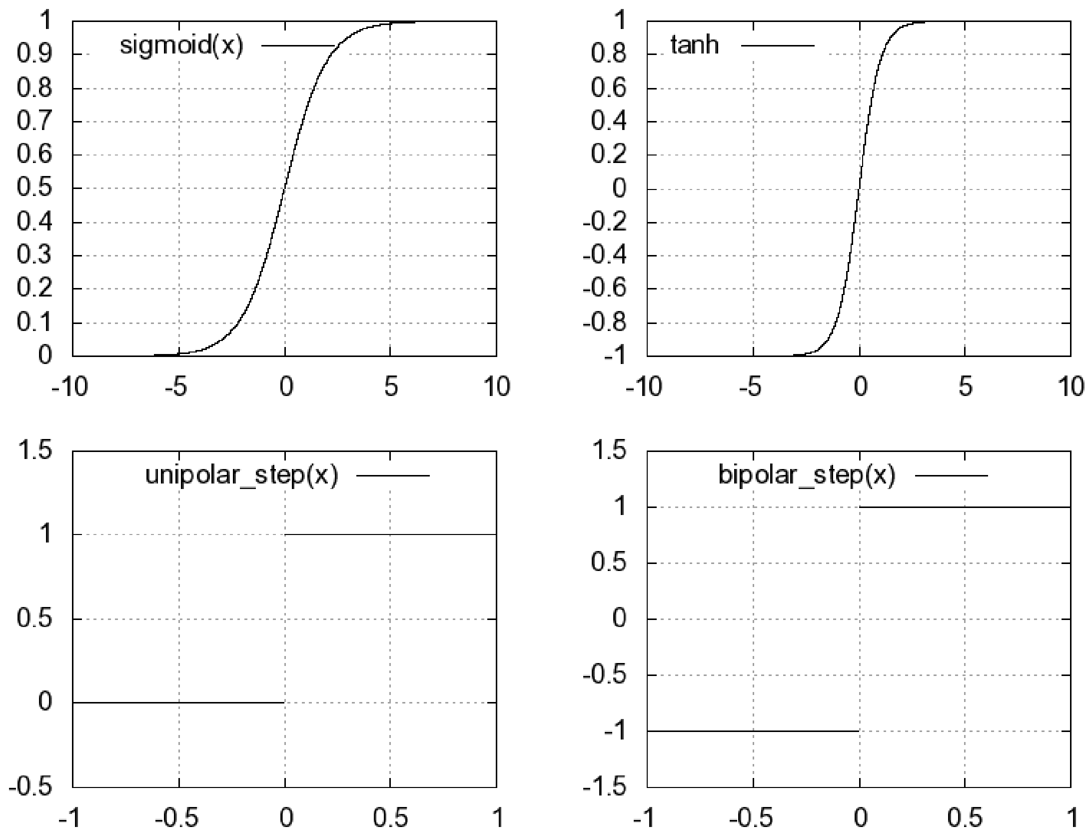

Figure 2.2: Activation functions

$$y = f(net + \theta) \quad (2.7)$$

Mathematically, a threshold θ is a value that is added to the neuron's potential as shown in equation (2.7). It is figuratively a hidden weight with a constant input data of the value of one. The value of this virtual weight then enters the weighted sum of the potential as another element. By adding the threshold to the potential, every input of the activation function is shifted on the x axis by the same offset. The activation function of the particular neuron itself then appears to be shifted permanently on x axis. Furthermore, since the threshold is effectively just another weight, it can be determined together with the rest of the weights during the learning process.

2.4 Basic topology

When Marvin Minsky and Seymour Papert derived that the logical function of exclusive disjunction can be realized by a network of three perceptrons, they also implied the most prominent building blocks of neural network architectures. That is, layers. Just like their three perceptrons network was composed of two successive layers, the following research also worked with neural networks like structures organized into layers. A neural network can be composed of a single layer, but usually, it is more. When a neural network has more

than one layer, then we call the first layer the *input layer* and the last layer the *output layer*. If the network has more layers than these two, then we speak of *hidden layers*. The number of neurons in each layer can differ from one another. The layers can have only one dimension in the form of a row of neurons, but any higher number of dimensions is possible. The neural networks working with image data usually have two-dimensional layers of neurons, for instance. A basic illustration of a one-dimensional neural network with one hidden layer can be seen in Fig. 2.3.

Another topological property of neural networks is the direction of the information flow. In other words, whether a feedback loop exists somewhere in the network. Usually, neural networks are feed-forward networks with information flowing from the input layer through hidden layers to the output layer and the outside world. It is also possible, however, to take a part of the output data vector and feed it back to the input layer. The Hopfield networks are an example of this topology. Similar feedback loops can exist between hidden layers as well. We can call the network with this property *recurrent* networks as opposed to *feed forward* networks.

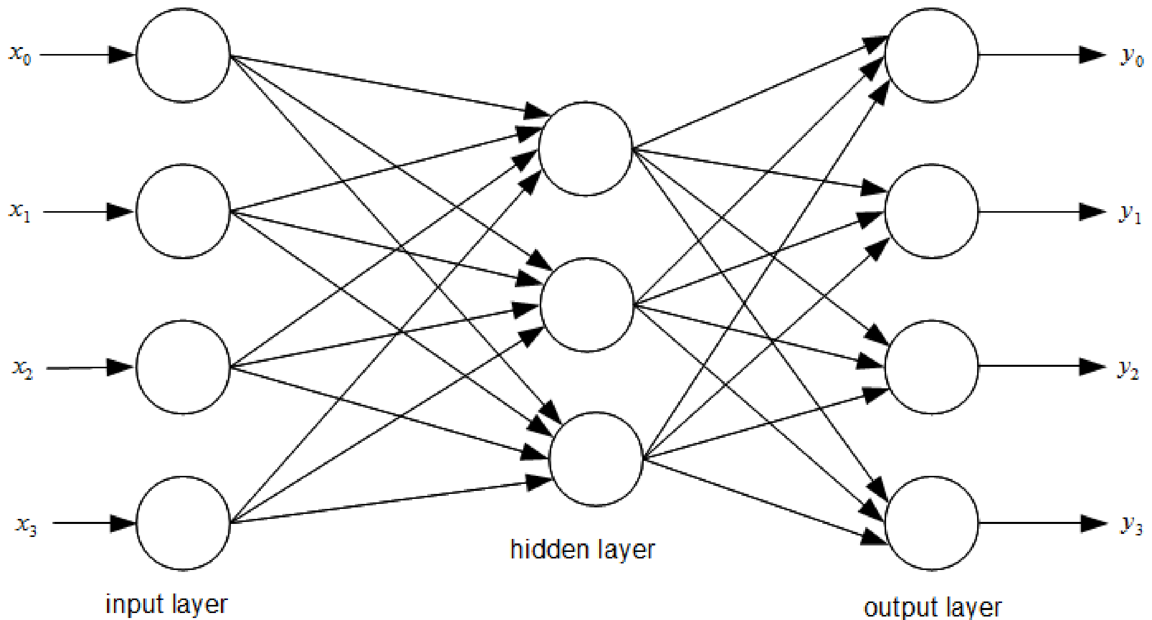

Figure 2.3: Backpropagation model

2.5 Learning of neural networks

At the core of learning, algorithms lay the basic principle of how neural networks store their knowledge. It is the set of the particular network weights that stores that knowledge. Similar to the human brain, the weights represent the strength of connections between particular neurons and, therefore, the way how the information flow through different parts of the network. Just like the human brain „wires“ itself to create neural paths that realize all our memories and skills, the artificial neural networks also basically realize their capabilities using the fact that the weights implement the way how the particular network is effectively wired.

Therefore, when we speak about a network learning to perform a specific task, we practically speak about setting the network's weights. The learning process sets the weights to make the information flow from the network's input through the neurons, transforming the input data to produce the desired results.

Bearing this principle in mind, we can say that the learning process creates a function or a map that connects desired output vectors to the inputs. This map is not a strict one, however. The relations between inputs and outputs that neural networks gains during their learning phase are not necessarily one-to-one. The relations are soft and indirect. Furthermore, in that fact lies the true strength of artificial neural networks. They are able to discover and generalize the relations between data vectors during learning and then apply this generalized knowledge to new and previously unseen input data vectors.

Moreover, if the network's knowledge is robust and general enough, it will be able to infer the correct relation and present a correct output data vector. To put this into a practical example, a neural network that has learned to recognize images of cars using a set of pre-prepared images. If the learning is sufficient, then the network will correctly recognize even an image of a car it has never seen before.

The learning process is then a process of determining the values of weights. The network usually starts with a randomized set of weights. This set is continuously modified in the learning process to bring the network to the desired operation step by step. Many algorithms can take the network through this process, and we generally recognize basic types - *supervised learning* and *unsupervised learning*.

2.5.1 Supervised learning

Supervised learning is based on an idea of a supervisor - a figurative entity that has knowledge of the task it wants the neural network to learn to do. Practically that means that the input data set intended to be used in the learning process has its desired-output data set counterpart. There is a known desired output data vector for every pre-prepared input data vector. The supervisor can then judge the output the network produces in response to the input data vectors and use this judgment in the consequent modifications to the network weights set to lead to the desired behavior. The backpropagation described in subsection 2.5.3 algorithm is a typical example of this type of learning method.

2.5.2 Unsupervised learning

Unsupervised learning is, as the name suggests, a type of learning process that does not use a supervisor. That means it does not rely on pre-determined knowledge of how the network should ideally operate. Instead, it relies on the network to learn in a way that makes it discover the correct output by itself. In other words, unsupervised learning leads the network to discover patterns and features in the presented data and generalize the discovered relations. Deep learning is a typical example of unsupervised learning.

2.5.3 Backpropagation

The backpropagation algorithm was published by David Rumelhart, Geo Rey Hinton, and James McClelland [65] in 1986. It is a gradient descent-based learning algorithm for a feed-forward neural network with multiple layers with all neurons in a layer, except the input layer, connected to all neurons in the previous layer. The layers are fully interconnected. It is a supervised learning algorithm based on presenting pre-prepared input data vectors

to the network, comparing the produced output vectors to corresponding correct output vectors, and calculating the difference. This difference is what drives the modification of the weights in the subsequent phase when the difference is propagated back to the network, from the output layer through the hidden layers back to the input layer. This is where the algorithm got its name. Because many methods discussed in the further parts of this work reference this algorithm, it is worth exploring it a little deeper.

The algorithm

In this method, the supervisor feeds j number of input vectors $x_1, ..., x_j$. For every vector x_n the network calculates the output vector y_n. The supervisor calculates the difference between both vectors from the calculated vector and the expected and desired value t_n. When all the input vectors are presented to the network and all the differences collected, the supervisor calculates the sum of all the differences squared to determine the value of the *error function* E, as described by Equation (2.8):

$$E = \frac{1}{2} \sum_j (t_j - y_j)^2 \tag{2.8}$$

The algorithm's goal is to gradually reduce the error function's value and evolve the network closer to the desired configuration. Because it is the non-optimal value of the weights that cause the network to calculate differently than desired, we can clearly say that the error function is the function of the weights. In order to bring the network where the supervisor wants it to be, it is necessary to find the minimum of the error function. That can be done using simulated annealing or, more commonly, using gradient descent methods. Using gradient descend [45] we can derive the equation (2.9) for calculating the necessary weights modification in hidden layers and in the output layer:

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} + \Delta w_{ij}^{(n)} \tag{2.9}$$

Where:

$$\Delta w_{ij}^{(n)} = \eta \delta_j o_i + \alpha \Delta w_{ij}^{(n-1)} \tag{2.10}$$

Equation (2.9) describes modification of the wights by adding a difference value calculated using equation (2.10). The η and α values in the equation are positive real numbers that can be set at will as there is no preset way to calculate them. These values influence the algorithm's performance, and it is possible to modify them on the run to achieve better and quicker convergence.

If the activation function is the sigmoid function (2.3) and the network utilizes thresholds, using the function differential [45] we can define the δ_j for the output layer:

$$\delta_j = (t_j - o_j)\, o_j\, (1 - o_j) \tag{2.11}$$

Where o_j is the output value of the neuron j and t_j is its desired value - the particular element in the desired output data vector. Similarly, we can calculate the δ_j for the hidden layers:

$$\delta_j = o_j\, (1 - o_j) \sum_k \delta_k w_{jk} \tag{2.12}$$

Where k refers to the particular neuron in the output layer which back-propagated its δ_k value calculated using the equation (2.11) to the neuron j in the hidden layer.

$\sum_k$ is a sum of all the differences sent back from the output layer and multiplied by the corresponding weights.

These deltas then propagate back to the preceding hidden layer, which calculates their on deltas using the same equation (2.12).

Chapter 3

Fault tolerance of neural networks

The fact that neural networks are distributed, redundant, and relatively homogeneous computational models had the researcher think about their fault tolerance. By nature, neural networks seem to have the potential to be inherently robust. A system composed of many interconnected components that are similar to each other and compute simultaneously might be able to tolerate a fault in one or even several of them. However, the layered structure of neural networks makes neurons dependent on the computation of all the neurons that proceed them in the previous layers. Should any of them produce erroneous results, the error might propagate and escalate in the following layers. The question of how robust the neural network can really be and whether this robustness can be improved has attracted research efforts throughout the years.

3.1 Possible faults in neural networks

When we want to evaluate possible faults in neural networks, we must consider different ways of viewing them because different faults can occur on different levels of abstraction and can be recovered using different methods. The natural way is to approach a neural network as a whole entity, a computation component. On this general point of view, there are two basic types of errors that can occur in a network's operation:

- Computation error - the network generates erroneous outputs
- Communication error - the network does not communicate properly with the rest of the system

A problem or a fault within the network would most probably cause the first type of error. The second type may be caused either by a fault inside the network or by an incorrect design of the network, its implementation, communication protocol, or implementation incompatibility with the rest of the system. A synchronization discordance and incompatibility of communication interfaces or protocols would be the most obvious suspects.

When we look closer at the composition of a neural network, we see that several types of faults can occur in the computation components that make the network. From the general point of view, a neural network is composed of neurons interconnected with synapses. Both can be seen as individual components, and both can suffer from different kinds of faults:

- The component's output is permanently stuck on a constant value. This type of fault can also represent a missing component, considering its output is stuck at zero.

- The component's output is erroneous. The output is affected by a value Δ. The error can be additive (the Δ is added to the output value) or multiplicative (the output value is multiplied by Δ).

Based on these possibilities, we can distinguish a set of possible faults regarding particular components:

Loss of a neuron By a loss of a neuron, we distinguish a situation when a neuron is permanently removed from the network or its output value is stuck at zero. The neuron does not participate in the following neurons' computation in both cases.

Loss of a synapse By losing a synapse, we distinguish a situation when a synapse is permanently removed from the network or its output value is stuck at zero. This can also be caused by an error in the corresponding weight that is stuck at a zero value. Another cause might be a problem with the corresponding neuron input.

Saturated neuron By a saturated neuron, we distinguish a situation when a neuron output is permanently stuck on one of the extrema values of the activation function codomain, the range of all its possible values. In the case of bipolar sigmoid activation function, the saturated neuron would have the output value 1 or -1.

Saturated synapse By a saturated synapse, we distinguish a situation when a synapse output is permanently stuck on one of the extrema values of the particular data type.

Stuck neuron By a stuck neuron, we distinguish a situation when a neuron output is permanently stuck on a constant value c. This problem can also be caused by an error in the neuron arithmetic unit, the following registers or memory cells, or the output bus.

Stuck synapse By a stuck synapse, we distinguish a situation when a synapse output is permanently stuck on a constant value d. This can also be caused by an error in the corresponding weight that is stuck at the value d. Another cause might be a problem with the corresponding neuron input.

Transparent synapse By transparent synapse, we distinguish a situation when a synapse becomes transparent to the data passing through it. It does not perform the computation it is supposed to but its output is not stuck, it let the data coming to its input to pass to its output. Effectively, this means that the data are not multiplied by the particular weight as they are supposed to. This can be caused by bypassing the computation components. The reason to bypass them might be to recover the network from a fault detected in the computation components.

Noisy neuron By a noisy neuron, we distinguish a situation when a neuron output is affected by a value Δ. This situation can be caused by a transient error in the memory that participates in the neuron computation.

Noisy synapse By a noisy synapse, we distinguish a situation when a synapse output is affected by a value Δ. This situation can be caused by a transient error in the memory that holds the corresponding weight. Another cause might be a noise in the corresponding neuron input.

Restricted neuron and synapse By a restricted neuron or synapse, we distinguish a situation when the output value of the neuron or the synapse is artificially limited in a particular range that is smaller than the activation function codomain (the range of all its possible values) or the used datatype. This situation can be caused by a permanent error in one of the memory cells that hold a particular weight or a result of the neuron computation.

For several reasons, it can be helpful to distinguish these fault types on the level of the components that make the neural network. The first reason is that we can select a particular type of fault, or a group of types, and harden the affected component against them while disregarding others to save resources or simplify the implementation. Moreover, we can also choose to harden only a particular selected set of components that we consider essential. We can also use an enhanced learning algorithm to harden the selected components to harden the network during the learning process. In this context, we can refer to two groups of components:

Critical neuron or synapse By a critical neuron or a synapse, we understand a component in which erroneous computation would have a significant impact on the quality of the results of the entire neural network. A fault in a critical component could cause the inability of the network to provide correct results.

Non-critical neuron or synapse By a non-critical neuron or a synapse, we understand a component in which erroneous computation would not have a significant impact on the quality of the results of the entire neural network. Even with the fault in the non-critical component, the network would still primarily provide correct or close to correct results.

We can also take a more detailed look beyond the abstract component and focus on the structure and building blocks of neurons. Like other computation component, those that build up into neurons and neural networks are prone to possible low-level faults like Single Event Upset (SEU) and others. Given the functionality of neurons, we can expect the design to utilize some of the following components:

- Adders that are collecting potential.
- Multipliers for multiplying by weights.
- Adders, multipliers, and other arithmetic units for computing the activation function.
- Registers and memory cells involved with the computation components and transferring data.

When we take a look at these components, we can predict a possible impact of faults occurring in them:

- Fault in the components computing the activation function would generally lead to the activation function changing its shape and therefore into an erroneous computation and output results.

- A fault in the adder responsible for computing the neuron potential would cause erroneous data to enter the activation function making all the following computations incorrect. This component's permanent fault would shift the activation function on the x axis.
- A fault in the multiplier responsible for multiplying the input data with corresponding weights would also cause erroneous data to enter the activation function. It could potentially lead to saturation or generate noise.

We can consider several approaches based on different principles to counter the possible faults on different levels of neural network design or enhance the neural networks' inherent robustness granted by their naturally redundant structure. We can increase network redundancy by adding more neurons, duplicating them, creating backup neurons, duplicating entire layers, or deploying techniques based on the majority principle like the Triple Modular Redundancy technique. We can introduce redundancy into the underlying components or even communication buses.

Different methods can be based on utilizing the learning process to harden the network naturally. If we set up a goal to learn the network to be more fault-tolerant, we can do so by modifying the learning algorithm or the condition in which they operate. In this approach, we can also focus on particular types of faults and errors.

3.2 Hardening neural networks using learning

Methods of hardening neural networks using learning are based on one of the known learning algorithms that were modified and expanded in a way that during the learning process, the neural network would not only learn to operate in the desired way but also learn to be fault-tolerant. These methods take into account different scenarios of possible fault occurrence and are designed to mitigate the impact of those scenarios using learning.

3.2.1 Methods based on faults injection

The methods based on fault injections utilize the fact that in the learning process, the neural network would gradually converge to the desired state regardless of its initial state or whether its state is externally changed. Given that the learning algorithm and the data used for learning are robust enough, the neural network would converge even when faults are introduced into its configuration. These faults would influence the quality of the network's output vector, increasing the error function. The faults would then become other variables in the error function, and their impact would be gradually reduced in the consequent learning. The learning phase would then naturally harden the network.

Methods based on this principle would use different types of faults to inject. They can remove neurons or synapses from the network in order to simulate their failure. They can also inject errors into network data, usually to the weights. It was shown that *injecting faults* into neural network weights during the learning phase would not only harden the network against faults causing errors in the weights but it would also *improve the network generalization capabilities* [15]. The injected faults would figuratively disrupt the network's convergence to the desired state the learning algorithm leads it to. The learning would be slower and harder so it would naturally force the network to discover deeper patterns in the input data. Therefore it would force the network to learn better potentially.

The first possible method based on this principle is a *random faults injection* into neurons weights in the hidden layers, one neuron at a time [7, 68]. In every iteration of the learning algorithm, a single random faults would be injected into a different neuron. The network would therefore learn to tolerate a single fault occurring in one of its neurons. We can also inject faults into *more than one* neuron or more than one weight in each iteration in order to harden the neural network against faults in multiple neurons. The fault injection, however, negatively impacts the error function, and it is possible to prevent the network from converging to the desired state at all by injecting too many faults during the learning process and therefore making it to fail.

Another method utilizing fault injection can be based on accounting for all the possible faults in a selected neuron [6, 7]. This method also uses fault injection during each iteration of the backpropagation learning algorithm. Suppose we have pre-prepared P *input data vectors*, and the hidden layer (suppose a single hidden layer in this description) is *composed of N neurons*. Then during each iteration of the learning algorithm, every input data vector is introduced to the network N times, with a fault being injected into a different neuron every time. The errors the fault injection caused in the output data are measured and a set of N errors are determined for every input vector, one for every neuron. The errors are then summarized into a single error value and propagated back through the network in the learning step of the backpropagation algorithm. Errors caused by faults in every particular neuron are part of the backpropagation algorithm's error function. By getting the error values for all neurons and including them in the backpropagation learning step, the network naturally learns to tolerate faults in all neurons in the hidden layer.

This method can be modified to teach the network to tolerate faults in more than one neuron. We can present each of the input data vectors N^m times and inject N^m combinations of faults when m is the number of faulty neurons we want the network to be able to tolerate. However, the number of injected faults can increase rapidly with rising m. As we mentioned earlier, it is possible to slow down the network convergence by injecting too many faults or events to prevent it from being able to learn at all.

These methods can be expanded to work with a neural network with more than one hidden layer. The methods must be applied to all the hidden layers in the described way.. It is also possible to select a subset of layers or neurons to be hardened while leaving the network vulnerable to faults in the others.

Another method of injecting multiple faults into a hidden layer is based on injecting an entire vector of faults. [28] In every iteration of the learning algorithm, a vector composed of n faults with n being equal to or lower than the number of neurons in the particular layer. The number of fault vectors is also n, and the $x-th$ element of the $x-th$ fault vector has a fixed value. Every vector, therefore, has one fixed value on the position corresponding to the vector index in the dataset. The rest of the element values are chosen randomly. By having a fixed value in each vector on the different indexes, the method assures that every neuron in the layer (or in the selected subset) is injected with a fault while also injecting random faults into the rest of the layer. While iterating over the set of the fault vectors, the method always injects at least one fault and a random number of random additional faults. The random generation of the additional faults is supposed to give the network a better chance to converge with its learning process because the authors presume that the injection of a fixed set of pre-selected have a chance to disrupt the network convergence.

Another approach to include hardening into the backpropagation learning algorithm is to construct a set of faults we want the network to harden against specifically and then use a modified backpropagation algorithm [80] utilizing a modified error function E. The

modified algorithm would evaluate the impact of all the faults in the set on the overall value of the error function ahead of learning. Then it would incorporate the error value caused by the faults into the error function during the learning process.

It is also possible to go beyond just a fault injection. Method published in [10] is based on fault injection and modification of the network structure. In each iteration of this algorithm, a random set of a fixed number of neurons is selected. Half of the selected neurons are then figuratively removed from the network by setting their outputs to zero, while the rest have their outputs artificially saturated. After that, a small set of synapses is randomly selected, and their weights are injected with random faults by adding a random number from a $\langle -1, 1 \rangle$ interval to the weight value. The network then learns to tolerate multiple types of faults - lost neurons, saturated neurons, and lost or noisy synapses.

In order to harden a network against faulty synapses, the authors suggested proceeding to modifications of the network structure. After the network learned its task, each neuron's impact on the network performance was measured by removing it from the network, followed by testing the network. The testing data vectors were introduced to the network's input, and the difference between the new output data vectors and the initial data vectors was measured. Suppose removal of the neuron proves to cause a less significant difference in the output data (the authors accepted differences up to ten percent). In that case, the neuron is removed from the network. After the removal, the network goes through the learning process again. This process repeats until neurons can no longer be removed.

After the previous phase is finished and no more neurons can be removed from the network, the network is further hardened by successive replicating of the most critical neurons. The method determines the neurons whose removal had the most considerable impact on the output data quality. Then the neuron is replicated. The weights of the replicated neuron's input synapses are kept the same, and the weights of the output synapses of both the original neuron and its replica are halved. By halving the weights, the method assures the network returns to its original state even with the additional neuron. The reason is that the combined value both neurons provide to the neurons in the successive layer will sum up to the same potential. Halving the weights of both neurons also reduces their influence on the network performance, making a loss of either of them less critical. This process is repeated with other critical neurons until the replication no longer improves the network robustness.

3.2.2 Methods based on restricting the weights

These methods are based on the realization that the most fault-tolerant neural networks are those that have the most uniform set of weights, that have weights whose value do not significantly vary from one another. The more uniform the values of weights are, the more the possibility that some of them may be critical to a neural network's operation is reduced. However, most networks do not have this property because the learning process usually produces networks with a nonuniform set of weights. The network often has some weights that are critical to its function, usually with high values. A fault injected into these weights could significantly impact the network performance. Many of the other weights are less critical to the network operation. If the weights of the network could be made uniform, then a fault in any of them would have a more negligible impact on the network performance. This can be achieved by artificially restricting the values of the weights.

One of the methods how for reducing the number of critical weights is based on continuous evaluation of their impact on the output data. Those weights that prove to have a

large impact are then reduced in value, so their influence would also reduce. If the values of the weights are continuously restricted, the learning algorithm is forced to produce a network with more uniform weights. However, limiting the time interval during which this method is applied might be necessary because it might slow the learning process down or event prevent the network from converging to the desired state at all. On the other hand, this method can produce a network with higher generalization capabilities [20].

Another option to prevent the learning algorithm from generating weight with high values is to force it to minimize them during learning [10, 78, 79]. For the algorithm to do that, the error function (2.8) needs to be modified. The summarized value of the output errors is expanded with a sum of all the weights values squared as illustrated in equation (3.1), where W are the set of all weights.

$$E = \left(\frac{1}{2} \sum_{j} (t_j - y_j)^2 \right) + \sum_{w \in W} w^2 \tag{3.1}$$

This modification means that the error function rises with the values of the network weights. The learning algorithm is then forced to minimize the network's weights as well together with the output error in order to minimize the error function. This method can be modified to only consider the weights of synapses connecting the hidden layer to the output layer, which might prove to be a more efficient approach to hardening the network [22, 23].

Besides modifying learning algorithms to make them harden the network, it is possible to approach the problem as a direct optimization problem. The need for the network to learn and the desire to harden it against fault can be seen as an optimization problem solvable with the Minimax method [12]. The algorithm's objective function that is supposed to be minimized represents the optimization problem. In this case, the objective function is the error function. The constraints set to the method are designed to make the algorithm as uniform a set of weights as possible and thus preventing the occurrence of critical weights. The Minimax algorithm is not the only optimization algorithm that can be used for hardening a neural network. The usage of quadratic programming [14], an optimization method, was also successfully demonstrated.

It was also shown that Hopfield neural networks [46] can be hardened by restricting the weights as well [29]. Hopfield networks are recursive networks containing a feedback loop that serves as an associative memory.

3.2.3 Methods based on activation and basis functions modifications

These methods are based on the idea that it is possible to harden the network using neurons functions manipulation - the basis function that calculates the neuron potential and the activation function that calculates the neuron output from its potential.

The first method is a method modifying the *basis function*. This method changes the way of calculating the potential by replacing the original summation in the basis function with calculating a *median* value instead. The neuron's inputs are still being multiplied by the respective weights, but the median value is computed instead of the weighted data being summarized. By applying the median, the method filters out the input data influenced by weights with very high or very low values. Faults might cause these extreme values, and their influence would be mitigated by filtering them from the potential computation. However, because the median is not a continuous function and therefore is not a fully differentiable

function, the backpropagation algorithm needs to be modified. The experiments in the original paper show that a neural network composed of neurons with a median basis function showed a 10% better classification results than the original network with a weighted sum basis function. The results also show that the method performs the best when only the neurons in the output layer have their basis function replaced by the median [67].

Another possible approach is based on manipulating the *activation function* [30, 81]. This method is based on modifying the slope of a sigmoid activation function of the neurons to make it steeper. The neural network does through the learning phase with activation functions of its neurons having a modest slope. The slope is gradually made steeper and steeper. At the end of the learning process, the slope is so steep that the activation functions effectively turn into step functions. With the activation function like that, an output of a neuron mainly falls into one of the extrema. Should some fault influence the input data of the neuron and, therefore, its potential, there would be a significant chance that it would not affect the neuron's output at all as the output would still fall into the same extrema. This method is, however, suitable only for classification tasks.

3.2.4 Methods using relearning

These methods are based on the idea that if some permanent fault should occur in a neural network, it can be mitigated by letting the network relearn. The advantage of this approach is that there is no need to modify the learning algorithm or the network structure. It is the rest of the network that is unaffected by the fault and is still functioning that is used to mitigate the fault by relearning how to perform the given task in the presence of the fault. However, the network learns and later operates with fewer resources because some were lost due to the fault. Therefore, the network might not be able to recover fully. Also, the fact that the network relies on relearning is another disadvantage because the learning process can be time-consuming, and the original data the network learned from must be accessible. Both temporal and spatial complexity of this method is high.

The most simple but the least practical approach to recovering the network using learning is just to start the process of learning after the fault is detected. However, inserting a new neuron into the network to take the role of the faulty neuron and then take the network through relearning might be more effective [68]. This approach can recover the network into its fully operational state; however free resources to insert the additional neuron into the network must be available.

In the case of neural networks implemented in hardware, more than errors in data might occur but also errors in timing and synchronization. These can be troubling because they might not be easily detectable. A method based on the backpropagation algorithm was proposed to solve this problem [11]. This method aims to determine and verify the working frequency of the device that makes the device work the most reliably. This method is based on simulating the computation of the particular neural network with different deviations from the estimated ideal working frequency. The results of the simulations are applied to the backpropagation algorithm that lets the network learn to work on different frequencies. The overall error values the algorithm achieved are monitored. For each iteration of the method, the set of weights and the corresponding frequency are saved, and the final setting of the network's weights and working frequency is based on which frequency achieved the smallest overall error. The selected frequency should be the optimal working frequency for the particular network and the device it is running on.

3.2.5 Other learning-based methods

The backpropagation algorithm is not the only gradient descend-based learning algorithm that can also be used to harden the network. A learning algorithm based on the gradient descent and the Kullback-Leibler divergence was suggested [73]. During each iteration, this algorithm includes vectors of Gaussian noise to inject faults into the network weights. During learning, the noise is incorporated into the weights and the networks become more resilient against faults with Gauss noise characteristic affecting the weights.

In the case of Hopfield neural networks implemented in software, a method hardening the network against errors in data of the learning algorithm due to a corrupted memory is in [39]. All-access to data arrays in the memory access was protected by modulo operation to ensure that the correct region of the memory was accessed [82]. The iteration counter in the core of the learning algorithm, which serves for convergence evaluation, was hardened against data corruption by calculating a logical disjunction of its value with the value of an additional auxiliary variable. The variable was initialized with a non-zero value that is not a multiple of two. The iteration counter was initialized with the same value. This ensures that one erroneous bit in the iteration counter would not change its value to zero, thus stopping the learning algorithm prematurely. Using the logical dis-junction with another variable ensured that the iteration counter value was overwritten each iteration with a correct value.

Another measure of hardening the network learning was based on connecting an identical neural network to the hardened network. The second network serves as a golden model producing a correct value in case a fault occurs in the original network. The key idea is that even if the first network is not able to converge, the second one still is. If the second network were faulty, then it would not affect the first one.

3.3 Methods based on redundancy

The neural networks are, by principle, massively parallel and redundant structures. This redundancy provides them with inherent fault tolerance. A neural network can withstand a fault of a neuron or a synapse, especially if hardening techniques were applied during the learning process or modifications of the network's properties were used to improve its robustness, as described in the previous sections. However, their robustness can potentially be improved by techniques based on adding more redundancies to the network.

3.3.1 Triple modular redundancy

The triple modular redundancy (TNR) is a classic method of hardening a system. This technique is based on triplicating the system or its subsystems and adding a voter that would evaluate the outputs of all three instances and vote for the correct one by the principle of majority. Even if one of the three instances were faulty, the system would still operate correctly.

This technique can be applied to neural networks as well. It can be applied to the entire network or its components on different levels. Of course, triplicating the whole network is an easy way to harden it without needing to modify the network's structure or doing any significant interference. However, it also triplicates the network spacial complexity, which can be already high in the case of most neural networks. The number of neurons and especially the number of weights grow quickly with the complexity of the implemented

task, and it might not be possible to expand it much more, let alone multiply it. Depending on the network implementation, triplicating may also increase the temporal complexity.

The TMR method can be applied to the level of individual neurons and synapses. The advantage is that we can individually choose what to harden and how. We can choose only a subset of neurons and synapses we deem critical. The method can also be used on the level of computing blocks themselves. We can triplicate the adders, multipliers, and other low-level blocks. We can even go as far as using this technique on the bits themselves. A method has been proposed for hardening a neural network using TMR while keeping the overhead as low as possible. The method is called Relaxed Triple Modular Redundancy (RTMR) [37] based on hardening the computational blocks and the interconnecting buses on the bit level but only on the level of the selected subset of high bits. In this approach, only the more minor part of the bits is hardened, and the rest of the design is still vulnerable. However, it is the least significant bits that are left unhardened. Naturally, the low bits have a lower influence on the overall values entering the computation. Its influence may be less critical if an error occurs in the lower bits. The advantage is that the spacial overhead consumed by replicated resources is significantly lower than in the case of the full TMR. It is at the expense of a trade-off between hardening and resource consumption, but it can be accommodated to the situation with the proper choice of hardened bits.

In the case of triplicating entire neural networks, voters with weighted inputs can also be used as demonstrated in [5, 35, 63]. The principle of this modification is that the voter at the output of the hardened system considers its inputs to have different priorities, and it uses those priorities given by their weights in the voting process. The input with a higher weight gets priority. For example, the weights can be determined using a backlog of faults that occurred in all three replicas. The weights can also be learned [84]. In this approach, the system is composed of three neural networks. However, the networks are not identical. Each network learns individually and independently from others in three different steps. In the first step, the networks learn to perform the desired task. In the second step, they learn again, but their input data were individually injected with faults. In the third step, they learn one more time and face simulated loss of some of their neurons. The weights for the voter are then determined by how well each network operated during the first steps. The more reliable and quality each network proved to be, the higher its weight gets.

In [54] the authors experimented with different replication orders. The root of the experiment was to create a set of different neural networks with different numbers in the hidden layers and let them learn to perform a classification task called Sonar [19]. In the next step, the networks were replicated in different orders. After that, their tolerance to loss of one of the neurons or synapses was evaluated. The authors came to two interesting conclusions. The first conclusion was that when the resulting neural network had the same size, it was the network that was created by a lower number of replications while using a larger starting network that proved more reliable. On the contrary, when the network was created by more extensive replication but from smaller original networks, it was less reliable. So, for example, a neural network composed of eight neurons in the hidden layer created by duplicating a network composed of four neurons was more reliable than a network created by four replications of a network with two neurons.

The second conclusion was that neural networks created by replicating smaller networks were more reliable than a network of that same size that was not created by replication. For example, a neural network composed of eight neurons in the hidden layer created by replicating the original four-neuron network was more reliable than a neural network constructed and learned with eight neurons without any replications.

Another successful use of TMR was demonstrated in [66]. The authors determined how critical each neuron was for the considered network computation and ordered them by the measured criticality. They selected a number of the most critical neurons and hardened them using TMR. They removed the same number of the least critical neurons to compensate for increased computation complexity. Their method managed successfully to harden the selected network against lost and noisy neurons.

3.3.2 Inserting new neurons

Neural networks can also be hardened by inserting new neurons into hidden layers. This approach comes with less overhead than replicating an entire network or its significant part. Naturally, the hardening effect is lower. As such, they are a middle ground between complete replication methods and replication-free methods based on relearning and modification of neural network configurations. These methods are always trade-offs between reliability and spatial complexity. It is also important to point out that these methods only apply to hidden layers because the respective data vector sizes give the input and output layers sizes.

The most basic technique is based on adding a single neuron into a hidden layer [3]. This technique not only hardens the network but also allows us to detect a fault in any of the neurons of the hardened layer. The technique works as follows. Suppose the neurons in the hardened layer are equipped with enough memory storage for two vectors of weights. One vector is for storing the operational weights, and the other is the testing vector for storing weights used for fault detection. The neurons are then tested in pairs. The weights of the first neuron in the pair are set to the values of the testing vector belonging to the second neuron in the pair. After that, both neurons receive the same input vector, and their output values are compared. If the values are different, then the network is possibly affected by a fault in one of the neurons in the pair. The second neuron then becomes the first in its own pair, and its subsequent neighbor in the layer becomes the second neuron in the pair. Then the test is repeated. Therefore, every neuron is tested twice (the last neuron in the layer will be paired with the newly inserted neuron). If a neuron suffers from a fault, it likely fails both tests and therefore is detected as faulty. The newly inserted neuron can take the faulty neuron role by setting its operation weights vector accordingly. Using this approach, we can build a neural network that is able to detect a single faulty neuron and recover from the fault. Naturally, every hidden layer can be hardened using this method.

Another approach [52, 53, 85] is based on multiple replications of all neurons. The neurons are replicated h times, then the thresholds of the neurons in the following layer are multiplied by the same value. This ensures the network computes the same way despite the replicated neurons while also being hardened against the faults in the replicated layer. If changing the thresholds does not suit the design for some reason, the same effect can also be achieved by dividing the weights of the synapses between the layers by the number h [16]. Both methods have the same result. The potentials of the neurons in the following layer will have the same values as before replication; therefore, the layer will compute the same way.

3.3.3 Temporal redundancy

All the techniques discussed before were based on redundancy and utilized spatial redundancy to harden neural networks. However, it is also possible to achieve the effects of redundancy by performing additional computations. The most trivial method is just to

compute the result several times in a row and then compare the results to detect an error or select the correct result based on the majority principle. This approach can also be utilized together with spatial redundancy. Suppose the network is hardened in a way that uses replication and majority principle, but the fault detection fails. In that case, it is possible to recompute the results and then retry the fault detection.

An approach utilizing both spatial and temporal redundancy was introduced in [26]. The arithmetic unit used for computing the weighted sum was divided into three smaller units. Its operands were also divided into three parts based on the significance of the bits. The computation was divided into three phases. In the first phase, the data part with the least significant bits was introduced into all three arithmetic units, and three semi-results were computed. The second and third parts of the original operands were introduced to the arithmetic units in the two consequent parts. Their results were combined according to the significance of the particular parts. Eventually, three independent full-operand results were produced using three independent arithmetic units. The results were compared, and the final result was selected using the majority principle.

Chapter 4

Field Programmable Neural Arrays

The concept of Field Programmable Neural Arrays (FPNAs) [17] is designed to enable a resource-efficient implementation of artificial neural networks in Field Programmable Gate Arrays by adjusting the network's properties and especially its structure in order to make them more efficiently implementable into the gate arrays. For instance, FPNAs were used for implementing large scale spiking networks [21]. The efficiency comes from the FPNA's main feature - a highly customizable structure that enables the designer to build it in a way that allows sharing the FPGA's resources by merging sets of synapses into several dedicated components. This concept also simplifies the interconnection structure compared to the original neural network. The number of neurons remains the same.

By the original definition by B. Girau [17], an FPNA is directed graph (N, E) where N is a set of nodes and E is a set of directed edges that connect the nodes:

Definition 4.0.1 (FPNA [17])**.** We say that structure (N, E) is an *FPNA* if the following statements hold true:

1. N is a set of nodes called *Activators*. Activators represent the original neural network neurons.
2. E is a set of directed edges called *Links*. Link connect activators.
3. Each activator n has a set of predecessors: $Pred(n) = \{p \in N, (p, n) \in E\}$
4. Each activator n has a set of successors: $Succ(n) = \{s \in N, (n, s) \in E\}$
5. There is a set of input nodes: $N_i = \{n \in N, Pred(n) = \emptyset\}\,; N_i \subset N$
6. Each link $(p, n) \in E$ has an *affine operator*: $\alpha_{(p,n)} = W_n(p)x + T_n(p)$
7. Each non-input activator $n \in N$ has an *iteration operator*: $i_n : \mathbb{R} \to \mathbb{R}$ to calculate its potential.
8. Each non-input activator $n \in N$ has an *function operator*: $f_n : \mathbb{R} \to \mathbb{R}$ to calculate the activation function.

Definition 4.0.1 states that the original neurons are represented by the nodes in the graph, the *activators*. The activators use their iteration operators i_n to calculate their potentials and then apply their function operators f_n to calculate the activation functions over the potentials and thus generate their outputs. Therefore, the activators principally closely mimic the function of neurons.

The activators are interconnected by edges, by *links*. The links calculate an affine transformation of their inputs using their affine operators α. By doing this, they approximate multiplying the data by the corresponding weights. Therefore they participate in calculating the weighted sum by taking this part of the computation out of activators. Moreover, each link can approximate multiple synapses for multiple activators. Therefore, the weighted data are calculated for each activator in parallel by a set of links that connect them to the preceding activators. The definition allows the activators to be connected by more than a single link. It is possible to create chains of links between activators or layers and, therefore, to split each synapse and the corresponding weight into a set of successive affine operators. Because this possibility exists, it is helpful to introduce a unifying term for both activators and links - *neural resources*.

FPNAs resemble restructured original neural networks they implement; however, they still miss some necessary properties and parameters to achieve their main goal - to convert the neural networks into structures suitable for implementation in an FPGA. The additional details must be defined using *Field Programmable Neural Network* to reach this goal.

4.1 Field Programmable Neural Network

FPNN (Field Programmable Neural Network) [17] is one of the possible configurations of an FPNA. It defines the interconnections between neural resources and, therefore the FPNN's actual structure, and it defines concrete settings of the parameters and operators:

Definition 4.1.1 (FPNN [17])**.** We say that structure (N, E) is an *FPNN* if the (N, E) is an FPNA and each non-input activator $n \in N$ and each link $(p, n) \in E$ have the following defined:

1. $\Theta_n \in \mathbb{R}$ - initial value of the variable used by the iteration operator i_n. This value represents a threshold.

2. $a_n \in \mathbb{N}$ - the number of iterations to performed by the i_n operator.

3. $W_n(p), T_n(p) \in \mathbb{R}$ - the setting of the affine operator.

4. $\forall p, p \in Pred(n) : r_n(p)$ - a binary flag indicating whether the link (p, n) and the activator n are connected.

5. $\forall s, s \in Succ(n) : S_n(s)$ - a binary flag indicating whether the activator n and the link (n, s) are connected.

6. $\forall p, s; p, s \in Pred(n), s \in Succ(n) : R_n(p, s)$ - a binary flag indicating whether the link (p, n) and the link (n, s) are connected.

Moreover, every input activator $n \in N_i$ has the following defined:

1. $c \in \mathbb{N}$ - the number of inputs

2. $\forall s, s \in Succ(n) : S_n(s)$ - a binary flag indicating whether the input activator n and the link (n, s) are connected.

Definition 4.1.1 declares several binary flags that indicate local connections between activators and links and between links and other links. These flags, as well as the order of

neural resources defined by their *Pred* and *Succ* sets, describe the actual structure of the particular FPNN. It also defines concrete values of other parameters.

The FPNNs do not have the same structures as neural networks, although they can be constructed that way. They are based on a different model that can be structurally different from the original neural network. This also means that the FPNA can differ in its capabilities. In principle, the FPNNs are not a straightforward implementation of neural networks but rather their approximation designed in an FPGA-friendly way. Since the FPNNs can be constructed in various ways and types, the approximation accuracy can be different.

4.1.1 The computation

The neural resources are autonomous components that work with others in parallel, processing the data received from their predecessors (or the FPNN's input) and propagating their results to their successors. The communication between them is based on the request-acknowledgment model. When a neural resource finishes its current computation, it propagates the results to its output and generates a request for each of its successive connected resources (defined by its S_n and $R-N$ flags). These requests notify each successor individually. However, the resources may already be busy processing other requests. Therefore, the original resource waits until all successors accept their requests and send back the corresponding acknowledgment. It only resumes the operation only after all acknowledgments are received. Then, it selects a request from its own input to process or wait until a new request comes.

The operation of a neural resource can be summarized in the following successive steps:

1. The neural resource selects one of the requests waiting on its input. If there are no pending requests, the neural resource waits for a new one to come.
2. The neural resource acknowledges the acceptance of the selected request to the request origin.
3. The neural resource processes the request:
   - *Link* applies the affine operator α.
   - *Activator* applies an iteration operator i_n. Suppose the iteration counter equals a_n, indicating that this is the last supposed iteration, and all the necessary data from all the predecessors have been collected. In that case, the cumulative result of the iteration operator is presented to the function operator f_n, which computes the activation function. The iteration counter is reset. If this is not the last iteration yet, the computation returns to step 1.
4. The result of the computation is propagated to the neural resource's output, and requests for all successors are generated.
5. The neural resource waits until it receives acknowledgments for all generated requests.
6. Return to step 1.

The way a request is selected for processing in step 1 is essential. It is necessary the requests were selected in a way that ensures that all of them will be processed eventually and that the predecessors waiting for the acknowledgments will not be left waiting longer

than necessary. It is also necessary for activators to keep track of the number of iterations they have been through with the current data set to ensure that they work with the correct set and not with the data belonging to a successive set. Failing to do so would cause the entire FPNN result to be wrong, and it could also block the FPNN from processing the following data by breaking the synchronization. The recommended method of selection is Round&Robin.

The grid structure

As we mentioned above, the purpose of FPNNs is to implement neural networks in gate arrays less resource-consuming way by sharing resources between synapses and by simplifying the interconnections. The primary tool to reach this goal is to shape the FPNN into a grid-like structure, illustrated in Fig. 4.2. In the figure, the wide arrows represent links. The thin arrows show the connections between neural resources. It can be seen that the connections are only local between close neural resources. This is to mitigate the need for long buses and complicated routing that would consume a lot of FPGA resources. The intended locality of connections is also why the synapses are broken in chains of several links - so that the entire FPNN would make the intended grid-like structure that would inherently keep the connections local and short.

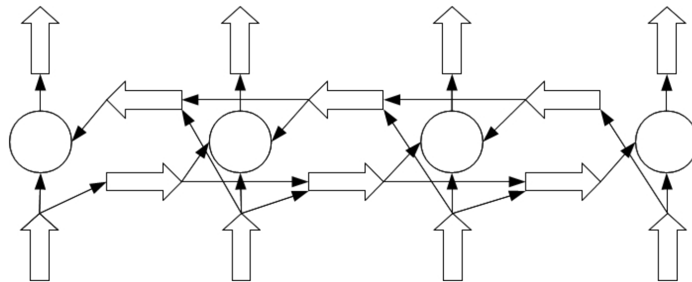

Figure 4.1: A grid FPNN. Circles represent activators, wide arrows illustrate links and thin arrows show the way how the neural resources are connected to each other.

The output of each activator is connected to a single link that is subsequently connected to the following layer. It is connected to an activator in the next layer's corresponding position and to two chains of links (see Definition 4.1.2) that go through the layer in opposite directions. We call this set of chains the *interconnection chain* (Definition 4.1.3).

Definition 4.1.2 (Chain of links)**.** By a chain of links, we understand a sequence of links interconnected in a way that every link is connected to no more than one link on its input and no more than one link on its output, and the flow of data through the entire chain is one-directional.

Definition 4.1.3 (Interconnection chain)**.** By an interconnection chain, we understand a set of two chains of links going through a layer of activators in opposite directions in order to allow the carrying of data from a previous layer to all activators in this layer.

Example FPNN

Let us illustrate the operation of an FPNN by an example. Let the neural network in Fig 4.3 be the original neural network we want to implement using an FPNN. It is a network

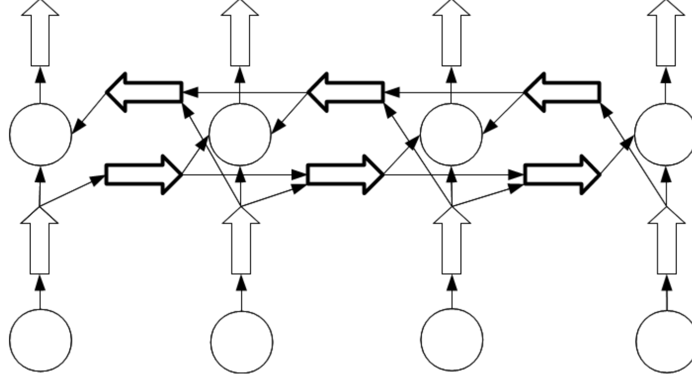

Figure 4.2: Interconnection chain. The highlighted links belong to the interconnection chain.

composed of seven neurons. Three neurons are in the input layer $(n_1, ..., n_3)$, three are in the hidden layer $(n_4, ..., n_6)$ and one is in the output layer (n_7). Corresponding activators will directly replace the original neurons. Chains of interconnected links will replace the synapses, as can be seen in Fig. 4.4. In the figure, the wide arrows represent links. The thin dashed arrows show the local connections between the neural resources.

Using the Definitions 4.0.1 and 4.1.1, we can describe the example FPNN as follows:

$N = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$
$N_i = \{n_1, n_2, n_3\}$
$E = \{(n_1, n_4), (n_2, n_5), (n_3, n_6), (n_4, n_5), (n_5, n_6), (n_6, n_5), (n_5, n_4)$
$, (n_4, n_7), (n_5, n_7), (n_6, n_7)\}$
$i_{n_4} = i_{n_5} = i_{n_6} = i_{n_7} = ((x, x') \rightarrow x + x')$
$f_{n_4} = f_{n_5} = f_{n_6} = f_{n_7} = (x \rightarrow tanh(x))$
$\{\theta_4, \theta_5, \theta_6, \theta_7\} \in \mathbb{R}$
$\forall (p, n) \in E : \exists W_n(p) \in \mathbb{R}; \exists T_n(p) \in \mathbb{R}$
The flags for for $n_4 : r_{n_4}(n_1), r_{n_4}(n_5), S_{n4}(n_7)$ are set.
The flags for for $n_5 : r_{n_5}(n_2), r_{n_5}(n_4), r_{n_5}(n_6), S_{n_5}(n_7)$ are set.
The flags for for $n_6 : r_{n_6}(n_3), r_{n_6}(n_5), S_{n_6}(n_7)$ are set.
The flags for for $n_7 : r_{n_7}(n_4), r_{n_7}(n_5), r_{n_7}(n_6)$ are set.
The flags for links : $R_{n_4}(n_1, n_5), R_{n_5}(n_2, n_6), R_{n_5}(n_2, n_4), R_{n_5}(n_4, n_6),$
$R_{n_5}(n_6, n_4), R_{n_6}(n_3, n_5)$ are set.
$a_4 = 3, a_5 = 3, a_6 = 3, a_7 = 3$
$c_{n_1} = c_{n_2} = c_{n_3} = 1$
All the other flags are not set

The description defines operators and parameters, but most importantly, it defines interconnections between neural resources using binary flags. The r flags define connections between activators and the links that precede them. In other words, these flags define from which links the activators receive their input data. The flags S, on the other hand, define the links to which the activators send their output.

The links themselves are named by the names of activators they lay in *between*. However, the fact that a link lies in between a certain pair of activators does not necessarily mean that it is locally connected to both activators because it also might be a part of an intra-layer chain of links that transfers data from one part of the FPNN to another. This is where the need to use the flags r and S to describe the actual connections between activators and surrounding links comes from.

The connections between links themselves are denoted by the R flags. The notation relays on the naming links scheme and it describes connection between link essentially as s two-link chain between source activator src to target activator trg while skipping the common activator cmn in between the two links: $R_{cmn}(src, trg)$. So the flag $R_{n_5}(n_4, n_6)$ says that link (n_4, n_5) is connected to and sends its output to link (n_5, n_6). (See Fig. 4.4.)

Similarly, the r and S flags use the notion of source activator src and target activator trg to describe connections to preceding and successive links. The flag $r_{trg}(src)$ indicates that activator trg is connected to and receives data from link (src, trg). Similarly, the flag $S_{src}(trg)$ indicates that activator src is connected and sends its output to link (src, trg).

Figure 4.5 illustrates how the example FPNN processes input data. The illustration is broken into six steps with a different set of neural resources working in parallel in each step. The grey filling illustrates the neural resources computing in parallel in the particular step, while the solid thin arrows show the directions in which they sent the results of their computation and corresponding requests. The steps are explained as follows:

1. The first step happens after the input data are introduced into the FPNN and propagated through the input nodes into the three links that connect the input layer to the hidden layer. After the links finish the computation, they generate requests for the successive neural resources - to the closest activator and the links in the chain going through the hidden layer.

2. In the second step, the activators process the incoming requests using their iteration operators. The links in the hidden layer's interconnection chain process and carry the data to deliver them to the activators that are not directly above each other in the layer.

3. In the third step, the activators process their second requests using the iteration operators. The two links at the end of the hidden layer chain generate the third and last requests for their successive activators. With those, the data from the furthest input activators in the input layer are delivered to the activators on the edges of the hidden layer.

4. In the fourth step, the activators have all requests they were waiting for; therefore, after they apply the iteration operator to the third and last request, they pass the cumulative result to the function operator and compute the activation functions.

5. In the fifth step, the three links between the hidden layer and the output layer process the data and deliver requests to the activator in the output layer.

6. In the last step, the activator in the output layer processes all the requests using the iteration operator and the function operator. The output it generates is the output of the entire FPNN.

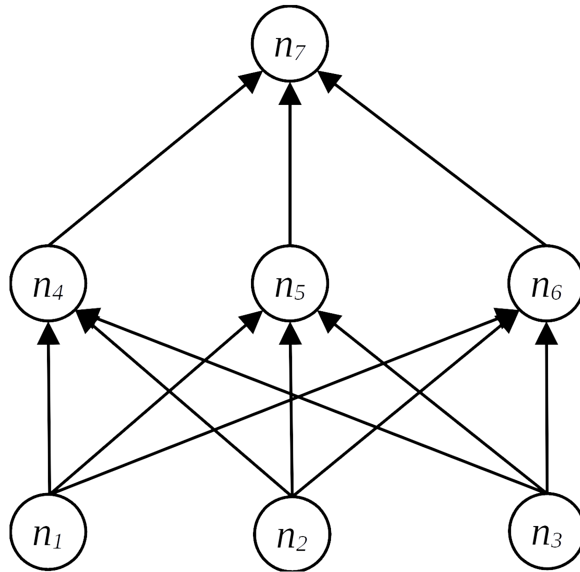

Figure 4.3: Original network for the example FPNN.

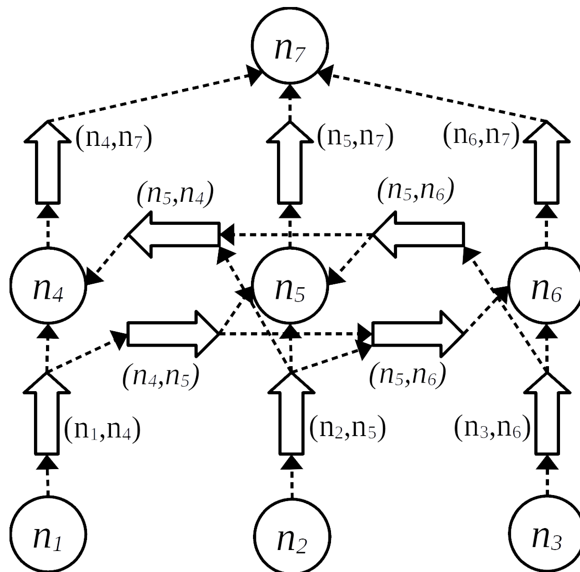

Figure 4.4: Example FPNN

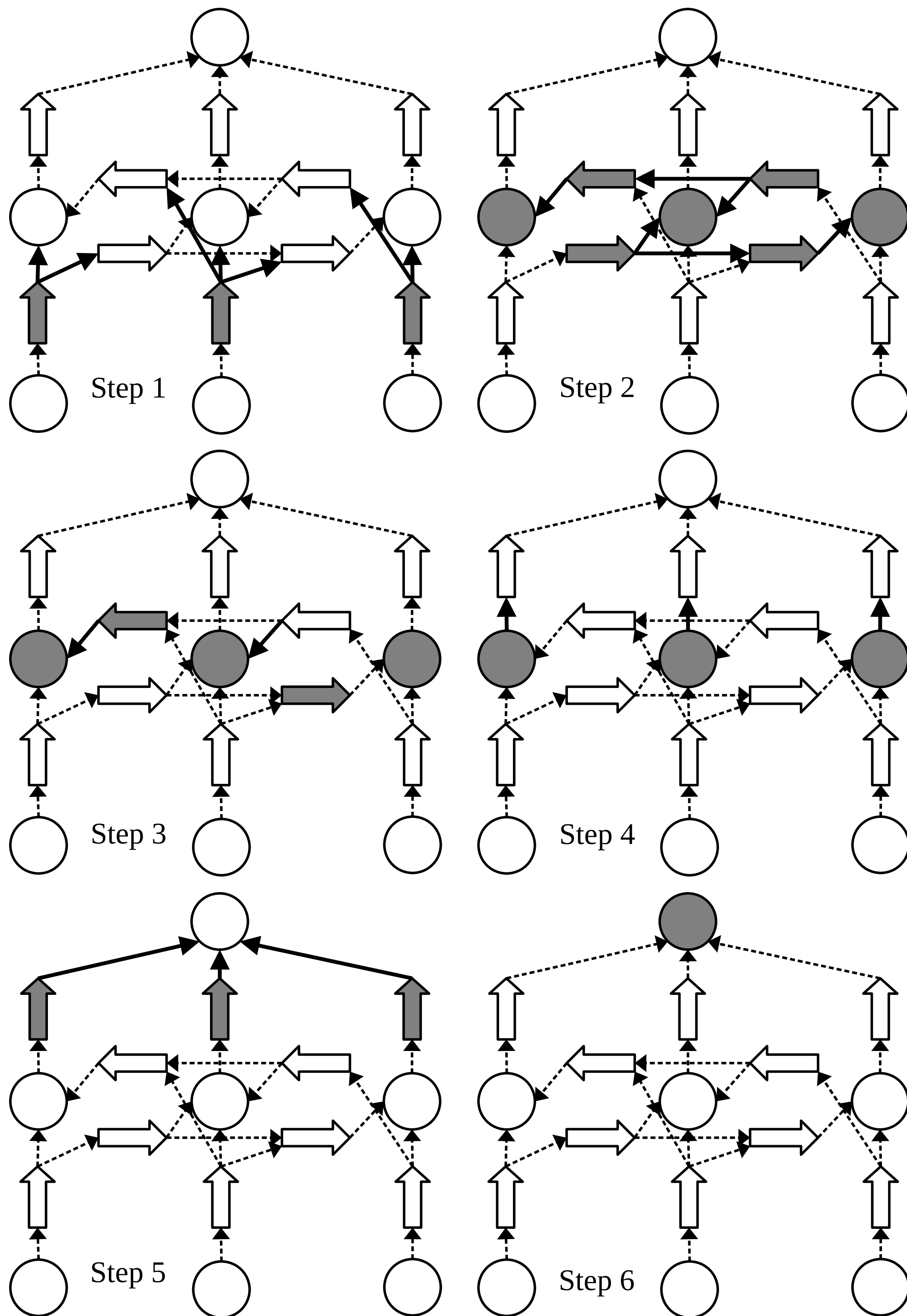

Figure 4.5: Successive steps during an FPNN operation. The grey-filled neural resources work in parallel and generated requests to their successors along the solid thin arrows.

Chapter 5

Research progress

In our research, we focused on improving the FPNN concept according to its defining focus - the effective use of FPGA resources to reduce their consumption. There are different ways of how FPNNs can be implemented and constructed from neural resources. Some FPNNs have been constructed in a way that utilized long connections between neural resources. However, it is the simplification of the connections structure and the effort to keep them as short as possible that can bring a significant reduction in FPGAs resource consumption. Therefore, our goal was to focus on preserving the connections between neural resources as short and local as possible while offering different levels of approximation capabilities coupled with different levels of resource consumption on those different approximation levels.

Another of our goals was to incorporate fault tolerance principles into the FPNN concept, following the focus of our research group to create a possibility to build fault-tolerant implementations and FPGA accelerators of neural networks. We aimed to consider and evaluate traditional methods based on redundancy but also investigate methods that would not rely on redundancies. Instead, we wanted to focus on methods that would utilize the already existing neural resources in the FPNN to harden it or to provide a way to recover from a possible fault. One way we aimed to investigate was supposed to be based on modifications of the given FPNN's parameters existing within the neural resources that the FPNN was built of. Such a technique would take advantage of the naturally redundant structure of the FPNNs; a feature inherited from the neural networks that inspired their creation.

Furthermore, just like the neural networks themselves, the FPNNs are generally soft-computing systems. By their nature, their outputs are always approximate to ideal results, the consequence of the learning process. Therefore, a bit-wise precision of their output values might not be as critical as it would be in systems based on high precision. Especially if the output values, although a bit different, still represent the correct pattern that would, for instance, still classify the input data into the correct class. Consequently, a certain margin of error in the output data might be acceptable in some tasks, even though it certainly would not be acceptable for some other tasks. The point is that given the particular task and margins, just like a neural network providing inexact results due to a fault or a different reason might not be critical, it might also not be critical for the FPNN to work with an error. Given this, a full recovery from a fault or a perfect hardening against them might not always be required. Also, smaller resource consumption and lower overhead might be preferable over a certain penalty in the precision due to a fault in some cases. Therefore, we decide to explore techniques that work only with the existing parameters of the FPNN

in order to harden it or recover it from a fault, even though they inherently can not provide as good results as the traditional and reliable methods based on redundancy.

Equivalently, to further build up the robustness of FPNN and following research in our research group, we decided to explore the possibilities of using FPGAs' build-in capability to reconfigure themselves. The dynamic partial reconfiguration allows the FPGA to re-program a part of its logic during its operation. We aimed to use online partial dynamic reconfiguration as a possible means to recover from a permanent fault in an FPGA.

5.1 Approximation capabilities

When it comes to FPNNs, especially grid-like FPNNs, there is not necessarily a direct relationship between the number of weights of the original neural network and the number of affine operators in the FPNN that implements it. The topology and structure of the FPNN may be different from the network. The designer can undoubtedly design the FPNN to have basically the same or closely similar structure to the network. The designer can equip the FPNN with many links and interconnect them so that the FPNN would resemble the original network. To do this is an accessible and good choice for small FPNNs. However, the goal of FPNNs is to reduce resource utilization when implemented in FPGAs. The primary tool to achieve that goal is to reduce long connections and simplify the overall structure of the FPNN using primarily only the local interconnections between neighboring neural resources. This reduction is achieved by structuring the FPNN into the grid structure. Using extra links beyond the grid goes against the goal because it introduces additional connections and consumes resources to implement the links as well.

Let us consider the grid structure presented in 4.1. Figures 5.1 and 5.2 illustrate two perspectives on how the original synapses are approximated by chains of links in the grid FPNN. In the figures, the colored solid arrows represent the synapses, and the thin dashed arrows represent the chains of links that approximate the synapse with the same color. The dashed arrows also show the path the data take from an activator in the previous layer to an activator in the following layer.

Figure 5.1 shows how synapses going from a single activator are approximated by chains of links that are subsets of the interconnection chain within the layer. In contrast, the shortest synapse, denoted in red, is realized by the single link that connects the two subsequent activators in the two subsequent layers. The blue synapse is approximated by the chain composed of the inter-layer link and the first link in the interconnection chain. The green synapse uses the same chain expanded by the following link in the interconnection chain, as does the violet synapse.

We can say that each synapse adds one more link into the mutual chain of links that approximates all the previous, shorter synapses. Therefore, each synapse adds an affine operator associated with the link to a series of affine operators that approximates its original weight. If we consider only the operator's multiplicative $W_n(p)$ term (the $T_n(p)$ term is zero), then each weight is approximated by a product of a series of $W_n(p)$ values. The final product can match the synapse's original weight thanks to the last multiplicand that each synapse adds to the series. Therefore, as long as a synapse has a dedicated link in the chain that approximates it and its affine operator is appropriately set, the approximation can be accurate because the final product of all affine operators in the chain can result in the original weight value.

However, that might prove challenging to achieve if the number of activators in the previous layer is higher than two. When there are only a couple of activators in the previous

layer, each occupies its own half of the interconnection chain - one of the two chains going through the layer. In that case, each synapse going out of the activators has its own dedicated link added to the chain, and the approximation is accurate.

When there are more activators, however, all the links in the interconnection chain have to be shared between them, which might put them into conflict and a need to find a compromise between different required values of the affine operators. Fig 5.2 illustrates this situation. The figure emphasises the (n_x, n_y) link. The link lies at the end of the interconnection chain a approximates three different synapses. It is the final link in three chains of links that approximate those synapses denoted in the corresponding colors. Therefore, it is the link that all three synapses need to finalize the product of the affine operators of their particular chain links with the final multiplicand in order to bring the overall product to match their weights. However, the link has only one affine operator by the Definition 4.1.1. Therefore, there are three different $W_n(p)$ values the affine operator needs to have to ensure all three synapses are approximated accurately, which creates conflict.

There are three ways how to handle the conflict. *The first possibility* is to add more links into the FPNN that would take the roles of dedicated links for the particular synapses, offering them the chance to have their own dedicated affine operators to get the product value right. As mentioned in the first paragraph, this solution would consume resources to implement the additional link and increase the complexity of interconnections, making the resource utilization even higher.

The second solution is to try to find a compromise between the conflicting values that would make the FPNN work as well as possible, even without additional resources. Some level of approximation accuracy degradation is inevitable, however. There are several possible ways how to find this compromise, and we focused on this problem in our paper *Mapping trained neural networks to FPNNs* (Paper A). Subsection 5.1.1 explains this research in more detail.

The third approach is to equip the links with more than one affine operator. More operators would provide the FPNNs to approximate the synapses better or even accurately. The level of improvement depends on how many operators we provide to the links. The obvious approach is to provide enough operators to fully approximate all the synapses and weight and make the FPNN accurate. However, it is also possible to provide less than required for accurate approximation in order to save resources. This approach was the focus of our papers *Comparison of FPNNs models approximation capabilities and FPGA resources utilization* (Paper B) and *Comparison of FPNNs Approximation Capabilities* (Paper C). Subsection 5.1.2 describe the research.

5.1.1 FPNNs with a single operator per link

In *Mapping trained neural networks to FPNNs* (Paper A), we followed our work in [33] and focused on the situation when links only have a single affine operator. As we have described above, FPNNs composed of links with singular operators face the challenge of finding a compromise between conflicting values; the FPNNs need their operators to have in order to achieve an accurate approximation of the given network.

The paper proposed a method for mapping the original neural network and its weights into this type of FPNN. The method worked by determining the chains of links approximating the particular synapses and calculating all the ideal values for affine operators for approximating the synapses. Then final compromise values of affine operators were determined as an arithmetic average between the conflicting values.

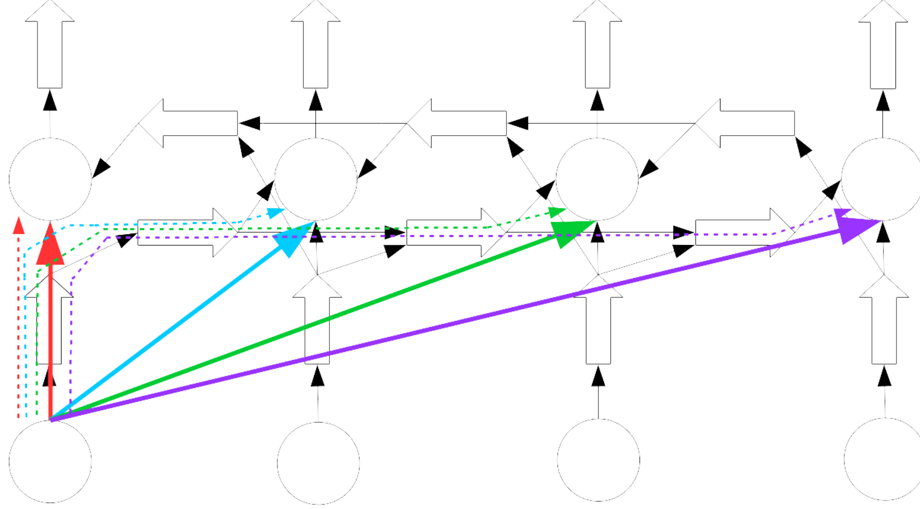

Figure 5.1: The chains of link approximating the particular synapses denote by the same color.

The paper also visited other approaches to finding a compromise we experienced with. The methods were based on weighted averages with weights assigned to the different conflicting values determined by several methods. The first method was based on the length of the synapses or, more precisely, on the length of chains approximating them. If a conflicting value of an affine operator was associated with a longer chain of links, then the value got a higher weight. Lets refer to the (n_x, n_y) link in Figure 5.2 again. The $W_n(p)$ value computed for the red synapse would get a higher weight than the value computed for the green synapse because the chain of links corresponds to the red synapse (denoted by the red dashed line) is longer. The $W_n(p)$ value related to the green synapse would get a higher weight than the one related to the violet synapse because the chain approximating the violet synapse is shorter. This method is called DIST_DP. The second method, called DIST_IP, is an inversion of the first one. This method assigns higher weights to values associated with shorter chains.

The third method is relatively straightforward. It uses the weight of the original synapse directly as the weights of the corresponding $W_n(p)$ values. This method is called WEIG_DP. Like the first method, this one has its inversion, WEIG_IP, which uses inverse proportion as the weights.

The fifth method determines the weights using the product value of the preceding part of the corresponding chain. In the case of (n_x, n_y) link in Figure 5.2, the $W_n(p)$ value computed for the red synapse would get weight computed as a product of $W_n(p)$ values of all the preceding links in the chain denoted by the red dashed line. The actual value of the product would be used as the weight. The method is called PROD_DP, and the derived method that uses inverse proportions of the products is called PROD_IP.

The last two methods are based on the order of the conflicting values. The values would get ordered by their values, and their corresponding weight would match their position in the ordered set. In the case of (n_x, n_y) link in Figure 5.2, there would be three conflicting $W_n(p)$ values. The three values would be directly ordered, and their weights would be their positions. This method is called PVAL_DP. The inverse method PVAL_IP works the same but uses reverse order.

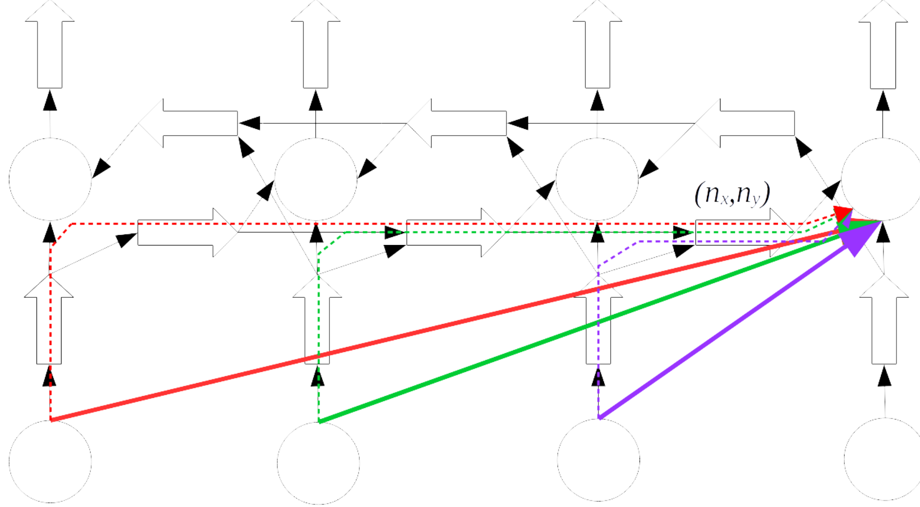

Figure 5.2: The link (n_x, n_y) is the final link in three chains approximating three synapses.

All the methods generally achieved the same approximation accuracy in terms of the number of correctly classified input vectors compared to the original neural network. The accuracy fell into proximity of 65% for all the methods (see Table I in the Paper A). To achieve better results, we have also tried to combine the methods. We used all possible combinations of two, three, and all four basic methods and their inverse version to determine the weights. By using the combinations, we actually achieved up to 8% improvement.

We have also tried to use an optimization algorithm. We selected the Nelder-Mead algorithm [47] to experiment with. The Nelder-Mead algorithm is based on searching for a maximum of an objective function using *simplex*. The simplex is a multi-dimensional polygon, a polytope, that moves around the objective function's graph, trying to fall to a maximum by moving its vertexes around the function graph and evaluating the objective value of its central point.

The objective function was the approximation accuracy; the vertexes were FPNNs with randomly generated $W_n(p)$ values. The algorithm was gradually modifying the $W_n(p)$ values to reach the objective function maxima and find the best performing FPNN. Using this algorithm, we achieved approximately 3% improvement over the best combination of weighted sum methods.

5.1.2 Reduced and Full FPNNs

In *Comparison of FPNNs models approximation capabilities and FPGA resources utilization* (Paper B), we focused on the possibility for the FPNNs to have multiple affine operators to achieve better approximation accuracy. We expanded on the initial Definition 4.1.1 in Definition II.1 in the Paper B. Based on the definition we defined three types of FPNN (Definitions II.6-II.8 in Paper B). The first type is called *Light FPNN*, and it is the type of FPNN we dealt with in Paper A and the previous section. Therefore, the Light FPNNs are FPNNs composed of links equipped with singular affine operators. The second type is called *Full FPNN*. This type of FPNNs has all the affine operators necessary to achieve accurate approximation. Their links have full sets of operators to approximate all the synapses they implement fully. The last type lies between the other two. The *Reduced FPNNs* have fewer affine operators than Full FPNNs but more than one like in the case of Light FPNNs. The

number of operators in Reduced FPNNs is determined by the number of each link connected to preceding neural resources. So, instead of having an affine operator for every synapse ending in it, a link has an operator for every neural resource connected to its input. Should the (n_x, n_y) link in Figure 5.2 lay in a Reduced FPNN, it would have two affine operators. One for the link that implements the violet synapse and the other one for the link that approximates both the red and the green synapse. The second operator would still have a conflict between the two, but it would have been less severe than in the case of the same FPNN of the Light type.

In Paper B we introduced a universal algorithm for mapping a neural network to FPNNs of all three types. We measured the approximation accuracy of Reduced FPNNs implementing different networks with different tasks and structures. The approximation accuracy could get as bad as 50% but also as good as 93%, showing that there might be some cases when reduced FPNNs can provide high approximation accuracy. Prior to these results, previous results of Reduced FPNNs accuracy experiments were published in Paper C.

We have also measured FPGA resource utilization of Reduced and Full FPNN to find out how much resources can be saved by using an FPNN with fewer affine operators. The results show that Full FPNNs can become multiple times more resource-consuming than the Reduced variants. That is especially true for DSP consumption which was generally three times higher. The Full FPNNs' consumption of Look-Up-Tables (LUTs) was also multiple of the consumption of the Reduced FPNNs. However, the consumption rise degree varied more in the case of LUTs than DSPs. Sometimes the consumption has risen only two or three times, but in several instances, it went as high as six times higher than the Reduced FPNNs. The increase of LUTs was correlated to the consumption of the DSPs, and it spiked the highest when the DSPs were depleted, and the implementation had to compensate with the increased use of LUTs.

Moreover, the Full FPNNs proved slower because their operating clock frequency could get lower than half of the Reduced FPNNs frequencies. The Reduced FPNNs all had their working clock period of about twelve nanoseconds. On the other hand, the clock periods of the Full FPNN varied. Some periods were close to the twelve nanoseconds; however, most were longer. A number of the period were close to twenty nanoseconds, with the longest period reaching as high as almost thirty-eight nanoseconds with the largest implementations.

5.2 Fault tolerance

When it comes to fault tolerance of FPNNs we considered both redundancy based and redundancy free methods of hardening. We proposed using Triple Modular Redundancy technique to harden FPNNs or their selected sub-components. Beside this traditional technique we focused on using modification of FPNNs parameters to harden the FPNNs without a need for redundancy that would go against the FPNNs' overall goal to be resource effective implementation of neural networks in FPGAs. We also proposed a method of detecting permanent faults in neural networks' synapses, method that is directly applicable to FPNNs as well.

5.2.1 Identity operators and mapping

In *Fault tolerant Field Programmable Neural Networks* (Paper E, the paper was shortened after it was accepted, the original full length is enlisted as Paper G), we focused on fault-

tolerant mapping of FPNNs equipped with *identity operators*. The idea of the identity operator is to change all inner computation operators of a given neural resource into an identity function. Practically, when applied, it would turn the particular neural resource into a register (see Figure 5.3). We suggested this technique to recover from a permanent fault in the neural resource computing blocks. Even if the resource cannot compute as expected, with the identity operator, it can still let at least pass the data through and let the rest of the FPNN function. However, with the neural resource effectively missing from the FPNN, the computation would most likely suffer a hit in performance and accuracy. The Paper E discusses the feasibility of using the process of mapping the initial neural network to FPNN to reduce the possible accuracy hit of an identity operator activation.

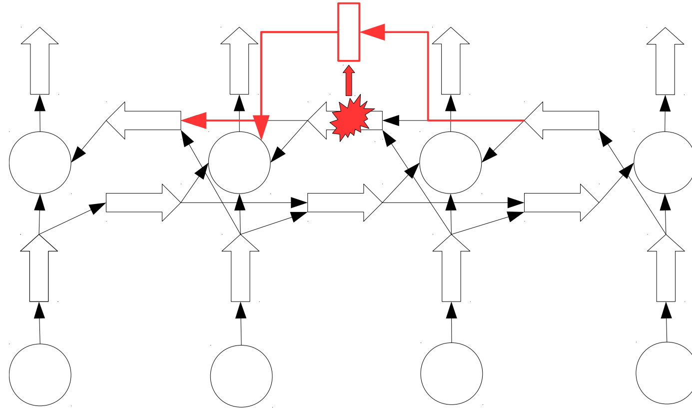

Figure 5.3: Identity operator effect.

We have also used the identity operator as a tool to identify critical links. We were successively activating identity activators, one link at a time, and measuring the impact it had on output results of the small FPNN we have experimented with. Some links have indeed shown to be more critical than others (see Table I in Paper E).

In the next phase, we tried to determine if the negative effects of identity operators can be mitigated using a modified mapping process. During mapping process (described in the full version of Paper E - Paper G and in Paper B), selected links have their identity operators activated. That removed them from chains of links to which the mapping process mapped the original synapses. That forced the mapping process to use other links in chains to approximate the particular weight. The FPNN we experimented with was a light FPNN (equipped with only one affine operator, see subsection 5.1.1). That meant that the value of the affine operator of the missing link came into conflict with the value of the operator taking its place in the mapping. The mapping process would then use the arithmetic average of the conflicting values to determine the final value.

In the experiment, we used the method to harden the FPNN against a fault (and the subsequent activation of the particular identity operator) in every link to which the method was applicable. We first measured the approximation accuracy of the hardened FPNN, then activated the identity operator in the link the FPNN was hardened against. We measured the approximation accuracy of the FPNN with the missing link due to the identity operator. We compared them with the original accuracy and the accuracy of the completely hardened FPNN. We also measured the effect when a different combination of links was considered in the mapping process. The results showed that even though sometimes the hardening

worked, other times it had a negative effect. As anticipated, the method was not universally applicable but rather a potential option to consider.

5.2.2 Triple modular redundancy

In *Triple modular redundancy used in field programmable neural networks* (Paper F), we focused on Triple Modular Redundancy in FPNNs. We utilized two types of triplicating. The entire neural resources were triplicated with the first type (referred to in the paper as type A). In the case of the second type (referred to in the paper as type B), the building blocks that implement the neural resources were triplicated. There are several components that the neural resources consist of. There are components responsible for the neural resource particular calculation. Other blocks take care of communicating with other neural resources using the request system described in Section 4.1.1. The communication blocks are composed of a multiplexer, a demultiplexer, a register, a generator for generating requests to successors, and a block for selecting a request to process from the pool of received requests from predecessors (see Figures 2 and 3 in Paper F and detailed explanation of the block in the paper's Section 2.1). These blocks and the system they implement are common to both types of neural resources, the link and the activators, because both types need to communicate. The difference between both types lies in their computation units. The links utilize a multiplier that implements its affine operator. The activators use an adder to implement the iteration operator and a unit that implements the function operator - the activation function, using an adder and a multiplier. These computation blocks, the multiplexers, and other blocks that implement the communication system are what we triplicated with type B.

We measured the FPGA resource consumption of all the building blocks as well as the consumption of the whole neural resources. Then we implemented the TMR technique on the two different levels and measured the consumption of the hardened versions of the implementation. Understandably, resource consumption often increased by more than 200%. The communication blocks often increased their consumption of registers significantly more than their consumption of Look Up Tables (LUTs). This is expected because the communication blocks do not utilize much computation. Also, given their relatively smaller size, the consumption of voters added by the TMR contributed significantly. On the other hand, the computation blocks that implement the neural resources' operators primarily increased their LUTs consumption.

When it comes to type B triplicating, when the whole neural resources were triplicated, the overall resource consumption was lower than when all the inside blocks were triplicated in type A. This is expected because type B does not suffer from the overhead coming with the additional voters and connections needed to make the inside blocks hardened with the TMR technique. The type B neural resources consumed fewer registers (the link consumed 2% less, and the activators consumed 11% less). The consumption of LUTs dropped even more to 15% for the link and 19% in the case of the activator. These results correlate to the fact that the computation blocks consume from 80% to 90% of resources.

We considered the two types because it might be feasible to harden only some subset of the blocks instead of the whole neural resources. The computing blocks are apparent candidates for hardening but hardening the communication blocks while leaving the computation blocks unhardened might be a valid option as well, even when they are relatively small and therefore less likely to experience a fault. Not only their resource overhead is significantly lower, but also the communication blocks are vital for the function of the entire

FPNN. Should one of the resources stop generating requests and responding to them, the whole FPNN would eventually stop computing. Computation blocks producing erroneous data are naturally undesired; however, given the soft computing nature of neural networks, such errors might significantly impact the outputs, but they also might not. On the other hand, failure in communication between neural resources will always be critical. Moreover, hardening the communication blocks can be easily combined with an idea of identity operator (discussed in Section 5.2.1) in order to produce low-overhead FPNN hardened against communication errors if not against errors in computation or data.

5.2.3 Detecting hard synapses fault

In *Detecting hard synapses faults in artificial neural networks* (Paper D), we focused on a way how to detect permanent faults in neural network synapses or their weights. In particular the work focused on *stuck* and *noisy synapses* as referenced to in Section 3.1. The general idea of the discussed method was related to the idea of identity operators discussed in Section 5.2.1. The algorithm (see Algorithm III.B in Paper D) was based on testing all the network's synapses iteratively, one at a time. For every synapse, the algorithm determined the set of synaptic sequences connecting the input layer through the hidden layers to the output layer that the tested synapse was part of. Let's say the synapse between neurons n_x and n_y - (n_x, n_y) was under test. Then the algorithm would determine all the sequences of synapses that connect the neuron n_x to the input layer through the neurons in the previous layers. Similarly, the algorithm would determine the sequences of synapses that connect the neuron n_y to the output layer through the neurons in all the successive layers. Sequences from both sets connect in the tested synapse, and therefore, they represent all the possible paths the data can flow from the network's input to its output while going through the (n_x, n_y) synapse.

In the next step, one of the sequences was selected. All the weight values in the sequence, save the weight of the synapse under the test that was left unchanged, were set to one to make the synapses transparent to data passing through them. Then, a predetermined testing data vector was introduced to the network's input. The vector was composed of identical values except for the element that went to the input neuron in the selected sequence. The output data were collected and compared to the expected result that was calculated separately. Since all the parameters of the network and the input data were known, it was possible to calculate the output data the network should ideally produce independently of the network itself. This reference data was then compared to the network's output data. If the data were equal, the network performed as expected, and the synapse passed the test. If the data differed, however, then there were three possibilities. First, the synapse under test might have produced erroneous data that caused the difference. Second, there was another synapse producing erroneous data in the tested sequence. It could also have been both. The algorithm would repeat the test with several different sequences containing the tested synapse to determine which was the case. If all the output data were different than expected, then the synapse under the test was faulty. If only some of the output data differed, another synapse was faulty. Which synapse it was would be determined when the synapse would go through the testing itself.

The paper also discusses a problem that may complicate fault detection. In cases that the value of the tested synapse's weight is high, it might cause saturation of the next neuron (the high input would make the neuron generate go to the minimum or the maximum of its activation function). If the synapse was faulty and the fault would cause an error that would

further increase the weight's value, the error might be left undetected. A faulty synapse may also influence other synapse tests by saturating the neurons when it is selected to the tested sequence. Therefore, it might lead to false positive detection. The paper suggests that the saturation problem may be mitigated by modifying not only the weight of the synapses in the testing sequence but also the neurons' activation function. Suppose the activation function is replaced by an identity function. In that case, its output could reach any value in the range of the used data type. Therefore it would mitigate the saturation problem and allow the erroneous value to propagate to the network's output unchanged. That would allow determining the actual value of the error. Section III.C of the Paper 5.2.1 describes the algorithm utilizing the activation function modifications.

The paper also discusses using different values to set the weight in the tested sequence as well as different choices of the input data. Section IV describes experiments we performed with the method that does not utilize the activation function modifications, as it is more challenging due to the saturation problems. Tables I-V present the results of the experiments. The results show that choosing a higher input value of the element entering the tested sequence as well as a smaller value for the other elements helps prevent the saturation problem. It also demonstrates that choosing low negative values for weights in the tested sequence improves the detection. Overall, the method showed relatively decent detection abilities despite the saturation problem.

5.2.4 The FPNNs robustness

In a yet unpublished paper *Fault tolerance of different Field Programmable Neural Networks types* (Paper I), we focused on the robustness of FPNNs themselves. We experimented with all the three types of FPNNs - light, reduced, and full as discussed in Section 5.1. The FPNNs utilized fixed-point computation with an 8-bits integer part, and the remaining 8-bits were used for the fraction part as recommended in [44]. We experimented with six different FPNNs with different structures, all of them in their light, reduced, and full versions. Therefore, the number of individual FPNNs under the experiment was eighteen. Twelve of the FPNNs were performing the *Diabetes* task, and the six remaining were implementing the *Thyroid task*, both tasks being neural network benchmark classification tasks from the *Proben* set of benchmarks [61].

Faults were injected into all affine operators of the particular FPNNs. That meant that each FPNN had a different number of faults injected into it. We chose this approach because the FPNNs had very different sizes; therefore, choosing a constant number of fault injections for all of the FPNNs would give some of the larger FPNNs an artificial advantage. The faults were injected as bit flips in the affine operators' variables. The bits to flip were chosen randomly (see Section III.B in Paper I). Each FPNN with an injected fault was presented with the testing data set, and its outputs were compared to the outputs of the original FPNN without a fault to see if the FPNNs classified the input data vectors into the same classes. Tables II and III in Paper I illustrate the worst, the best and the average results the faulty FPNNs achieved in terms of how many percent of the input data vectors the faulty FPNNs classified to the same classes. The **Min** column shows the worst result reached by any of the faulty FPNNs of the given type, task, and structure. The **Max** shows the best result, and the **Avg** shows the average success of the particular FPNN.

The results have shown that light FPNNs were more robust against the infected faults. However, we attribute this to the light FPNNs' lower approximation capabilities and, therefore, the approximation accuracy they can achieve. The lower accuracy can mask some of

the injected faults because the final computation result might be the same, even with an erroneous weight value.

The full FPNNs, on the other hand, proved to be most robust for the *Diabetes* task (see Table II in Paper I). Their high redundancy gave them the inherent robustness that massively parallel structures like neural networks have. We can see they performed best in terms of both best and worst results. Their average success rate was also the best of all three types.

However, in the case of the *Thyroid* task, it was the reduced FPNN that performed the best (see Table III in Paper I). The full FPNNs actually provided worse results than the FPNNs with fewer affine operators. We believe this phenomenon is due to the interconnection chain's length and the input layer's size. These two facts mean that there was a high number of affine operators in the relatively long interconnection chains. Any error introduced into these affine operators there would have a higher chance of causing a more significant effect as its influence would propagate through the interconnection chain and impact a higher number of the following affine operators in the successive links. Therefore, any error caused by the injected fault would more easily escalate into higher impact as opposed to the FPNN implementing the *Diabetes* task that did have smaller concentrations of the affine operators.

Recovery using θ parameters modifications

Besides evaluating how robust the FPNNs can be in this paper, we also experimented with a recovery method following the research described in 5.2.1. In that method we tried to recover faulty FPNNs using identity operators and utilizing the process of mapping the original neural network to the FPNN. In this paper we experimented with a method modifying the θ parameters of the activators. These parameters serve the same function like the neurons' thresholds. Therefore, modification of these parameters would effectively affected the activation functions. Just like with the method suing identity operators, the experiments with this method were to determine if the modification of the parameters could be used to recover from a fault in a link's affine operator. Therefore if this method presented another option to recover using modifications of existing FPNN parameters without utilizing more complex methods like remapping or retraining or relying on redundancy based methods. Similar to the method using the identity operators, we did not expect this method would prove universally usable but rather a possibility to consider.

The method relied on knowing the value of the error in the affected affine operator caused by the injected fault. The section 5.2.5 describes a method that could be used to determine the value. The method used the value of the error as a modifier to the θ operator of the closest following activator to the affected link. There were several ways the method used the value to modify the θ (see Table I. in I). We also expanded the method by using the modifiers only when the faulty link was a direct successor of an activator. It was because such a link would be at the beginning of some of the chains of link in the FPNN, therefore an error in one of its affine operators would have potentially the most significant impact because the erroneous data it would produce would go through links in the rest of the chain which could escalate the error's effect.

The results (see Table IV and Tabke V) was not very assuring. Even though there was measurable improvement in the recovered FPNNs' performance compared to faulty FPNNs, the results generally showed decrease of the recovered FPNN's performance. No scenario of modification showed consistent pattern of positive influence on the results and the potential

to be useful in recovery. What the results illustrated however, was the vulnerability of the FPNNs to the changes in the θ parameters. The recovery attempts behaved more like additional fault injections furthermore negatively impacting the FPNNs' performance.

5.2.5 Recovery using partial dynamic reconfiguration

One of the main goals of our research was to examine and potentially implement recovery from faults using partial dynamic reconfiguration (PDR). It is the ability of FPGAs to reconfigure one or more of their smaller parts, referred to as frames, instead of reprogramming the entire logic array. Not only is the process quicker, but more importantly, it allows the designs implemented in the remaining frames to continue to function without an interruption, which would be necessary for reprogramming the whole FPGA.

Our research group used partial dynamic reconfiguration extensively for works related to fault tolerant systems. We have used it for injecting faults into a hardened robot controller [55, 56, 57, 58, 59] as well as into a controller of an electronic lock [36, 60] during the experimental evaluation of their robustness. We have also developed a controller capable of supervising a recovery from fault using PDR [31, 41, 42, 71, 72] and we worked on hardening this reconfiguration controller as well[49, 50, 51]. We have also addressed the re-synchronization of the different parts of a design implemented in an FPGA after the design recovered from fault using PDR [74, 75, 76, 77].

Other researchers utilized partial dynamic reconfiguration in fault-tolerant designs as well. For instance, in [38], the authors introduced a hardened voter for systems based on Triple Modular Redundancy. The design used PDR to recover the voter from fault introduced by Single Event Upsets. A survey of various hardening techniques, including double and triple modular redundancy and suggesting DPR for recovery from faults caused by Single Event Upsets, was published in [69].

The authors of [13] considered the vulnerability of FPGAs and the PDR process to errors in the reconfiguration bitstream. They suggested using a partial hardening of the critical parts of the bitstream, the part that holds address and control information. This method was put in contrast with the method proposed by Xilinx based on hardening all the bitstream parts using CRC, which has a lot of spacial and temporal overhead.

The hardening approaches can also be combined. In [83] the authors presented a framework that would harden the provided design using TMR a recover from faults using PDR. The authors used the framework to harden a design implementing a neural network for hand-written characters recognition. The DPR was also used as a recovery technique of an OpenRISC processor implemented in FPGA [62]. The authors have broken the processor into reconfigurable modules and duplicated them in order to detect a SEU and identify modules that required to be reconfigured to mitigate the fault. The duplication was also used for fault detection in [4]. The authors implemented a pacemaker divided into a set of separated, parallel-operated modules and used the spare FPGA resources as a backup space for implementing a replacement module. If a fault was detected in a module, it would have been implemented in the spare space using dynamic reconfiguration to restore the design's functionality. Instead of TMR, the authors of [27] decided to use two hard-core processors to control the recovery process and temporarily take the place of a faulty module. Their application implemented an FIR filter that was duplicated for fault detection. If a SEU was detected, one of the processors identified the faulty region and triggered and controlled recovery using PDR while the other processor was computing the FIR filtration software.

The approach saved 41% of resources compared to TMR. The reduction was traded off with temporarily slowed-down computation.

The goal of partial dynamic reconfiguration in this work was different from the approaches used by our research group or by other researchers. All the mentioned works were based on traditional PDR that would take a correct pre-prepared bitstream representing the desired design and use it to reconfigure an FPGA to the desired state. However, within this work, we wanted to consider using *online* partial dynamic reconfiguration - a method that would not use pre-prepared bitstreams. Instead, the bitstream would be generated from scratch inside the FPGAs themselves. The reasoning behind this idea was that it would allow for recovery from a wide range of permanent faults, even those that it was not hardened against. With proper detection and localization methods, the FPGA would create a new bitstream implementing a new design that would perform the same function but mask the detected fault. As much as this idea is intriguing from the fault tolerance point of view, we deemed it unfeasible in the end.

The first problem is that if such a technique is supposed to be practical and usable, it would require extensive knowledge of the selected FPGA bitstream format and its inside representation. Such information is proprietary to FPGA manufacturers that are not keen to pride them in full scale to the FPGA community. A very demanding (and questionable) process of reverse engineering would be required, and even though projects such as *Project X-Ray* [2] exist, they do not provide complete information and are aimed to support the development of external design tools.

To generate bitstreams for PDR inside an FPGA would also require a lot of its resources. Implementing a design into an FPGA is a demanding process, especially routing the logic in the FPGA to the desired working design. Except that the design modifications that would go beyond simple changes in contents of selected Look Up Tables or minor modifications would ideally require an established set of synthesis tools to be present to perform the proper implementation. Both of these facts would introduce a need to have a processor core [8] implemented in the FPGA that would sufficiently support the needed tools. Even though different soft-core and hard-core processors are available for different FPGAs, the current development in the FPGA technology drives toward integrating FPGAs with other technologies. Also, the current trends lead to further improvements in the usefulness of partial dynamic reconfiguration used in the traditional sense. Improvements can provide generally more practical results than online reconfiguration can.

The rise of integrated devices utilizing processor cores together with a field programmable gate array area, such as Zynq, makes implementing hardened neural network accelerators easier than before. It is possible to utilize Linux operating systems [1, 34] that would provide helpful support for programming systems using partial dynamic reconfiguration to implement support systems utilizing the programmable area. The systems can be dynamic, changing according to particular situations and needs. Therefore these systems are ideal instruments for implementing hardened systems that can use dynamic reconfiguration to recover from faults such as SEU. Frameworks such as FRED [9, 48] and PYNQ [18] can be conveniently used for such purposes.

5.3 List of Publications Related to the Thesis

This thesis describes a research that was presented in several related papers the thesis refers to. The list of the related publications is as follows:

2015

- M. Krcma, J. Kastil and Z. Kotásek, „Mapping Trained Neural Networks to FPNNs,“ *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 2015, pp. 157-160, doi: 10.1109/DDECS.2015.50.

  Author participation: 75% | Conference rank: B3 (Qualis)

- M. Krcma, Z. Kotasek and J. Kastil, „Fault tolerant Field Programmable Neural Networks,“ *2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)*, 2015, pp. 1-4, doi: 10.1109/NORCHIP.2015.7364381.

  Author participation: 80% | Citations: 1 | Conference rank: B3 (Qualis)

- KRČMA Martin. FPNN – neuronové sítě v FPGA. In: *Počítačové architektury a diagnostika PAD 2015.* Zlín: Faculty of Applied Informatics, Tomas Bata University in Zlín, 2015, pp. 13-18. ISBN 978-80-7454-522-1.

  Author participation: 100%

2016

- M. Krcma, Z. Kotasek and J. Lojda, „Implementation of fault tolerant techniques into FPNNs,“ *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 297-298, doi: 10.1109/FPT.2016.7929559.

  Author participation: 80% | Citations: 1

- M. Krcma, J. Kastil, Z. Kotásek and J. Lojda, „Comparison of FPNNs Approximation Capabilities,“ *Proceedings of the Work in progress Session held in connection with DSD 2016*, 2016, pp. 1-2, ISBN:978-3-902457-46-2.

  Author participation: 80% | Conference rank: B1 (Qualis)

- KRČMA Martin. Koncept Field Programmable Neural Networks odolný proti poruchám. In: *Počítačové architektury a diagnostika PAD 2016.* Bořetice - Kraví Hora: Faculty of Information Technology BUT, 2016, pp. 97-100. ISBN 978-80-214-5376-0.

  Author participation: 100%

2017

- M. Krcma, Z. Kotasek and J. Lojda, „Comparison of FPNNs models approximation capabilities and FPGA resources utilization,“ *2017 13th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2017, pp. 125-132, doi: 10.1109/ICCP.2017.8116993.

Author participation: 85% | Conference rank: C (Qualis)

- M. Krcma, Z. Kotasek and J. Lojda, „Triple modular redundancy used in field programmable neural networks,“ *2017 IEEE East-West Design & Test Symposium (EWDTS)*, 2017, pp. 1-6, doi: 10.1109/EWDTS.2017.8110128.

Author participation: 85% | Citations: 5

2019

- M. Krcma, Z. Kotasek and J. Lojda, „Detecting hard synapses faults in artificial neural networks,“ *2019 IEEE Latin American Test Symposium (LATS)*, 2019, pp. 1-6, doi: 10.1109/LATW.2019.8704637.

Author participation: 80%

5.3.1 Author's contributions to papers related to The Thesis

The author of this thesis is the main author of all of the papers related to this thesis. The author was responsible for all the research presented in the papers as well as for the implementation and experimental work. The author however relied on the research group with papers corrections, proof-reads, technical assistance and most importantly on discussions of the research.

5.4 List of Other Publications, unrelated to the Thesis

Beyond the research presented in this thesis, the author participated in several other research efforts unrelated to the thesis. The list of the resulting publications is as follows:

2016

- J. Lojda, J. Podivinsky, M. Krcma and Z. Kotasek, „HLS-based fault tolerance approach for SRAM-based FPGAs,“ *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 301-302, doi: 10.1109/FPT.2016.7929561.

Author participation: 5% | Citations: 4

2017

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, LOJDA Jakub, ŠIMKOVÁ Marcela, KRČMA Martin and KOTÁSEK Zdeněk. Functional Verification Based Platform for Evaluating Fault Tolerance Properties. *Microprocessors and Microsystems*, vol. 52, no. 5, 2017, pp. 145-159. ISSN 0141-9331.

Author participation: 8% | Citations: 4 | Impact factor: 1.045 (Q3)

- J. Lojda, J. Podivinsky, Z. Kotasek and M. Krcma, „Data types and operations modifications: A practical approach to fault tolerance in HLS,“ *2017 IEEE East-West Design & Test Symposium (EWDTS)*, 2017, pp. 1-6, doi: 10.1109/EWDTS.2017.8110113.

  Author participation: 5% | Citations: 8

2018

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin, BURGET Radek, HRUŠKA Tomáš and KOTÁSEK Zdeněk. „A Processor Optimization Framework for a Selected Application,“ In: *Proceedings of IEEE East-West Design & Test Symposium.* Kazan: IEEE Computer Society, 2018, pp. 564-574. ISBN 978-1-5386-5710-2.

  Author participation: 20% | Citations: 1

- J. Lojda, J. Podivinsky, Z. Kotasek and M. Krcma, „Majority Type and Redundancy Level Influences on Redundant Data Types Approach for HLS,“ *2018 16th Biennial Baltic Electronics Conference (BEC)*, 2018, pp. 1-4, doi: 10.1109/BEC.2018.8600951.

  Author participation: 10% | Citations: 3

2019

- ČEKAN Ondřej, PODIVÍNSKÝ Jakub, LOJDA Jakub, PÁNEK Richard, KRČMA Martin and KOTÁSEK Zdeněk. Testing Reliability of Smart Electronic Locks: Analysis and the First Steps Towards. In: *Proceedings of the 2019 22nd Euromicro Conference on Digital System Design.* Kalithea: Institute of Electrical and Electronics Engineers, 2019, pp. 506-513. ISBN 978-1-7281-2861-0.

  Author participation: 19% | Citations: 1 | Conference rank: B1 (Qualis)

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin, BURGET Radek, HRUŠKA Tomáš and KOTÁSEK Zdeněk. Multidimensional Pareto Frontiers Intersection Determination and Processor Optimization Case Study. In: *Proceedings of the 2019 22nd Euromicro Conference on Digital System Design.* Kalithea: Institute of Electrical and Electronics Engineers, 2019, pp. 597-600. ISBN 978-1-7281-2861-0.

  Author participation: 20% | Citations: 2 | Conference rank: B1 (Qualis)

2020

- LOJDA Jakub, PÁNEK Richard, PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin and KOTÁSEK Zdeněk. Analysis of Software-Implemented Fault Tolerance: Case Study on Smart Lock. In: *2020 IEEE East-West Design and Test Symposium, EWDTS 2020 - Proceedings.* Varna: Institute of Electrical and Electronics Engineers, 2020, pp. 24-28. ISBN 978-1-7281-9899-6.

  Author participation: 7% | Citations: 1

- LOJDA Jakub, PODIVÍNSKÝ Jakub, ČEKAN Ondřej, PÁNEK Richard, KRČMA Martin and KOTÁSEK Zdeněk. Automatic Design of Reliable Systems Based on the Multiple-choice Knapsack Problem. In: *Proceedings - 2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2020.* Novi Sad: Institute of Electrical and Electronics Engineers, 2020, pp. 1-4. ISBN 978-1-7281-9938-2.

  Author participation: 10% | Citations: 2 | Conference rank: B3 (Qualis)

- PODIVÍNSKÝ Jakub, LOJDA Jakub, PÁNEK Richard, ČEKAN Ondřej, KRČMA Martin and KOTÁSEK Zdeněk. *Evaluation Platform For Testing Fault Tolerance: Testing Reliability of Smart Electronic Locks. In: 2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS).* San José: IEEE Circuits and Systems Society, 2020, pp. 1-4. ISBN 978-1-7281-3427-7.

  Author participation: 10% | Citations: 2 | Conference rank: B5 (Qualis)

- LOJDA Jakub, PÁNEK Richard, PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin and KOTÁSEK Zdeněk. Hardening of Smart Electronic Lock Software against Random and Deliberate Faults. In: *Proceedings - Euromicro Conference on Digital System Design, DSD 2020.* Kranj: Institute of Electrical and Electronics Engineers, 2020, pp. 680-683. ISBN 978-1-7281-9535-3.

  Author participation: 12% | Citations: 3 | Conference rank: B1 (Qualis)

- PODIVÍNSKÝ Jakub, ČEKAN Ondřej, KRČMA Martin, BURGET Radek, HRUŠKA Tomáš and KOTÁSEK Zdeněk. Iterative Algorithm for Multidimensional Pareto Frontiers Intersection Determination. In: *2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS).* San José: IEEE Circuits and Systems Society, 2020, pp. 1-4. ISBN 978-1-7281-3427-7.

  Author participation: 20% | Conference rank: B5 (Qualis)

2020

- J. Lojda, R. Panek, J. Podivinsky, O. Cekan, M. Krcma and Z. Kotasek, „Testing Embedded Software Through Fault Injection: Case Study on Smart Lock," *2021 IEEE 22nd Latin American Test Symposium (LATS),* 2021, pp. 1-6, doi: 10.1109/LATS53581.2021.9651770.

  Author participation: 5%

5.5 Research Projects and Grants

- FIT-S-20-6309 — *Design, Optimization and Evaluation of Application Specific Computer Systems*, Brno University of Technology, team member.

- 8A18014, Proposal ID 783119-2 — *SECREDAS - Product Security for Cross Domain Reliable Dependable Automated Systems*, ECSEL Joint Undertaking, team member.
- FIT-S-17-3994 — *Advanced parallel and embedded computer systems*, Brno University of Technology, team member.
- LQ1602 — *IT4Innovations excellence in science*, MŠMT CZ, team member.
- 7H14002, 621439 — *Algorithms, Design Methods, and Many-Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing*, Artemis Joint Undertaking, team member.
- FIT-S-14-2297 — *Architecture of parallel and embedded computer systems*, Brno University of Technology, team member.
- LD12036 — *Methodologies for Fault Tolerant Systems Design Development, Implementation and Verification*, MŠMT CZ, team member.
- 7HZC13005 — *Portable and Predictable Performance on Heterogeneous Embedded Manycores*, Artemis Joint Undertaking, team member.
- ED1.1.00/02.0070 — *The IT4Innovations Centre of Excellence*, MŠMT CZ, team member.

Chapter 6

Conclusions

The research presented in this thesis focused on developing multiple types of Field Programmable Neural Networks in order to provide different options with different trade-offs between their resource consumption and their approximation accuracy. One of the goals was to remain faithful to the core idea of the FPNN concept - to minimize resource consumption by keeping the connections between neural resources as short and local as possible. We can achieve this by making the FPNNs in a grid structure suitable for keeping all the connections only between the closest neural resources that neighbor each other. This measure can prevent the need to have many long connections across several neural resources and, therefore, reduce resource consumption.

In our research, we decided to keep to this principle. We researched three types of grid FPNNs that differed in the number of affine operators they were equipped with. The key idea was to reduce the number of affine operators, which would reduce the amount of resources needed to store them. And because the usual FPNN has more links than activators, reducing each link's resource consumption could provide a significant savings of resources. It is not only the amount of resources needed to store the values of affine operators that would be reduced. Assuming that in many cases, the computing units realizing the computation with the affine operators would be composed of Look Up Tables, the reduction of the number of affine operators would make the computing units simpler and, therefore, smaller and faster. With a low number of affine operators, some of the computing units could even be replaced by simple constant multiplication in some cases.

However, just like the reduction in the size of the affine operators set brings savings of resources, it also reduces the approximation power of the FPNN. With fewer operators, the FPNN also holds less information; therefore, its computation ability is reduced. With reduced computation power, the FPNN is less capable of reproducing the results of the original neural network it implements, so its approximation capabilities and accuracy is affected negatively.

One of the questions we have sought to answer in our research is how severely reduced the approximation accuracy actually is and what are the trade-offs with resource consumption. The experiments show that the answer depends on the original neural network the FPNN approximates, the task the neural network learned to perform, and naturally on the number of affine operators we decided to equip the FPNN with. Some Reduced FPNNs showed only a few percent decrease in the approximation accuracy while being times smaller than the Full FPNNs. Other Reduced FPNNs, however, showed a significantly higher decrease in the approximation accuracy by tens of percent. This was the case with Light FPNNs as

well. Ultimately, the designer's choice must be led by the given situation and the neural network intended to be implemented with an FPNN.

Another goal of our research was to explore fault-tolerant properties of FPNNs and methods of their hardening. FPNNs, just like neural networks, are massively parallel but relatively homogeneous structures. These properties might provide them with some inherent robustness; however, grid FPNNs are composed of potentially long chains of links that approximate the original synaptic interconnections between layers. This introduces an additional dependency between neural resources. If a fault occurs in an FPNN, it might be amplified while its consequences would propagate through the FPNN due to this dependency. It may also prevent the FPNN from proper operation. This is the reason why we have also focused on permanent faults that could seize the FPNNs from function.

Even though we worked with the traditional hardening method based on replication - the Triple Modular Redundancy technique, we primarily focused on methods that would not rely on replicating the components. This was following the FPNNs main goal of reducing resource consumption. Aligned with this goal, we investigated recovery methods based on modifications of FPNNs' parameters rather than utilizing their resources' replicas.

We proposed a replication-free method of recovering from a permanent fault in a link that would stop its computation. The method utilized *identity operators* that would make the faulty link transparent to the passing data, so it would allow the FPNN to continue its operation even with the permanently faulty link. In order to minimize the impact of using the identity operators in critical links, we proposed a method of identifying the critical links and also a modified mapping algorithm. The algorithm would map the original neural network to the FPNN in a way that would potentially harden it against the impact of the identity operators' activation. However, just like the approximation accuracy of FPNNs was dependent on the particular condition, it was the same with this hardening method as could be expected since the method did not rely on replicating the FPNNs' components.

Another method of recovery from permanent fault in a link that we have proposed was based on modifications of activators parameters. The method utilized changes in the θ parameters that would affect the activation function. Even though this technique improved the performance of faulty FPNNs in some cases, it also harmed them in others. Depending on the situation, this could be a way to recover from faults; however, it needs to be considered cautiously because it might introduce even more problems to the FPNN.

We have also proposed a method of detecting permanent faults in neural networks' synaptic interconnections. The method was directly applicable to FPNNs as well. The method would detect the fault by successively and repeatedly changing the weights in the network in a way that would let the sought fault effect to propagate through the network to its output. There, it could be detected by comparing the outputs influenced by the fault to the correct results. The current configuration of the changes introduced to the weights would reveal the location of a fault and possibly the value that the affected weight was stuck at.

We have also performed experiments to determine the level of robustness of the three types of FPNNs against faults causing bit-flips in affine operators' values. The bit-flip were injected into different FPNNs of all types and with different structures performing benchmarking tasks. The impact of the fault was measured. The experiments revealed the dependency of the FPNNs' robustness on their structure. Particularly on the size of their layers or, strictly speaking, the lengths of their interconnection chains. The longer the interconnection chains were, the more the particular FPNN was prone to faults.

We have also investigated the possibility of recovering from permanent fault using online partial dynamic reconfiguration. This reconfiguration application was supposed to construct the reconfiguration bitstreams on the fly inside the FPGA itself rather than using stored pre-prepared bitstreams or bitstreams provided by outside systems. While this would be a novel approach, we deemed it impractical. Rather than relying on the online reconfiguration, it would be more practical to utilize the usual partial dynamic reconfiguration to recover from fault, as many authors proposed.

6.1 Contributions

The research described in this thesis represents the following contributions:

- Introduction of three different FPNN types with a different number of affine operators, approximation accuracy, and resource consumption:
  - *Full FPNNs* are the FPNNs with the whole set of affine operators provided for approximation of the original neural network. Given that this type of FPNNs are equipped with as many affine operators as they need, they can approximate the given network accurately. However, they are also the most resource-consuming type of FPNNs, with their resource utilization potentially being multiple times higher than the resource utilization of the Reduced FPNNs.
  - *Reduced FPNNs* are the FPNNs with a reduced set of affine operators. Each link would have as many affine operators as it has preceding links directly connected to its input. These FPNNs have significantly reduced resource consumption. Compared to the Full FPNNs, their consumption could be multiple times lower, but they suffer reduced approximation accuracy. The penalty to the accuracy can be only a few percent or as significant as tens of percent. Therefore, the reduced FPNNs can provide an interesting trade-off between accuracy and resource consumption. Still, they need to be carefully considered in regards to the particular situation, the original neural network type, and its task.
  - *Light FPNNs* are the FPNNs with the lowest resource consumption but also the lowest capabilities and the lowest approximation accuracy. This makes their usage limited, and the trade-offs and possible gains need to be considered compared to the reduced FPNNs.
- *A method of detecting permanent synapses faults* was proposed in our research. The method is based on modifications of the neural network's weights in a particular way that allows a possible permanent fault to be detected by comparing the modified neural network's output data with the expected data. This method is also directly applicable for detecting permanent faults in FPNNs' links as well.
- *FPNNs hardened with TMR* applied on different implementation levels were proposed in alignment with this traditional method's proven effectiveness. *A replication-free method* method of recovering the FPNNs from permanent faults using *identity operators* was proposed. Together with this method, a modified algorithm for mapping neural networks to FPNNs that can potentially harden the FPNNs against the effect of the identity operators' activation was also proposed. Also, a method to detect which links are critical to an FPNN's function was proposed. These methods showed the potential to increase an FPNN's robustness but were not universally effective.

Therefore they present an option to consider as an alternative to replication-based techniques like the TMR in some particular situations.

- *A different replication-free method* to recover from faults in an FPNN's link was proposed. This method was based on modifications of activators' parameters that influence their activation functions. The method proved to be more disruptive than helpful, so its usage has to be carefully considered.
- *Measurements of FPNNs' robustness* against fault causing bit-flips in affine operators' values were presented. The experiments revealed the dependency of the Reduced FPNNs robustness on the sizes of their layers.

6.2 Possibilities of Future Research

The topics of this thesis surely invite future research. Further research could advance the Field Programmable Neural Network both in their capabilities to approximate neural network and their robustness and beyond. We believe further research is possible in at least these pathways:

- More replication-free methods of hardening and recovery based on parameters' modifications can be proposed. Heuristics providing guidance to these recovery methods can be introduced. Moreover, methods based on a modification of the structure together with the modification of the parameters can be investigated.
- Additional methods of FPNNs' construction and mapping neural networks to them can be introduced. In an unpublished line of our work, we have experimented with FPNNs mapped using an evolutionary algorithm, and given the results, we would encourage this line of future research.
- The methods of recovery from faults based on the partial dynamic reconfiguration proposed in multiple research efforts could be integrated as a replacement of the online reconfiguration.

Bibliography

[1] *Linux kernel FPGA Subsystem*. Available at: https://www.kernel.org/doc/html/latest/driver-api/fpga/index.html.

[2] *Project X-Ray*. Available at: https://github.com/f4pga/prjxray.

[3] Ahmadi, A., Sargolzaie, M. H., Fakhraie, S. M., Lucas, C. and Vakili, S. A Low-Cost Fault-Tolerant Approach for Hardware Implementation of Artificial Neural Networks. In: *Computer Engineering and Technology, 2009. ICCET '09. International Conference on*. Jan 2009, vol. 2, p. 93–97. DOI: 10.1109/ICCET.2009.204.

[4] Alkady, G. I., El Araby, N. A., Abdelhalim, M. B., Amer, H. H. and Madian, A. H. Dynamic fault recovery using partial reconfiguration for highly reliable FPGAs. In: *2015 4th Mediterranean Conference on Embedded Computing (MECO)*. 2015, p. 56–59. DOI: 10.1109/MECO.2015.7181865.

[5] Alpaydin, E. Multiple neural networks and weighted voting. In: *Pattern Recognition, 1992. Vol.II. Conference B: Pattern Recognition Methodology and Systems, Proceedings., 11th IAPR International Conference on*. Aug 1992, p. 29–32. DOI: 10.1109/ICPR.1992.201715.

[6] Arad, B. and El Amawy, A. Robust fault tolerant training of feedforward neural networks. In: *Circuits and Systems, 1994., Proceedings of the 37th Midwest Symposium on*. Aug 1994, vol. 1, p. 539–544 vol.1. DOI: 10.1109/MWSCAS.1994.519296.

[7] Arad, B. S. and El Amawy, A. On Fault Tolerant Training of Feedforward Neural Networks. *Neural Networks*. 1997, vol. 10, no. 3, p. 539 – 553. DOI: http://dx.doi.org/10.1016/S0893-6080(96)00089-5. ISSN 0893-6080. Available at: http://www.sciencedirect.com/science/article/pii/S0893608096000895.

[8] Bhandari, S., Pujari, S., Rai, A. and Subbaraman, S. Methodology for on the fly partial reconfiguration for computation intensive applications on FPGA. In: *2010 International Conference on Computer Applications and Industrial Electronics*. 2010, p. 597–601. DOI: 10.1109/ICCAIE.2010.5735004.

[9] Biondi, A., Balsini, A., Pagani, M., Rossi, E., Marinoni, M. et al. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In: *2016 IEEE Real-Time Systems Symposium (RTSS)*. 2016, p. 1–12. DOI: 10.1109/RTSS.2016.010.

[10] Chin, C.-T., Mehrotra, K., Mohan, C. and Rankat, S. Training techniques to obtain fault-tolerant neural networks. In: *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on.* June 1994, p. 360–369. DOI: 10.1109/FTCS.1994.315624.

[11] Deng, J., Rang, Y., Du, Z., Wang, Y., Li, H. et al. Retraining-based timing error mitigation for hardware neural networks. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2015.* March 2015, p. 593–596.

[12] Deodhare, D., Vidyasagar, M. and Sathiya Keethi, S. Synthesis of fault-tolerant feedforward neural networks using minimax optimization. *Neural Networks, IEEE Transactions on.* Sep 1998, vol. 9, no. 5, p. 891–900. DOI: 10.1109/72.712162. ISSN 1045-9227.

[13] Di Carlo, S., Gambardella, G., Indaco, M., Prinetto, P., Rolfo, D. et al. Dependable Dynamic Partial Reconfiguration with minimal area & time overheads on Xilinx FPGAS. In: *2013 23rd International Conference on Field programmable Logic and Applications.* 2013, p. 1–4. DOI: 10.1109/FPL.2013.6645549.

[14] Elsimary, H., Mashali, S. and Shaheen, S. A method for training feed forward neural network to be fault tolerant. In: *Virtual Reality Annual International Symposium, 1993., 1993 IEEE.* Sep 1993, p. 436–441. DOI: 10.1109/VRAIS.1993.380747.

[15] Elsimary, H., Mashali, S. and Shaheen, S. Generalization ability of fault tolerant feedforward neural nets. In: *Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century., IEEE International Conference on.* Oct 1995, vol. 1, p. 30–34 vol.1. DOI: 10.1109/ICSMC.1995.537728.

[16] Emmerson, M. and Damper, R. Determining and improving the fault tolerance of multilayer perceptrons in a pattern-recognition application. *Neural Networks, IEEE Transactions on.* Sep 1993, vol. 4, no. 5, p. 788–793. DOI: 10.1109/72.248456. ISSN 1045-9227.

[17] Girau, B. FPNA: Concepts and Properties. In: Omondi, A. R. and Rajapakse, J. C., ed. *FPGA Implementations of Neural Networks.* Springer US, 2006, p. 63–101. ISBN 978-0-387-28487-3. 10.1007/0-387-28487-7-3. Available at: http://dx.doi.org/10.1007/0-387-28487-7-3.

[18] Goeders, J., Gaskin, T. and Hutchings, B. Demand Driven Assembly of FPGA Configurations Using Partial Reconfiguration, Ubuntu Linux, and PYNQ. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 2018, p. 149–156. DOI: 10.1109/FCCM.2018.00032.

[19] Gorman, R. and Sejnowski, T. J. Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks.* 1988, vol. 1, no. 1, p. 75 – 89. DOI: http://dx.doi.org/10.1016/0893-6080(88)90023-8. ISSN 0893-6080. Available at: http://www.sciencedirect.com/science/article/pii/0893608088900238.

[20] Hammadi, N. C. and Ito, H. A Learning Algorithm for Fault Tolerant Feedforward Neural Networks. *IEICE Trans. Information and Systems.* 1996, vol. 80, p. 21–27.

[21] Harkin, J., Morgan, F., Hall, S., Dudek, P., Dowrick, T. et al. Reconfigurable platforms and the challenges for large-scale implementations of spiking neural networks. In: *2008 International Conference on Field Programmable Logic and Applications*. Sept 2008, p. 483–486. DOI: 10.1109/FPL.2008.4629989. ISSN 1946-147X.

[22] Haruhiko, T., Hidehiko, K. and Terumine, H. Partially weight minimization approach for fault tolerant multilayer neural networks. In: *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*. 2002, vol. 2, p. 1092–1096. DOI: 10.1109/IJCNN.2002.1007646. ISSN 1098-7576.

[23] Haruhiko, T., Hidehiko, K. and Terumine, H. Fault tolerant training algorithm for multi-layer neural networks focused on hidden unit activities. In: *Neural Networks, 2006. IJCNN '06. International Joint Conference on*. 2006, p. 1540–1545. DOI: 10.1109/IJCNN.2006.246616.

[24] Hebb, D. *The Organization of Behavior: A Neuropsychological Theory*. L. Erlbaum Associates, 2002. ISBN 9780805843002. Available at: http://books.google.cz/books?id=gUtwMochAI8C.

[25] Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*. National Academy of Sciences. april 1982, vol. 79, no. 8, p. 2554–2558. ISSN 1091-6490. Available at: http://www.pnas.org/content/79/8/2554.abstract.

[26] Hsu, Y.-M., Piuri, V. and Swartzlander, J. Time-redundant multiple computation for fault-tolerant digital neural networks. In: *Circuits and Systems, 1995. ISCAS '95., 1995 IEEE International Symposium on*. Apr 1995, vol. 2, p. 977–980 vol.2. DOI: 10.1109/ISCAS.1995.519929.

[27] Ilias, A., Papadimitriou, K. and Dollas, A. Combining Duplication, Partial Reconfiguration and Software for On-line Error Diagnosis and Recovery in SRAM-Based FPGAs. In: *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. 2010, p. 73–76. DOI: 10.1109/FCCM.2010.20.

[28] Ito, T. and Takanami, I. On fault injection approaches for fault tolerance of feedforward neural networks. In: *Test Symposium, 1997. (ATS '97) Proceedings., Sixth Asian*. Nov 1997, p. 88–93. DOI: 10.1109/ATS.1997.643927. ISSN 1081-7735.

[29] Kamiura, N., Isokawa, T. and Matsui, N. Learning based on fault injection and weight restriction for fault-tolerant Hopfield neural networks. In: *Defect and Fault Tolerance in VLSI Systems, 2004. DFT 2004. Proceedings. 19th IEEE International Symposium on*. Oct 2004, p. 339–346. DOI: 10.1109/DFTVS.2004.1347858. ISSN 1550-5774.

[30] Kamiura, N., Taniguchi, Y., Isokawa, T. and Matsui, N. An improvement in weight-fault tolerance of feedforward neural networks. In: *Test Symposium, 2001. Proceedings. 10th Asian*. 2001, p. 359–364. DOI: 10.1109/ATS.2001.990309. ISSN 1081-7735.

[31] Kastil, J., Straka, M., Miculka, L. and Kotasek, Z. Dependability Analysis of Fault Tolerant Systems Based on Partial Dynamic Reconfiguration Implemented into FPGA. In: *2012 15th Euromicro Conference on Digital System Design*. 2012, p. 250–257. DOI: 10.1109/DSD.2012.40.

[32] Kohonen, T. *Self-organization and associative memory*. Springer-Verlag, 1984. Springer series in information sciences. ISBN 9783540121657. Available at: http://books.google.cz/books?id=LYZQAAAAMAAJ.

[33] Krma, M. Akcelerace neuronovch st v FPGA. *Master's thesis, Faculty of Information Technology,Brno University of Technology; Brno*. Fakulta informanch technologi, Vysok uen technick v Brn. 2013. Available at: https://wis.fit.vutbr.cz/FIT/st/rp.php/rp/2013/DP/15754.pdf.

[34] Langenbach, U., Wiehler, S. and Schubert, E. Evaluation of a declarative Linux kernel FPGA manager for dynamic partial reconfiguration. In: *2017 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*. 2017, p. 13–18. DOI: 10.1109/FPGA4GPC.2017.8008960.

[35] Latif Shabgahi, G., Hirst, A. and Bennett, S. A novel family of weighted average voters for fault-tolerant computer control systems. In: *European Control Conference (ECC), 2003*. Sept 2003, p. 642–646.

[36] Lojda, J., Panek, R., Podivinsky, J., Cekan, O., Krcma, M. et al. Hardening of Smart Electronic Lock Software against Random and Deliberate Faults. In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 2020, p. 680–683. DOI: 10.1109/DSD51259.2020.00110.

[37] Mahdiani, H. R., Fakhraie, S. M. and Lucas, C. Relaxed Fault-Tolerant Hardware Implementation of Neural Networks in the Presence of Multiple Transient Errors. *IEEE Transactions on Neural Networks and Learning Systems*. Aug 2012, vol. 23, no. 8, p. 1215–1228. DOI: 10.1109/TNNLS.2012.2199517. ISSN 2162-237X.

[38] Mahmoud, D. G., Alkady, G. I., Amer, H. H., Daoud, R. M., Adly, I. et al. Fault secure FPGA-based TMR voter. In: *2018 7th Mediterranean Conference on Embedded Computing (MECO)*. 2018, p. 1–4. DOI: 10.1109/MECO.2018.8406016.

[39] Mansour, W., Velazco, R., Ayoubi, R., Falou, W. E. and Ziade, H. Fault-tolerance capabilities of a software-implemented Hopfield Neural Network. In: *Communications and Information Technology (ICCIT), 2013 Third International Conference on*. June 2013, p. 205–208. DOI: 10.1109/ICCITechnology.2013.6579550.

[40] McCulloch, W. and Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*. Springer New York. 1943, vol. 5, p. 115–133. ISSN 0092-8240. 10.1007/BF02478259. Available at: http://dx.doi.org/10.1007/BF02478259.

[41] Miculka, L. and Kotasek, Z. Generic partial dynamic reconfiguration controller for transient and permanent fault mitigation in fault tolerant systems implemented into FPGA. In: *17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*. 2014, p. 171–174. DOI: 10.1109/DDECS.2014.6868784.

[42] Miculka, L., Straka, M. and Kotasek, Z. Methodology for Fault Tolerant System Design Based on FPGA into Limited Redundant Area. In: *2013 Euromicro Conference on Digital System Design*. 2013, p. 227–234. DOI: 10.1109/DSD.2013.33.

[43] Minsky, M. and Papert, S. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969. ISBN 9780262630221. Available at: http://books.google.cz/books?id=4e5wPAAACAAJ.

[44] Moussa, M., Areibi, S. and Nichols, K. On the Arithmetic Precision for Implementing Back-Propagation Networks on FPGA: A Case Study. In: Omondi, A. R. and Rajapakse, J. C., ed. *FPGA Implementations of Neural Networks*. Springer US, 2006, p. 37–61. ISBN 978-0-387-28487-3. 10.1007/0-387-28487-7_2. Available at: http://dx.doi.org/10.1007/0-387-28487-7_2.

[45] Munakata, T. Neural Networks: Fundamentals and the Backpropagation Model. In: Munakata, T., ed. *Fundamentals of the New Artificial Intelligence*. Springer London, 2007, p. 7–36. Texts in Computer Science. ISBN 978-1-84628-839-5. 10.1007/978-1-84628-839-5–2. Available at: http://dx.doi.org/10.1007/978-1-84628-839-5--2.

[46] Munakata, T. Neural Networks: Other Models. In: Munakata, T., ed. *Fundamentals of the New Artificial Intelligence*. Springer London, 2007, p. 41–58. Texts in Computer Science. ISBN 978-1-84628-839-5. 10.1007/978-1-84628-839-5–3. Available at: http://dx.doi.org/10.1007/978-1-84628-839-5--3.

[47] Nelder, J. A. and Mead, R. A Simplex Method for Function Minimization. *The Computer Journal*. 1965, vol. 7, no. 4, p. 308–313. DOI: 10.1093/comjnl/7.4.308. Available at: http://comjnl.oxfordjournals.org/content/7/4/308.abstract.

[48] Pagani, M., Balsini, A., Biondi, A., Marinoni, M. and Buttazzo, G. A Linux-based support for developing real-time applications on heterogeneous platforms with dynamic FPGA reconfiguration. In: *2017 30th IEEE International System-on-Chip Conference (SOCC)*. 2017, p. 96–101. DOI: 10.1109/SOCC.2017.8226015.

[49] Panek, R., Lojda, J., Podivinsky, J. and Kotasek, Z. Partial Dynamic Reconfiguration in an FPGA-based Fault-Tolerant System: Simulation-based Evaluation. In: *2018 IEEE East-West Design & Test Symposium (EWDTS)*. 2018, p. 1–6. DOI: 10.1109/EWDTS.2018.8524728.

[50] Panek, R., Lojda, J., Podivinsky, J. and Kotasek, Z. Reliability Analysis of Reconfiguration Controller for FPGA–Based Fault Tolerant Systems: Case Study. In: *2020 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. 2020, p. 1–4. DOI: 10.1109/VLSI-DAT49148.2020.9196269.

[51] Panek, R., Lojda, J., Podivinsky, J. and Kotasek, Z. Reliability Analysis of the FPGA Control System with Reconfiguration Hardening. In: *2021 24th Euromicro Conference on Digital System Design (DSD)*. 2021, p. 553–556. DOI: 10.1109/DSD53832.2021.00089.

[52] Phatak, D. and Koren, I. Fault tolerance of feedforward neural nets for classification tasks. In: *Neural Networks, 1992. IJCNN., International Joint Conference on*. Jun 1992, vol. 2, p. 386–391 vol.2. DOI: 10.1109/IJCNN.1992.226957.

[53] Phatak, D. and Koren, I. Complete and partial fault tolerance of feedforward neural nets. *Neural Networks, IEEE Transactions on*. Mar 1995, vol. 6, no. 2, p. 446–456. DOI: 10.1109/72.363479. ISSN 1045-9227.

[54] Phatak, D. and Tchernev, E. Synthesis of fault tolerant neural networks. In: *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*. 2002, vol. 2, p. 1475–1480. DOI: 10.1109/IJCNN.2002.1007735. ISSN 1098-7576.

[55] Podivinsky, J., Cekan, O., Lojda, J. and Kotásek, Z. Verification of Robot Controller for Evaluating Impacts of Faults in Electro-Mechanical Systems. In: *2016 Euromicro Conference on Digital System Design (DSD)*. 2016, p. 487–494. DOI: 10.1109/DSD.2016.38.

[56] Podivinsky, J., Lojda, J., Cekan, O. and Kotasek, Z. Evaluation Platform for Testing Fault Tolerance Properties: Soft-core Processor-Based Experimental Robot Controller. In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. 2018, p. 229–236. DOI: 10.1109/DSD.2018.00051.

[57] Podivinsky, J., Lojda, J., Cekan, O., Panek, R. and Kotasek, Z. Reliability Analysis and Improvement of FPGA-Based Robot Controller. In: *2017 Euromicro Conference on Digital System Design (DSD)*. 2017, p. 337–344. DOI: 10.1109/DSD.2017.15.

[58] Podivinsky, J., Lojda, J. and Kotasek, Z. An Experimental Evaluation of Fault-Tolerant FPGA-Based Robot Controller. In: *2018 IEEE East-West Design & Test Symposium (EWDTS)*. 2018, p. 1–7. DOI: 10.1109/EWDTS.2018.8524627.

[59] Podivinsky, J., Lojda, J. and Kotasek, Z. Extended Reliability Analysis of Fault-Tolerant FPGA-based Robot Controller. In: *2019 IEEE Latin American Test Symposium (LATS)*. 2019, p. 1–4. DOI: 10.1109/LATW.2019.8704554.

[60] Podivinsky, J., Lojda, J., Panek, R., Cekan, O., Krcma, M. et al. Evaluation Platform For Testing Fault Tolerance: Testing Reliability of Smart Electronic Locks. In: *2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)*. 2020, p. 1–4. DOI: 10.1109/LASCAS45839.2020.9068977.

[61] Prechelt, L. P. and Informatik, F. F. — *A Set of Neural Network Benchmark Problems and Benchmarking Rules*. Universitat Karlsruhe; 76128 Karlsruhe, Germany, 1994.

[62] Psarakis, M. and Apostolakis, A. Fault tolerant FPGA processor based on runtime reconfigurable modules. In: *2012 17th IEEE European Test Symposium (ETS)*. 2012, p. 1–6. DOI: 10.1109/ETS.2012.6233007.

[63] Richards, W., Seung, H. S. and Pickard, G. Neural voting machines. *Neural Networks*. 2006, vol. 19, no. 8, p. 1161 – 1167. DOI: http://dx.doi.org/10.1016/j.neunet.2006.06.006. ISSN 0893-6080. Neurobiology of Decision MakingNeurobiology of Decision Making. Available at: http://www.sciencedirect.com/science/article/pii/S0893608006001511.

[64] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*. november 1958, vol. 65, no. 6, p. 386–408.

[65] Rumelhart, D., McClelland, J. and California, S. D. P. R. G. University of. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Foundations*. Mit Press, 1986. Computational Models of Cognition and Perception. ISBN 9780262680530. Available at: http://books.google.cz/books?id=eFPqqMBK-p8C.

[66] Ruospo, A., Gavarini, G., Bragaglia, I., Traiola, M., Bosio, A. et al. Selective Hardening of Critical Neurons in Deep Neural Networks. In: *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. 2022, p. 136–141. DOI: 10.1109/DDECS54261.2022.9770168.

[67] Rusiecki, A. Fault tolerant feedforward neural network with median neuron input function. *Electronics Letters*. May 2005, vol. 41, no. 10, p. 603–605. DOI: 10.1049/el:20058169. ISSN 0013-5194.

[68] Sequin, C. and Clay, R. Fault tolerance in artificial neural networks. In: *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. June 1990, p. 703–708 vol.1. DOI: 10.1109/IJCNN.1990.137651.

[69] Shuler, R. L., Bhuva, B. L., O'Neill, P. M., Gambles, J. W. and Rezgui, S. Comparison of Dual-Rail and TMR Logic Cost Effectiveness and Suitability for FPGAs With Reconfigurable SEU Tolerance. *IEEE Transactions on Nuclear Science*. 2009, vol. 56, no. 1, p. 214–219. DOI: 10.1109/TNS.2008.2010320.

[70] Steinbuch, K. Die Lernmatrix. *Biological Cybernetics*. 1961, vol. 1, no. 1, p. 36–45.

[71] Straka, M., Kastil, J. and Kotasek, Z. Fault Tolerant Structure for SRAM-Based FPGA via Partial Dynamic Reconfiguration. In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. 2010, p. 365–372. DOI: 10.1109/DSD.2010.12.

[72] Straka, M., Kastil, J. and Kotasek, Z. Generic partial dynamic reconfiguration controller for fault tolerant designs based on FPGA. In: *NORCHIP 2010*. 2010, p. 1–4. DOI: 10.1109/NORCHIP.2010.5669477.

[73] Sum, J., Leung, C. sing and Hsu, L. Fault tolerant learning using Kullback-Leibler divergence. In: *TENCON 2007 - 2007 IEEE Region 10 Conference*. Oct 2007, p. 1–4. DOI: 10.1109/TENCON.2007.4429073.

[74] Szurman, K. and Kotasek, Z. Coarse-Grained TMR Soft-Core Processor Fault Tolerance Methods and State Synchronization for Run-Time Fault Recovery. In: *2019 IEEE Latin American Test Symposium (LATS)*. 2019, p. 1–4. DOI: 10.1109/LATW.2019.8704639.

[75] Szurman, K. and Kotasek, Z. Run-Time Reconfigurable Fault Tolerant Architecture for Soft-Core Processor NEO430. In: *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. 2019, p. 1–4. DOI: 10.1109/DDECS.2019.8724636.

[76] Szurman, K., Miculka, L. and Kotasek, Z. State Synchronization after Partial Reconfiguration of Fault Tolerant CAN Bus Control System. In: *2014 17th Euromicro Conference on Digital System Design*. 2014, p. 704–707. DOI: 10.1109/DSD.2014.103.

[77] Szurman, K., Miculka, L. and Kotasek, Z. Towards a state synchronization methodology for recovery process after partial reconfiguration of fault tolerant systems. In: *2014 9th International Conference on Computer Engineering & Systems (ICCES)*. 2014, p. 231–236. DOI: 10.1109/ICCES.2014.7030963.

[78] Takase, H., Kita, H. and Hayashi, T. Weight minimization approach for fault tolerant multi-layer neural networks. In: *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on*. 2001, vol. 4, p. 2656–2660 vol.4. DOI: 10.1109/IJCNN.2001.938789. ISSN 1098-7576.

[79] Takase, H., Shinogi, T., Hayashi, T. and Kita, H. Evaluation function for fault tolerant multi-layer neural networks. In: *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*. 2000, vol. 3, p. 521–526 vol.3. DOI: 10.1109/IJCNN.2000.861361. ISSN 1098-7576.

[80] Tan, Y. and Nanya, T. Fault-tolerant back-propagation model and its generalization ability. In: *Neural Networks, 1993. IJCNN '93-Nagoya. Proceedings of 1993 International Joint Conference on*. Oct 1993, vol. 3, p. 2516–2519 vol.3. DOI: 10.1109/IJCNN.1993.714236.

[81] Taniguchi, Y., Kamiura, N., Hata, Y. and Matsui, N. Activation function manipulation for fault tolerant feedforward neural networks. In: *Test Symposium, 1999. (ATS '99) Proceedings. Eighth Asian*. 1999, p. 203–208. DOI: 10.1109/ATS.1999.810751. ISSN 1081-7735.

[82] Velazco, R., Mansour, W., Pancher, F., Marques Costa, G., Sohier, D. et al. Improving SEU Fault Tolerance Capabilities of a Self-Converging Algorithm. *Nuclear Science, IEEE Transactions on*. Aug 2012, vol. 59, no. 4, p. 818–823. DOI: 10.1109/TNS.2012.2188303. ISSN 0018-9499.

[83] Yang, J. and Keezer, D. C. A Framework for Design of Self-Repairing Digital Systems. In: *2019 IEEE International Test Conference (ITC)*. 2019, p. 1–10. DOI: 10.1109/ITC44170.2019.9000155.

[84] Zarafshan, F., Latif Shabgahi, G. and Karimi, A. Notice of Retraction A novel weighted voting algorithm based on neural networks for fault-tolerant systems. In: *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*. July 2010, vol. 9, p. 135–139. DOI: 10.1109/ICCSIT.2010.5565122.

[85] Zhou, Z.-H., Chen, S.-F. and Chen, Z.-Q. Improving tolerance of neural networks against multi-node open fault. In: *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on*. 2001, vol. 3, p. 1687–1692 vol.3. DOI: 10.1109/IJCNN.2001.938415. ISSN 1098-7576.

Related Papers

Paper A

Mapping trained neural networks to FPNNs

M. Krcma, J. Kastil and Z. Kotásek, „Mapping Trained Neural Networks to FPNNs," *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 2015, pp. 157-160, doi: 10.1109/DDECS.2015.50.

Mapping trained neural networks to FPNNs

Martin Krcma, Jan Kastil, Zdenek Kotasek
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
Email: ikrcma@fit.vutbr.cz, ikastil@fit.vutbr.cz, kotasek@fit.vutbr.cz

***Abstract*—This paper introduces a set of methods for mapping the trained neural networks into the lighted grid structured Field Programmable Neural Networks without the use of a training data set. These methods use information obtained from original neural networks such as a network structure, connection weights and biases. The principles of these mapping methods are described and the used grid FPNNs are explained. The results of experiments are presented and summarized.**

I. Introduction

The concept of the Field Programmable Neural Arrays (FPNAs) [1] is meant to simplify the implementation of artificial neural networks in FPGAs. The simplification originates from its main feature - a strongly customizable structure which makes it possible to share a lot of resources between the original synaptic connections due to a more simple interconnection model.

A. FPNAs

The FPNAs use two different types of units called *neural resources* for approximation of the neurons and the weighted connections (synapses) of the original neural network.

An *activator* is the first type of neural resource. Activators implement the functionality of neurons. The activator collects a potential and computes an activation function. The other type of neural resource is called *link*. The links apply the weight multiplication to incoming data. By doing this it implements the synapses and serves as the interconnection of activators. But unlike the traditional neural networks, the structure of connections is much more flexible. The FPNAs do not prescribe any mandatory type of connection model. It makes it fully optional. In the FPNAs it is not necessary to connect only activators. It is possible to connect the links as well and to chain them. This feature allows us to construct new types of connection models. For example, it is possible to connect activators with a sequence of a few links and make all original connections go through this sequence. Therefore, one link in the sequence approximates a number of original synaptic connections. The links are shared between a set of original connections. This sharing leads to spare hardware resources of FPGAs. Moreover, by using this feature it enables us to construct the networks with a very simple interconnection and make its structure suitable for FPGAs.

When the connection model is designed and concrete parameters of neural resources are assigned, the resulting object is called Field Programmable Neural Network (FPNN) [2]. It is one of the possible instances of an FPNA. One FPNA can be interconnected and parametrized in many ways, so it is possible to create many FPNNs from one FPNA.

B. Grid FPNNs

In order to obtain as many resources saving FPNNs as possible, we have designed a special type of the grid FPNNs which became the core of our work [5][6]. An example of one layer of this type of FPNN is illustrated in Fig. 1. The circles represent activators, the large arrows represent links and the thin arrows represent connections between the neural resources. The orientation of the connection arrows show the way of the passing data. As the picture illustrates, there is only one link on the output of every activator. The link realizes the connection to another layer. It is directly connected to one successive activator. The connection to the other activators goes through the sequence of links within the whole layer. There are two sequences going in the opposite direction and realizing most of the connection. Together they are called *interconnection chain*. Every FPNN layer with more than one activator has an interconnection chain. The whole connection is realized by the $layer_size + 2 \times (layer_size - 1)$ links only. Synaptic connections are implemented as a sequence of multiplying by the weight of each link.

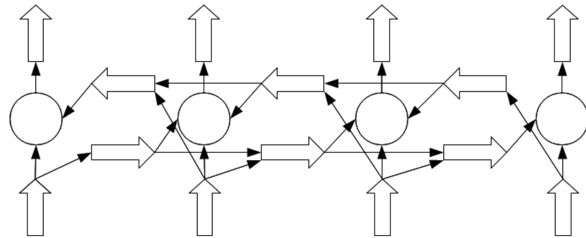

Fig. 1. A grid FPNN composed of activators (circles), links (thick arrows) and data connections (thin arrows)

This kind of structure relatively matches the structure of an interconnection bus in FPGAs. It consists of a lot of shared resources with a very simple local interconnection. Accordingly it is suitable for the implementation in FPGAs. In order to obtain even more resource savings we have changed the original definition [1] and limited the number of links affine operators (the weights) to one. Thus, every link (a member of the set of all links E) has an affine operator performing the multiplication of its input value x by a constant parameter W (1)(2). This single parameter is common for every approximated synapse. By doing this the universal multiplier in the link is turned into a less expensive constant multiplier. We call this type of link as *light link* and the grid FPNN using only light links as *light grid* FPNN.

$$\forall e \in E(\exists \alpha_e); \alpha_e : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \quad (1)$$

$$\alpha_e = W_e \times x_e; W_e, x_e \in \mathbb{R} \quad (2)$$

In the light grid FPNN, the original synapse $w \in S$, S is a set of original synapses weights, is approximated with value w' computed (4) as a product of all W parameters of all links in a sequence approximating the synapse determined by seq function (3).

$$seq : S \rightarrow E^n, n \geq 1 \tag{3}$$

$$w' = \prod_{e \in seq(w)} W_e \tag{4}$$

We have used the grid FPNNs for the implementation of basic feedforward layered neural networks for classification tasks. We have constructed an FPNN from already trained neural networks and tried to map these networks to FPNNs as precisely as possible. For mapping purposes we have used only the information coming from an original neural network, such as the network structure, weights and bias values. Our goal was to avoid the use of training data and thus be capable of creating the FPNNs from the networks, the training data of which were already lost or are not easily available.

II. Experiments

We have experimented with classification tasks from the Proben1 neural networks benchmarking data set [3]. The experiments have been programmed in Python of version 3 and performed on personal computers. All the presented results were obtained during experiments with a diabetes8 task from the Proben1. The FPNN has eight inputs, sixteen activators in the hidden layer and two activators in the output layer.

A. Basic mapping method

As all the links W parameters in a sequence have to be known for the approximation of a synapse and as prefixes of the sequence can approximate another synapses, it is needed to start the mapping from the beginning of every sequence and with the following steps moved further and further.

The weight approximation (4) can be reformulated as multiplying the partial product of the sequence prefix by its last l links W parameter (5). As the prefix value is already known from previous steps of mapping, the W_l^w is needed to be determined in an actual step. It is computed as the division of original synapse w weight and the prefix value (6). All synapses approximated by the link l are placed in a set S_l.

$$w' = \left(\prod_{e \in seq(w) \setminus \{l\}} W_e \right) \times W_l^w ; w \in S_l \tag{5}$$

$$W_l^w = \frac{w}{\prod_{e \in seq(w) \setminus \{l\}} W_e} ; w \in S_l \tag{6}$$

As one link can be placed in more than one sequence and $|S_l| \geq 1$, there is a whole set P_l ($|P_l| = |S_l|$) of W_l^w computed parameters for it. However, in the case of light grid FPNN, the link disposes a single W parameter. Thus, there is a need for compromise. In the basic method (called ARIT) an arithmetic average has been used.

B. Measuring

Since the FPNNs are seen in this paper as an FPGA suitable for approximation of neural networks, the quality of results was measured as a difference between a neural network output classification and an FPNN output classification. The number of identically classified data vectors and differently classified data vectors were measured. The representative value was the percentage rate of match.

C. The accuracy

During our experiments with the grid FPNNs we have found that FPNNs with two inputs and one output have been mapped with 100% accuracy. This is not surprising considering the grid FPNNs structure. There are two inputs and two link sequences in a hidden layer to connect them with each activator, so there is no need for sharing the links between the synapses (there is one link in the sequence for the approximation of each synapse) and it is possible for an approximation to be accurate. Secondly, there is only one output, thus there is no interconnection chain, no sharing links between synapses and approximation is accurate.

Next, we have found that the accuracy of differently organized FPNNs was not very high. If there are more than two inputs and one output, the sharing comes into place having a strong negative influence on the accuracy. In different kinds of approximated neural networks we usually did not get over the 70% rate of match level. Usually we reached an over 60% rate of match.

Because of these results, we have decided to find a mapping method capable of achieving a satisfying accuracy without the use of structural changes. Our goal has been at least 90% rate of match. in order to obtain these results two sets of new methods have been designed.

III. Methods based on weighted average

These methods are based on a weighted average (7) and differ in a way of computation of the weights v. The computation is based on information the FPNN provides.

$$W_l = \frac{\sum v_i * p_i}{\sum v_i} ; l \in E, p_i \in P_l, i \in \langle 1, |P_l| \rangle \tag{7}$$

The first method computes the weights from a distance between an actual computed link and a source activator of computing connection in a previous layer. The weight is determined as a count of links between the actual link and the source activator. Thus, longer connections have a bigger effect to a final value of links weight. Another version of the method uses a reciprocal value of the distance between the actual link and the source activator. Shorter connection had a bigger effect in this case. We call this method DIST_DP for direct proportion and DIST_IP for inverse proportion. Computation uses $dist$ and $clos$ functions. If a sequence L is composed of n links, $L = \{l_1..l_n\}, n \geq 1$, then the functions can be defined as (8)(9). By using these functions, the weight computation can be performed as (10)(11).

$$\forall e_m \in L : dist(e_m) = m \tag{8}$$

$$\forall e_m \in L : clos(e_m) = (n - m) + 1 \tag{9}$$

$$v_i = dist(p_i); p_i \in P_l, i \in \langle 1, |P_l| \rangle \quad (10)$$

$$v_i = clos(p_i); p_i \in P_l, i \in \langle 1, |P_l| \rangle \quad (11)$$

The second method is based on the value of the original synapses weight. The value is directly used as a weight (12). Synapses with a bigger weight have a greater effect on a link weight. Another version of this method uses an inverse proportion to make connections with a smaller weight to have a bigger effect on a result (13). We call this method WEIG_DP for direct proportion and WEIG_IP for inverse proportion.

$$v_i = w_i; w \in S_l, i \in \langle 1, |S_l| \rangle \quad (12)$$

$$v_i = \frac{1}{w_i}; w \in S_l, i \in \langle 1, |S_l| \rangle \quad (13)$$

The third method is based on the value of the sequence prefix. The value is directly used as a weight (14). Synapses with a higher prefix value have a greater effect on a result. Another version of this method uses an inverse proportion to make synapses with a lower value of a prefix to have a bigger effect on a result (15). We call this method PROD_DP for direct proportion and PROD_IP for inverse proportion.

$$v_i = \left(\prod_{e \in seq(w) \setminus \{l\}} W_e \right); w_i \in S_l, i \in \langle 1, |S_l| \rangle \quad (14)$$

$$v_i = \frac{1}{\left(\prod_{e \in seq(w) \setminus \{l\}} W_e \right)}; w_i \in S_l, i \in \langle 1, |S_l| \rangle \quad (15)$$

The last method is based on the usage of a position of a computed approximation value p_i in ascending ordered set of all approximation values. For completeness, the inverse proportion is used as well. We call this method PVAL_DP for direct proportion and PVAL_IP for inverse proportion.

Table I contains the results of all methods on the testing data set. The column *Method* contains the name of the method and the column *Data set* specifies the training or testing data set. The column *Match* contains the number of input vectors accordingly classified by both the FPNN and the original neural network. The column *Mismatch* contains the number of differently classified vectors, and the column *Rate* contains the rate of matches.

Method	Match	Mismatch	Rate [%]
ARIT	250	133	65.274
DIST_DP	237	146	61.879
DIST_IP	250	133	65.274
WEIG_DP	250	133	65.274
WEIG_IP	250	133	65.274
PROD_DP	251	132	65.535
PROD_IP	249	134	65.013
PVAL_DP	250	133	65.274
PVAL_IP	250	133	65.274

TABLE I. THE METHODS COMPARISON

As the table shows, the best results were achieved with the PROD_DP method with a 65.535% rate of match. However, all the results are almost identical and there is hardly any improvement on the results according to the basic ARIT method. In an effort to gain a better improvement, combinations of methods were tested.

A. Combinations of methods

All the possible combinations of two, three or all four weighted methods were used. A resulting weight v_i was computed as an addition of weights computed by all methods in a combination. The results are summarized in Table II. The results are ordered from the best to the worst. Only the results better than 60% are listed. Table III contains the concrete values of the gained improvement and these tables contain only those methods which achieved an improvement over the PROD_DP (the best simple method). The column *Increase* contains the percentage increase of the rate of the match.

Combination of methods	Match	Mismatch	Rate [%]
PVAL_DP, PROD_IP, DIST_DP	283	100	73.890
PVAL_DP, PROD_IP, WEIG_IP, DIST_DP	280	103	73.107
PVAL_DP, DIST_DP	279	104	72.845
PVAL_DP, WEIG_IP, DIST_DP	277	106	72.323
PVAL_IP, PROD_DP, WEIG_IP	259	124	67.624
PVAL_IP, PROD_DP	253	130	66.057
PVAL_IP, PROD_DP, WEIG_DP, DIST_DP	252	131	65.796
PVAL_DP, PROD_DP, WEIG_IP, DIST_IP	251	132	65.535
PVAL_IP, PROD_IP, WEIG_IP, DIST_IP	250	133	65.274
PVAL_DP, PROD_DP, WEIG_DP, DIST_IP	249	134	65.013
WEIG_IP, DIST_DP	237	146	61.879
PVAL_IP, PROD_IP, DIST_DP	230	153	60.052

TABLE II. THE COMPARSION OF THE COMPLEX METHODS

Combination of methods	Rate [%]	Increase [%]
PVAL_DP, PROD_IP, DIST_DP	73.89033	8.35533
PVAL_DP, PROD_IP, WEIG_IP, DIST_DP	73.10704	7.57204
PVAL_DP, DIST_DP	72.84595	7.31095
PVAL_DP, WEIG_IP, DIST_DP	72.32375	6.78875
PVAL_IP, PROD_DP, WEIG_IP	67.62402	2.08902
PVAL_IP, PROD_DP	66.05744	0.52244
PVAL_IP, PROD_DP, WEIG_DP, DIST_DP	65.79634	0.26134
PVAL_DP, PROD_DP, WEIG_IP, DIST_IP	65.53524	0.00024

TABLE III. THE IMPROVEMENT OF THE COMPLEX METHODS

The combination of methods PVAL_DP, PROD_DP, WEIG_DP and DIST_DP gained the best results with a 73.890% rate. The best achieved improvement was 8.35533%. In eight methods, improvements were achieved.

IV. METHODS BASED ON OPTIMIZATION ALGORITHM

Up till this point only the weights were mapped. The neurons biases were directly transferred into activators without any modification. In an effort to increase results even more, we have developed two methods of mapping, including a biases modification. Both methods are based on the Nelder-Mead optimization simplex algorithm [4]. The error function minimized with this algorithm is the sum of the differences between the activators output and the original neurons output. The methods differs in the way a simplex is constructed and the way the error function is applied. The points of the simplex in both methods are made of a different set of links weight and activator biases. The FPNNs created using methods mentioned in the previous section are used as the initial points. The FPNNs are listed in Table III plus the basic ARIT method. Thus, these new methods serve as an extension of both the simple and complex methods.

A. Computation layer by layer

The first method creates the points of simplex from a set of link weights and activator biases in one layer. Every layer is optimized separately. The equation (16) is the error function. The T is the testing data set and the H is the set of activators (neurons) in a hidden layer. o_i^t is the activator output on the input vector t and d_i^t is the neuron output. This extension is called TLAY. Table IV summarizes the achieved improvement over simple methods.

$$Err = \sum_{t \in T} \sum_{i \in H} o_i^t - d_i^t \tag{16}$$

Combination of methods	Rate [%]	Increase [%]
PVAL_IP, PROD_DP, WEIG_IP, TLAY	68.66840	3.13340
PVAL_DP, WEIG_IP, DIST_DP, TLAY	67.62402	2.08902
PVAL_DP, PROD_DP, WEIG_DP, DIST_DP, TLAY	67.10182	1.56682
PVAL_DP, PROD_IP, WEIG_IP, DIST_DP, TLAY	66.31853	0.78353
PVAL_IP, PROD_DP, TLAY	66.05744	0.52244
PVAL_DP, PROD_DP, WEIG_IP, DIST_IP, TLAY	65.79634	0.26134

TABLE IV. THE IMPROVEMENT OVER THE SIMPLE METHODS

It can be observed that the greatest improvement of 3.13340% over simple methods this extension was achieved with the combinations of PVAL_IP, PROD_DP and WEIG_IP methods. This improvement is smaller compared with only using the combinations of the methods without this extension.

B. Computation activator by activator

The second method creates the points of simplex from one activator bias and from the weights of all links in sequences connected to that activator. Every activator is optimized separately. The final weights of links are computed as a weighted average of the values obtained from all activator optimizations. The weight in the average is determined as the distance between an optimized activator and a computed link. The links weight is computed after all optimizations are done. The equation (17) is the error function. The T is the testing data set, o_i^t is the activator output on the input vector t and d_i^t is the neuron output. This extension is called TACT. Table V summarizes the achieved improvement over the simple methods.

$$Err = \sum_{t \in T} o_i^t - d_i^t \tag{17}$$

Combination of methods	Rate [%]	Increase [%]
PVAL_DP, DIST_DP, TACT	75.45691	9.92191
PVAL_DP, WEIG_IP, DIST_DP, TACT	75.19582	9.66082
PVAL_IP, PROD_DP, WEIG_IP, TACT	72.32375	6.78875
PVAL_IP, PROD_DP, TACT	67.62402	2.08902
PVAL_DP, PROD_IP, DIST_DP, TACT	67.36292	1.82792
ARIT, TACT	65.53524	0.00024

TABLE V. THE IMPROVEMENT OVER THE SIMPLE METHODS

As the table shows, the best results this expansion achieved with the combination of the methods PVAL_DP and DIST_DP with a 9.92191% improvement over the simple methods. This is the best result achieved with all methods, combinations and extensions.

V. CONCLUSIONS AND FUTURE RESEARCH

We have developed a set of mapping methods based on the weighted average and different kinds of weight determination. We have combined these methods into a new set of complex methods. We have extended these methods using the Nelder-Mead optimization algorithm. The combinations of methods gained an 8.35533% improvement over the simple methods. The best result all the methods have gained is a 75.45691% rate of match with the combination of PVAL_DP, DIST_DP methods and the TACT extension.

Some of the designed combinations of the methods have achieved an improvement over the simple methods, while some have produced the same results as the simple methods. Also the extensions made an increase of rate of match with some combinations of the methods.

In our future research, we are going to perform significantly more experiments with other neural networks. We are going to examine other mapping methods and optimization algorithms, develop methods of using redundancy to increase the approximation accuracy and examine the light grid FPNNs enhanced with links using more than one affine operator in place of light links suffering with a high approximation error. Beside this, we are going to focus on fault tolerant properties of FPNNs and examine the possibilities of using redundancy on the neural resources level in order to increase the fault tolerance. We are also going to test special algorithms performing the mapping of neural networks to FPNNs and establishing a redundancy free fault tolerance of selected neural resources.

ACKNOWLEDGMENT

This work was supported by the following projects: National COST LD12036 -"Methodologies for Fault Tolerant Systems Design Development, Implementation and Verification", project Centrum excelence IT4Innovations (ED1.1.00/02.0070), EU COST Action IC1103 - MEDIAN - Manufacturable and Dependable multIcore Architectures at Nanoscale and BUT project FIT-S-14-2297.

REFERENCES

[1] Girau, B.: FPNA: Concepts and Properties. In *FPGA Implementations of Neural Networks*, editace A. R. Omondi; J. C. Rajapakse, Springer US, 2006, ISBN 978-0-387-28487-3, p. 63–101, 10.1007/0-387-28487-7-3. http://dx.doi.org/10.1007/0-387-28487-7-3

[2] Girau, B.: FPNA: Applications and Implementations. In *FPGA Implementations of Neural Networks*, editace A. R. Omondi; J. C. Rajapakse, Springer US, 2006, ISBN 978-0-387-28487-3, p. 103–136, 10.1007/0-387-28487-7-4. http://dx.doi.org/10.1007/0-387-28487-7-4

[3] Prechelt, L. P.; Informatik, F. F.: — A Set of Neural Network Benchmark Problems and Benchmarking Rules. Technical report, Universitat Karlsruhe; 76128 Karlsruhe, Germany, 1994.

[4] Nelder, J. A.; Mead, R.: A Simplex Method for Function Minimization. *The Computer Journal*, year 7, n. 4, 1965: p. 308–313, http://comjnl.oxfordjournals.org/content/7/4/308.abstract

[5] Krcma, M.: *The neural networks acceleration in FPGA*. Bachelor's thesis, Faculty of Information Technology, Brno University of Technology; Brno, 2012. https://wis.fit.vutbr.cz/FIT/st/rp.php/rp/2011/BP/13719.pdf

[6] Krcma, M.: *The neural networks acceleration in FPGA*. Master's thesis, Faculty of Information Technology, Brno University of Technology; Brno, 2014. https://wis.fit.vutbr.cz/FIT/st/rp.php/rp/2013/DP/15754.pdf

Paper B

Comparison of FPNNs models approximation capabilities and FPGA resources utilization

M. Krcma, Z. Kotasek and J. Lojda, „Comparison of FPNNs models approximation capabilities and FPGA resources utilization,“ *2017 13th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2017, pp. 125-132, doi: 10.1109/ICCP.2017.8116993.

Comparison of FPNNs Models Approximation Capabilities and FPGA Resources Utilization

Martin Krcma, Zdenek Kotasek, Jakub Lojda
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
Email: ikrcma@fit.vutbr.cz, kotasek@fit.vutbr.cz, ilojda@fit.vutbr.cz

Abstract—**This paper presents the concepts of FPNA and FPNN, used for the approximation of artificial neural networks in FPGAs and introduces derived types of these concepts used by the authors. The process of transformation of a trained artificial neural network to an FPNN is described. The diagram of the FPGA implementation is presented. The results of experiments determining the approximation capabilities of FPNNs are presented and the FPGA resources utilization are compared.**

I. Introduction

The artificial neural networks [9] are one of the important models of sofcomputing and artificial intelligence. Their structure is inspired by the structure of the human brain and they dispose of a high capability of learning and memorizing to solve various types of tasks. Basically, the goal of the artificial neural network is to learn the relation between two sets of data vectors, to generalize the relation, to determine its features and to use it for the determining the relation of the unknown vectors belonging to the same problem. This capability can be used for classification tasks, for timeseries and functional prediction, to control tasks, to image recognition, clustering and other tasks.

Neural networks are composed of a set of *neurons* computing the *activation function* over the *weighted sum* of their inputs. The neurons are interconnected with the weighted connections called *synapses*. The learning of the neural network is basically a process of setting the weights.

The networks have been implemented in various kinds of devices starting from analog computers to the most modern processors, VLSIs, graphical processing units and FPGAs. This paper deals with one of the possible implementations of artificial neural networks in FPGAs - FPNA/FPNN.

The concept of Field Programmable Neural Arrays/Networks (FPNAs/FPNNs) [1], [2] in design is meant to simplify the implementation of artificial neural networks in FPGAs by adjusting its properties to be more suitable for the implementation into their logic. The simplification originates from its main feature - a highly customizable structure which makes it possible to establish resource sharing between the original synaptic connections of the neural network. This is done by using its customizability to simplify the interconnection model. The concept were used for implementing large scale spiking networks [11], [12].

The FPNNs are not the same structures as neural networks, although they can be constructed in that way. The FPNNs represent a different model which can structurally differ from the implemented neural network. They can also have different capabilities, which means that they are not only an implementation of the neural networks, they are an approximation of neural networks as well. Since the FPNNs can be constructed in various ways and types, the approximation accuracy can be different.

The goal of this paper is to describe the types of FPNNs and compare the approximation capabilities of these types. The FPGA resources utilization of the FPNNs is compared as well.

The FPNNs were formally defined in [1], [2]. In order to follow the original definitions, the presented work is based on these definitions and on definitions derived from them. For our purposes we modified the original definitions in order to suit them to our way of using the concept. This step allowed us to use different level of the approximation accuracy. Our further definitions specify special derived types of FPNNs. Also, they allow us to describe easily the algorithms of mapping trained neural networks to FPNNs.

In our paper published at NORCAS 2015 conference [7], we described the concept of Field Programmable Neural Networks for artificial neural networks implementation in FPGA. We also presented a model of fault tolerant FPNNs and various fault tolerance improving techniques based on the model. Experimental results were also provided. Now, in this paper we describe how we continue in our research - the formal definitions of the FPNA/FPNN concept are presented. The problem of process of direct transformation of the trained neural network to FPNN together with the related algorithms are described. The goal of experiments presented in the paper is to determine the approximation capabilities of different FPNNs of the reduced and the full type, the results are described in the paper. In the earlier paper [6] we dealt with the mapping of the neural networks to FPNNs of the most simple type with a number of methods. This paper follows this work by extending it to other types of FPNN with more detailed description of the models, methods and algorithms.

The paper is organised as follows - the first section introduces the FPNA/FPNN concept. The second section deals with the problem of neural networks transformation to FPNNs and describes our transformation algorithms while the third section presents the diagrams of FPGA implementation of FPNNs. In the fourth section experiments and results are described. The last section summarizes the paper.

II. FPNN

For purposes of our research we developed a new definition of an FPNN (see Definition II.1, original definition by B. Girau [1], [2]). According to this definition, an FPNN is a structure composed of two types of units (together called *neural resources*). The units of the first type are called *activators* (the set N) and represent original neural network neurons. They perform the same actions as neurons - they iteratively gather input data into *potential*, then apply an activation function to obtain the output. The activation function is represented by the *function operator* "f" and the *iteration operator* "i" is responsible for input data processing to provide the input to the function operator.

The interconnections between activators are realized by the other unit type called *links* (the set L). The links perform approximation of the original synaptic weights (they compute the weight multiplication) according to the rest of the FPNN parameters. The actual data interconnection model is presented by an oriented graph (N, E), where E is a set of valued edges interconnecting the activators. Every edge is usually split up to a sequence of links which allows us to construct various structures. The more we split the edges into links, the more flexibility we obtain.

Definition II.1 (FPNN [1])**.** We say that structure (N, L, E, ϕ, ω) is an *FPNN* if the following statements hold true:

1) N is a set of units called *activators* that dispose of:
   a) An iterative variable t_n: $\forall n \in N : \exists t_n \in \mathbb{R}$
   b) A default value of t_n: $\forall n \in N : \exists o_n \in \mathbb{R}; t_{n_0} = o_n$
   c) A number of iterations: $\forall n \in N : \exists a_n \in \mathbb{N}$
   d) An iterative operator (x_n is an input data): $\forall n \in N : \exists i_n : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$; $t_{n_a} = i_n(t_{n_{a-1}}, x_n); a = 1..a_n$
   e) A function operator: $\forall n \in N : \exists f_n : \mathbb{R} \to \mathbb{R}$
2) L is a set of units called *links* that dispose of:
   a) A set of *link operators* $\forall l \in L : \exists A_l$: $A_l = \{\alpha_n(x) | \alpha_n(x) = W_n \times x; W_n \in \mathbb{R}; n = 1..c\}$
3) E is a set of valued oriented edges: $(m, n) \in E; m, n \in N$). The edge value is defined: $\forall e \in E : \exists W_e \in \mathbb{R}$
4) (N, E) is an oriented graph denoting the interconnection between activators.
5) ϕ is a function $E \to L^+$, so that: $\forall e \in E : \phi(e) = (l_1..l_n); l_1..l_n \in L; n > 0$
6) ω is a function $E \to L^+$, so that: $\forall e \in E : \phi(e) = (l_1..l_n); l_1..l_n \in L; \omega(e) \subseteq \phi(e); 0 < n \leq |\phi(e)|$
7) Edge-to-operator functions $\sigma_l : E \to A_l; l \in L$: $\forall e \in E \wedge \forall l \in \phi(e) : \sigma_l(e) = \alpha_l^x; \alpha_l^x \in A_l$
8) Operator determining $\psi_l : E^+ \to A_l, l \in L$: $\forall l \in L : \psi_l(e_1..e_n) = \alpha_x \Leftrightarrow \alpha_x \in A_l \wedge l \in \omega(e_1) \wedge .. \wedge l \in \omega(e_n) \wedge \sigma_l(e_1) = .. = \sigma_l(e_n) = \alpha_x$
9) A set of input nodes exists: $\exists N_i = \{n \in N \mid deg^+(n) = 0\}$ $\forall n \in N_i : i_n = \emptyset; f_n(x) = x$

The actual FPNN structure is determined by the (N, E) graph and the ϕ function. The edges are split up to the sequences of interconnected links, given by the ϕ functions which realize the interconnection between activators and the approximation of the edges weights. The edges weight approximation is determined by the ω, σ, ψ functions and realized by the *link operators*. Link operators are functions which are applied to the data passing through the links. Every link disposes of one or more link operators (the A_l sets). To determine a link operator which should be assigned to the particular edge (to the data which would originally pass through the edge) the σ functions are constructed. All the link operators in the sequence (realizing an edge) are applied to all the passing data (according to the σ functions). To establish the weight approximation, it has to be decided which links in the sequences will be used for the approximation by the construction of the ω function. The actual approximation is determined by the ψ functions which construct link operators for the assigned edges (by the ω, σ functions). This will be further explained in the section III.

To preserve the consistency with the original definition [1] we add the following: If only the graph (N, E), iteration and function operators are defined, the structure is called FPNA (Field Programmable Neural Array) [1] and it defines the whole class of possible FPNNs. Every FPNN can bee seen as an instance of some FPNA.

A. Grid FPNN

For our research purposes we developed a special type of FPNN based on the above provided definitions. *Grid FPNN* (definition II.2) is an FPNN with an enforced limitation of the structure causing it to form a grid shape. The reason for this is to make an FPNN suitable for the implementation in FPGAs due to the similarity of the grid FPNNs structure and FPGAs interconnection bus and the sharing of resources in links.

Definition II.2 (Grid FPNN)**.** We say that FPNN is the *grid FPNN* if the the following statements hold true:

1) The activators are organized into layers.
2) The two sequences of interconnected links exist in all layers composed of more than one activator. The number of links in every sequence is one less than the number of activators in the layer. The output of every link is connected to the input of the nearest activator. The sequences go in the opposite ways.
3) The output od every activator is connected only to a single link which provides the connection to the next layer. The output of the link is connected to the nearest activator and to the nearest links of one or both link sequences in the layer (which realizes connection to all other activators).

An example of a grid FPNN can be seen in Fig. 1. In the figure, the circles represent activators, wide arrows represent links and the thin arrows represent data interconnections. The orientation of the connection arrows shows the way of the passing data. The straight wide dashed/dotted arrows represent the original neural networks synapses. The thin dashed/dotted arrows represent the sequences of links approximating the particular synapses. The synapses and the particular sequences are drawn with the same line and arrow styles. As the picture

illustrates, there is only one link (called *initial*, Definition II.4) on the output of every activator which provides the connection to the following layer. It is directly connected to one successive activator in the following layer. The connection to the other activators goes through the sequence of links within the whole layer. Two sequences of the links are going the opposite ways. They are called *Interconnection sequence* (definitions II.3-II.5). Every layer with more than one activator has an interconnection sequence within.

Definition II.3. *A sequence of links* is generally a sequence of *directly interconnected* links.

Definition II.4. *An initial* is a link having no link predecessors. It has only an activator predecessor.

Definition II.5. *An interconnection sequence* is a sequence of links interconnecting activators within a layer. It is composed of two sequences going the opposite ways. The input of every link is connected to one or two preceding links, the output is always connected to the nearest activator and to the succeeding link in the sequence (if it exists).

B. Different levels of approximation

The approximation capabilities of the FPNN depend on the number of available link operators present in links and the number of edges assigned by the ω function for approximation to the links. Respectively, the ratio between the numbers of operators and assigned approximated edges is the essential parameter for the FPNN approximation abilities. The numbers can be equal. In that case, the ψ functions only determine the value approximating the given edge (the member of the multiplication sequence as described in the next subsection). Thus, every edge has its link operator counter part. In this case, the approximation of the original neural network weights (suppose that original synapses were directly transferred to edges, thus (N, E) graph is isomorphic to the original network) is accurate. We call the FPNN with these properties as *Full FPNN*.

The definition allows us to reduce the number of link operators. In that case, the ψ function is surjective and its purpose is to find a *compromise* between a number of edges mapped to one link operator. In this case, the approximation accuracy suffers from the decrease caused by *sharing* the link operators between multiple edges. However, this kind of sharing reduces the FPGA resources utilization (the main goal of the FPNN concept) since, the memories containing the link operators are smaller as well as related logic (multiplexors, possibly multipliers etc.). And the accuracy decrease does not have to be necessary critical since the neural networks are potentially robust against some weights losses. The usage of this technique of resource utilization reduction is a matter of preference and depends on concrete situation and a way of usage.

We distinguish two other types of FPNN. The *reduced* FPNN and the *light* FPNN. The meaning of light FPNN is simple - every link disposes of only one link operator ($\forall l \in L : |A_l| = 1$). In this case, the link multiplier turns into a constant multiplier which offers the highest spare of resources. However, the accuracy suffers the most.

The last type mentioned in this paper is the reduced FPNN which disposes of the same number of link operators as it has the number of direct link predecessors in the link sequences it belongs to. This type approximation capabilities are determined by an FPNN structure as the number of link operators is directly dependent on the number of existing link sequences and their interconnection. The explanation based on Fig. 1 would be the most appropriate. Consider the third (the rightmost) link of the lower part of the interconnection sequence (the sequence heading to the right). This link approximates three edges originating in the three leftmost activators. However, it has only two direct link connected predecessors. So, in the case of reduced FPNN, it would have two link operators. The first one approximates the edge originating in the third leftmost activator. Since there is no sharing of this link operator, the approximation of the edge is accurate. However, the second link operator is shared between two edges originating in the first two leftmost activators. An approximation of these edges would be hence less accurate due to sharing the link operators. But in case of the full FPNN, the approximation would be accurate since there would be three affine operators, one for every edge.

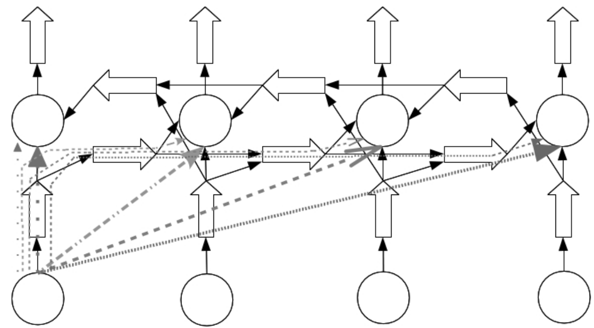

Fig. 1. Synapses (edges) approximation in a grid FPNN

Definition II.6 (Light FPNN)**.** We say that FPNN is a *light FPNN* if the folowing statemet holds true: $\forall l \in L : |A_l| = 1$.

Definition II.7 (Full FPNN)**.** We say that FPNN is a *full FPNN* if the following statement holds true:
$\forall l \in L \wedge \forall e \in E : |A_l| = |\{e | e \in \omega^{-1}(l)\}|$.

Definition II.8 (Reduced FPNN)**.** We say that FPNN is a *reduced FPNN* if the following statements hold true:

1) The edge equivalence is defined: $\forall e_1, e_2 \in E; l \in L : e_1 \equiv_l e_2 \Leftrightarrow \phi(e_1) = l_1^1..l_x l..l_n \wedge \phi(e_2) = l_1^2..l_y l..l_m; l_x = l_y$
2) $\forall l \in L$ the size of A_l is equal to the number of the equivalence classes generated by the $\equiv_l$.

III. Mapping of neural networks to FPNNs

Mapping is a process of direct transfer of an artificial neural network into an FPNN without using a training data set and without the need of learning (other works [3] deal with training of FPNNs). Mapping uses information obtainable from an original neural network such as weights, biases, activation functions and the network structure.

The first phase of mapping would be the construction of an appropriate FPNN. The first step is construction of (N, E) graph which should be (but does not have to be) isomorphic

to the original neural network. This step contain the mapping of neurons to activators - the basis functions are mapped to the iteration operators, the activation functions to the function operators and the biases to the o_n parameters. The second step is the links creation according to the intended shape of the FPNN. The next step is to assign the edges to the sequence of links (constructing the ϕ function) which specify the concrete shape of the FPNN. According to the approximation accuracy preferences, the A_l sets of link operators needed to be constructed now.

Since all data between activators will flow through the sequences of links (given by the ϕ function) interconnecting them, the data will be modified using link operators of all links in the sequences. Therefore, the σ functions have to be constructed to determine the relations between edges and the link operators.

However, not all link operators have to be used for the edges weights approximations. Some of them (or all but one) can be used as data passers which can be possibly shared between edges. For better understanding, consider an edge e, its weight $W_e = 3$ and $\phi(e) = l_1l_2l_3l_4$. Using σ functions we obtain a sequence of link operators $\alpha_1\alpha_2\alpha_3\alpha_4$ assigned to the edge. If it would be our intention to use all these operators to approximate the W_e, the approximation sequence would then be most likely composed of three operators with value of 1 and one operator with value of 3. Therefore, we would need to have an extra 1-valued operator in all the three links. Which would be easy to use, but costs more resources. I could be more wise to use other existing operators (approximating other edges) in some of links and special operators used for e approximation in others. In our case, if σ functions map the e to the operators in $l_1..l_3$ links with values of $1.5, 2, 2$, the link operator in l_4 used for approximation of the e would have the value of 0.25 (since $1.5 \times 2 \times 2 \times 0.25 = 3$). In this case, only one operator was used for the e approximation and the others were shared with other edges without influencing them (they were used for other edges approximation in the same way). To specify which link operators are used for approximation of concrete edges, the ω function is constructed. If sharing is not in our intention (for example for matter of data type accuracy which could suffer from multiple multiplications), the $\omega(e) = \phi(e)$ for all edges.

The last step is to determine the ψ functions. These functions serve for finding a compromise if more then one edges are mapped to a single link operator for approximation. If all the edges dispose of exclusive operators of their own (the full FPNN), no ψ functions are needed. In other cases (the reduced and light FPNNs) they have to be constructed. There are plenty of way of finding the compromises - arithmetic average, median, weighted average and others. We have described the results of using different compromises (ψ functions) in our paper at DDECS 2015 [6].

There are several possible ways how to determine the ω and the σ, ψ functions. Using evolution algorithms and optimization algorithms could be one of them. In this paper however we would like to describe a systematic approach of mapping trained layered feedforward neural network (perceptron like) to grid FPNNs. We suppose the (N, E) graph to be isomorphic to the original neural network and the L set and the ϕ function to be constructed to form a grid FPNN according to the definition II.2. The ω functions is constructed according to the equation 1. According to it, every edge is approximated by the link operator of the last link in the sequence given by the ϕ function. The σ functions need to be constructed to create groups of edges according to the equivalence classes generated by the $\equiv$ function from the definition II.8. The ψ functions were chosen as the arithmetic average (equation 2). The $opSeq_e$ is the sequence of link operators assigned to the edge e except the last one. The P_e is the value of the product of the link operators in the $opSeq$ set. It denotes the actual multiplication value in the last link before $\omega(e)$. According to the value and the value of the edge weight, the value A_e needed to accurate approximation is computed. In full FPNN, this value would be directly assigned to the link operator. In the presented equation, the arithmetic average is applied to all A_e values mapped to the link operator.

$$\forall e \in E : \omega(e) = l_n \Leftrightarrow \phi(e) = l_1..l_n \tag{1}$$

$$\begin{aligned}
&opSeq_e = (\alpha_l^e | \alpha_l^e = \sigma_l(e) \wedge l \in \phi(e) \setminus \omega(e)) \\
&P_e = \prod_{\alpha_l \in opSeq_e} \alpha_l \\
&A_e = \frac{W_e}{P_e} \\
&\forall l \in L : \psi_l(e_1..e_n) = \frac{\sum_{e_x \in \{e_1..e_n\}} A_e}{n}
\end{aligned} \tag{2}$$

$$\forall e \in E : A_e = \prod_{l \in \phi(e); \alpha_l^e = \sigma_l(e)} \alpha_l^e \tag{3}$$

A. Mapping algorithms

On the base of the presented principles we implemented the following algorithms which perform a mapping of trained neural network to the grid FPNN. The construction algorithm of the FPNN will not be described in explicit details in this paper, however the main idea was presented in the preceding section. The algorithms use the definitions and equations presented above as well as the declaration in Table I.

At first, the auxiliary variables have to be initialized in the **Algorithm 1**. Then, the ordered set of link sequences must be constructed using the **Algorithm 2**. The set is called $chains$ and it is constructed for each layer separately using the ϕ function and ordering the resulting link sequences by their length (ascending order). The reason why to order the sequences is that it is suitable to start mapping with the shortest sequences of the length 1 (initials - always present in the grid FPNN) and continue with longer sequences in the next steps, determining one additional operator (member of the multiplication sequence) in the sequence in the each subsequent step. This means that the links (their operators) are mapped one by one creating longer mapped sequences in each step.

In every step a new link is selected and its operators determined. In order to compute the values of the operators, it is needed to construct the groups of edges related to each link operator. Determining of these groups differs in case of

each type of FPNN. In case of reduced FPNN, the edges are separated according to the link predecessor they pass trough, as the definition specifies. The groups are constructed as the equvalence classes of the $\equiv_l$ function from the definition II.8. This is done by the **Algorithm 3**. In case of light FPNN, the equation on the second line of the **Algorithm 3** shall be replaced by the equation 4 assigning all the edges to the one group which will be mapped to the single link operator. If a full FPNN is being mapped, the line should be replaced by the equation 5 assigning every edge to the separated group.

$$groups \leftarrow \omega^{-1}(l) \quad (4)$$

$$groups \leftarrow \{\{e\}|e \in \omega^{-1}(l)\} \quad (5)$$

After the groups edges are constructed, the values of partial products P_e from the equation 2 as well as the approximation values A_e are computed for every edge in every group in the **Algorithm 4**. As the last step, the related link operator is computed for each group using the ψ function. The operators computation is complete and the link is removed from the chain a algorithm continues with the successive link.

TABLE I. DECLARATIONS

Declaration	Description
$FPNN = (N, L, E, \phi, \omega)$	a grid FPNN
$layers \in \{N^*\}^*$	The set of FPNN activators layers.
$chains \subset E^*$	An ordered collection of link sequences.
$sortByLength : E^n \to E$	Sorting by path length.
$firstNodeOf : E^n \to E$	Chain's first node.
$\forall e \in E : \exists P_e \in \mathbb{R}$	Partial product of the operators sequence.
$\forall e \in E : \exists A_e \in \mathbb{R}$	Approximation value for the operator computation.

1: **procedure** INITIALIZE($NN, FPNN$)
/* Init of the variables: */
2: **for all** $\forall e \in E$ **do**
3: $P_e \leftarrow 1.0$
4: $A_e \leftarrow 1.0$
5: **end for**
6: **end procedure**

Algorithm 1. Initialization algorithm

The presented algorithms represent the very basic mapping method. In our previous research [6] we developed a set of additional methods for mapping the light FPNNs which can be used to map the reduced FPNNs as well. The algorithms differ in the way of computing the ψ function. They are based on different usage of weighted algorithms with different weights

1: **function** DETERMINECHAINS($lr = \{n1..n_n\} \in N^n$)
2: $chains \leftarrow \emptyset$
3: **for all** $n \in lr \wedge s \in N$ **do**
4: **for all** $(n, s) \in E$ **do**
5: $chains \leftarrow chains \cup \phi((n, s))$
6: **end for**
7: **end for**
8: $sortByLength(chains)$
9: **return** $chains$
10: **end function**

Algorithm 2. Chain determination algorithm

1: **procedure** DETERMINEGROUS($l \in L$)
2: $groups \leftarrow [e]_{\equiv_l} \forall e \in \omega^{-1}(l)$
3: **return** $groups$
4: **end procedure**

Algorithm 3. Initialization algorithm

1: **procedure** MAPFPNN($NN, FPNN$)
2: INITIALIZE($NN, FPNN$)
3: **for all** $layer \in layers$ **do**
4: $chains \leftarrow$ DETERMINECHAINS($layer$)
/* Mapping path by path: */
5: **for all** $r \in chains$ **do**
6: $l \leftarrow firstLinkOf(r)$
/* Multiplicands comput.: */
7: **for all** $g \in$ DETERMINEGROUPS(l) **do**
8: **for all** $e \in g$ **do**
9: $P_e = \prod_{\alpha_l \in opSeq_e} \alpha_l$
10: $A_e \leftarrow \frac{W_e}{P_e}$
11: **end for**
/* Computing the link operators: */
12: $\alpha_{\sigma(e \in g)} = \psi(g) = \frac{\sum_{e \in g} A_e}{|g|}$
13: **end for**
/* Shortening the chain: */
14: $r \leftarrow r \setminus \{l\}$
15: $chains \leftarrow \{p | p \in chains \wedge p \neq \emptyset\}$
16: **end for**
17: **end for**
18: **end procedure**

Algorithm 4. Reduced FPNN mapping algorithm

determination as well as on more advanced principles. They also use different ways of results optimization. However, in order to explain the problem we used the basic method only since the other methods are more complicated and their results could depend more on the concrete network.

We implemented these algorithms into our framework [5] dealing with the FPNNs. The framework allow us to construct FPNNs and map the neural networks to FPNNs as well as simulating the computation of the FPNN and generating the VHDL design for every FPNN. Using the framework, the mapping is very fast, depending on the methods, FPNN size and used optimizations it takes seconds to minutes to be executed.

IV. IMPLEMENTATION OF FPNNS INTO FPGAS

The VHDL implementation of both types was created according to the original design and schematic [1]. Another implementation was proposed in [10]. Both, activators and links were designed as separated units communicating with signals. The communication is based on the asynchronous *request - acknowledgement* model. Every neural resource generates requests for all units directly connected to its output (successors) when its computation is done. Once a successor starts to process the request, it sends the acknowledgement back to the original resource. When the original resource receives acknowledgements from all successors, it selects a new input request to process, sends the acknowledgement and begins the computation. The activators also send a *flag* together

with the requests. The flag is a constant number and it is used by links to select the proper weight to multiply width the input data. The links then propagate the flag to all connected links. Only the full FPNNs use flags. The reduced and light FPNNs implementation do not contain the logic related to flags processing and transition.

The implementations of both types of neural resources are similar, however they differ in used computational units. The diagram of standard link implementation is illustrated in Fig. 2 and the diagram of the activator in Fig. 3. Both types are composed of a multiplexor, demultiplexor, register, computation units and units for processing requests. The meaning of common units is described bellow:

- **SELECT** selects one of the active requests for processing using the *Round&Robin* algorithm. The requests from preceding neural resources are indicated by the set bits on its input. When the request is selected, it sets the $start$ signal up.
- **MUX** is an input data vectors multiplexer. It is controlled by the SELECT unit.
- **REG** is a register storing the selected data vector.
- **ACK_DEMUX** delivers an acknowledgement (generated by the $start$ signal) to the proper predecessor. It is controlled by the SELECT unit.

These units are present in both links and activators. They serve for input requests processing and delivery of the input data to the computation part of the unit. Computation part of links and activators is composed of different units:

- **MULT_ADD** applies the weights to the data. The key to select the proper weight is the flag associated with the request. The flag is selected from all of the flags at the input $FLAG_IN$ by the value at the input s. The weights are stored in the memory inside this unit. Full FPNNs contain significantly more weights than reduced or light FPNNs.
- **ITER** iteratively computes the sum of all input data (simulates the neuron basis function). After a predefined number of iterations, it transmits the result to the TRANS unit and activates it using the fin signal. After every iteration it activates the $next$ signal which starts the processing of another request.
- **TRANS** computes the activation function (the output of the activator). The input is gained from the ITER unit. The activation function were sigmoid like function suitable for hardware implementation [13].

All computation units take the input data from the register REG, perform the computation of the result and transmit it to the neural resource output. They also activate the signal $ready$ which is an input of the output requests generators:

- **LINK_REQ_GEN** generates the requests to the connected successors when the $ready$ signal is set. It also receives the acknowledgements from the successors. Using the $free$ signal it controls the SELECT unit - it enables (when all acknowledgements are received) or disables (new request was selected - $start$ signal is up) its function.
- **ACT_REQ_GEN** is similar to the LINK_REQ_GEN, but it allows to activate the $free$ signal using the $next_req$ signal without the requests generation.

These units are responsible for the control of the neural resource. When the processing of the selected request is started, they block the SELECT unit preventing it from selecting another request before the actual one is processed. After the computation is done, they generate output requests and hold the entire neural resource inactive until all requests are successfully received by the successors.

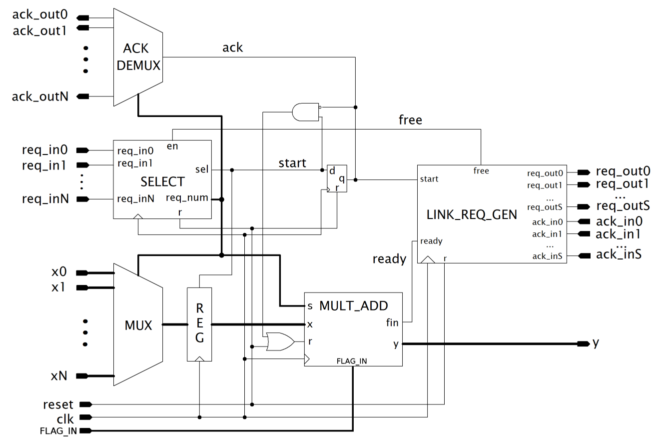

Fig. 2. Diagram of a link implementation - the interconnection of the inner units

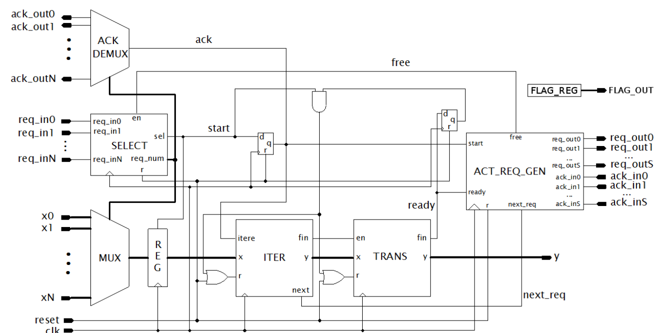

Fig. 3. Diagram of an activator implementation - the interconnection of the inner units

V. Experimental results

We experimented with the presented models and algorithms, the experiments and results will be now described and summarized. The experiments were focused on the approximation capabilities of the reduced and full FPNNs model, and on their FPGA resource consumption. The goals of the experiments were to show and compare the capabilities of both models and their space complexity. To perform the experiments we used our framework to simulate the FPNNs in order to get

the approximation accuracy. The VHDL design of every FPNN was generated using the framework as well.

The core of the experiments was a set of neural networks, and a set of structurally corresponding FPNNs of both types. Each trained neural network and the particular FPNN were both tested on a set of testing input vectors and their outputs were compared to each other to determine how the FPNN output differs from the reference neural network output. Since the FPNN serves as an approximation of the network, the match between their outputs is the essential information.

We worked with 15 neural networks of different structures trained for 3 classification tasks originating in the *Proben1* neural networks set of benchmarks [4]. The selected tasks were *Diabetes*, *Thyroid* and *Two Spiral*. The referential neural networks were constructed with respect to obtaining the set of networks with different structures not too big for the implementation in the selected FPGA, with no respect to their classification capabilities irrelevant for the FPNNs approximation quality determination.

Table II contains the information about reference neural networks. The *Name* column contains the network name (which is derived from the particular network task), the *Structure* column describes the network structure as numbers of neurons in each layer separated by the dashes. The *Neurons* and *Weights* columns summarize the numbers of neurons and weights of the network. The last column contains the number of links of the particular FPNN. The number of activators is equal to the numbers of neurons.

The experiments were run ten times and the best and the worst results of the approximation are presented in table III. Table III contains the approximation accuracy test results. The *Name* column refers to the particular FPNN (and the reference network), the *Reduced best* column contains the approximation accuracy of the reduced type FPNN output. The best case results are evident from this column. The *Reduced worst* column contains the approximation accuracy of the worst case results. It is the rate of the identically classified input vectors by both the network and the FPNN. The the rate of match of the Full FPNNs is 100%.

The created FPNNs were implemented using VHDL and were synthesized using the Xilinx ISE 14.4 tool. The target FPGA was the Xilinx Virtex-7 device *xc7v2000t-2flg1925*. All computations were implemented in fixed point form with 8 bits of the integer part and 8 bits of the fractional part [8]. The utilization of slice registers, slice LUTs, DSPs and minimum recommended clock period after synthesis were measured. The result are summarized in tables IV and V. The columns contain the utilization of the particular FPGA resources in the form of the total number and the percentual usage of the total available resources. The last column contains the minimum recommended clock period.

As the table III shows, the reduced FPNNs reached different levels of the approximation capabilities. Five of the reduced FPNNs were approximating the original network with the accuracy higher than 90 %. Three other FPNNs outreached the level of 70 % accuracy. However, some FPNNs did not cross the level of 50 % accuracy. The worst case results, which were in most cases very different from the best case results, were few times close to the 0% accuracy. These particular networks are unable to be approximated using the reduced FPNN and the full FPNN is the only choice. These findings show, how the mapping process is dependent on the concrete situation, the concrete neural network and the set of its weights. It shows that in some cases the mapping can be successful and the particular neural network can be approximated with the reduced FPNN, in other cases it is not possible. However, only the basic mapping method was used in these experiments. We developed a set of additional mapping methods which could provide better results. We presented and compared these methods and their optimizations in [6].

As the tables IV and V show, the results of the FPGA resources utilization experiments differ in case of both FPNN types. The slice registers consumption does not differ much, other results however differ significantly. As expected, the full FPNNs consume more resources than reduced FPNNs. Considering the number of consumed LUTs, the difference is in some cases only a few percent (diabetes1, twoSpiral1). In some other cases, the full FPNNs consume multiple number of resources than their reduced equivalents (diabetes4, diabetes5, thyroid2 and others). Also, full FPNNs consume multiple times more DSPs than the reduced FPNNs. This was expected since the multipliers are supposed to be more complex due to higher

TABLE II. THE LIST OF NEURAL NETWORKS AND THEIR PROPERTIES

Network name	Structure	Neurons	Weights	Links
diabetes1	8-16-8-2	34	272	78
diabetes2	8-64-2	74	640	200
diabetes3	8-64-32-2	106	2624	294
diabetes4	8-32-32-2	74	1344	198
diabetes5	8-32-32-32-2	106	2368	292
diabetes6	8-96-2	106	960	296
diabetes7	8-16-32-16-2	74	1184	196
diabetes8	8-16-32-64-2	122	2816	340
diabetes9	8-16-32-16-32-16-2	122	2208	336
thyroid1	21-21-3	45	504	86
thyroid2	21-42-3	66	1008	149
thyroid3	21-63-3	87	1512	212
thyroid4	21-84-3	108	2016	275
thyroid5	21-21-42-21-3	108	2268	271
thyroid6	21-63-21-3	108	2709	273
twoSpiral1	2-32-1	35	96	96
twoSpiral2	2-64-1	67	192	192
twoSpiral3	2-96-1	99	288	288
twoSpiral4	2-128-1	131	384	384
twoSpiral5	2-16-32-16-1	67	1072	188
twoSpiral6	2-16-32-48-1	99	2128	284

TABLE III. THE REDUCED FPNNS APPROXIMATION ACCURACY

FPNN name	Reduced best [%]	Reduced worst [%]
diabetes1	69.712	60.052
diabetes2	72.062	67.885
diabetes3	72.584	67.624
diabetes4	69.451	34.203
diabetes5	71.279	31.331
diabetes6	70.496	33.942
diabetes7	73.107	31.592
diabetes8	60.835	26.370
diabetes9	56.919	26.631
thyroid1	93.498	6.640
thyroid2	93.498	18.644
thyroid3	93.414	2.222
thyroid4	93.136	47.513
thyroid5	73.214	3.111
thyroid6	91.386	2.472
twoSpiral1	56.770	52.604
twoSpiral2	54.166	51.562
twoSpiral3	49.479	46.875
twoSpiral4	53.645	53.645
twoSpiral5	52.604	48.437
twoSpiral6	74.479	63.541

TABLE IV. THE RESULTS OF THE SYNTHESIS OF THE REDUCED FPNNS

Name	Regs (%)	LUTs (%)	DSPs (%)	MinPer [ns]
diabetes1	4726 (0%)	18235 (1%)	182 (8%)	12.725
diabetes2	11165 (0%)	45604 (3%)	464 (21%)	12.725
diabetes3	16252 (0%)	67049 (5%)	686 (31%)	12.725
diabetes4	11229 (0%)	45908 (3%)	462 (21%)	12.725
diabetes5	17270 (0%)	69225 (5%)	684 (31%)	12.719
diabetes6	16254 (0%)	67405 (5%)	688 (31%)	12.725
diabetes7	6028 (0%)	23996 (1%)	238 (11%)	12.725
diabetes8	18865 (0%)	77224 (6%)	796 (36%)	12.725
diabetes9	18699 (0%)	76614 (6%)	792 (36%)	12.725
thyroid1	5738 (0%)	20059 (1%)	182 (8%)	12.725
thyroid2	9164 (0%)	33763 (2%)	329 (15%)	13.629
thyroid3	12538 (0%)	48346 (3%)	476 (22%)	12.725
thyroid4	15898 (0%)	62788 (5%)	623 (28%)	12.725
thyroid5	15763 (0%)	62679 (5%)	619 (28%)	12.738
thyroid6	15906 (0%)	61993 (5%)	621 (28%)	12.733
twoSpiral1	5217 (0%)	21713 (1%)	228 (10%)	12.725
twoSpiral2	10209 (0%)	43543 (3%)	452 (20%)	12.733
twoSpiral3	15201 (0%)	64994 (5%)	676 (31%)	12.745
twoSpiral4	20193 (0%)	85449 (6%)	900 (41%)	12.874
twoSpiral5	10329 (0%)	42877 (3%)	448 (20%)	12.725
twoSpiral6	15413 (0%)	63528 (5%)	672 (31%)	12.725

TABLE V. THE RESULTS OF THE SYNTHESIS OF THE FULL FPNNS

Name	Regs (%)	LUTs (%)	DSPs (%)	MinPer [ns]
diabetes1	5108 (0%)	29564 (2%)	376 (17%)	11.519
diabetes2	11530 (0%)	69785 (5%)	904 (41%)	20.312
diabetes3	18078 (0%)	392493 (32%)	2160 (100%)	37.806
diabetes4	11536 (0%)	111880 (9%)	1608 (74%)	18.685
diabetes5	18630 (0%)	325759 (26%)	2159 (99%)	21.249
diabetes6	16618 (0%)	103702 (8%)	1352 (62%)	30.555
diabetes7	11460 (0%)	103727 (8%)	1448 (67%)	14.937
diabetes8	22166 (0%)	460280 (37%)	2159 (99%)	20.416
diabetes9	21418 (0%)	284021 (23%)	2159 (99%)	18.780
thyroid1	6780 (0%)	47190 (3%)	600 (27%)	11.519
thyroid2	13546 (0%)	132290 (10%)	1776 (82%)	13.402
thyroid3	13546 (0%)	132290 (10%)	1776 (82%)	13.402
thyroid4	17594 (0%)	215282 (17%)	2159 (99%)	34.575
thyroid5	18027 (0%)	286474 (23%)	2160 (100%)	16.469
thyroid6	18151 (0%)	391568 (32%)	2159 (99%)	17.423
twoSpiral1	5311 (0%)	17672 (1%)	228 (10%)	11.519
twoSpiral2	10303 (0%)	35684 (2%)	452 (20%)	11.406
twoSpiral3	15295 (0%)	53188 (4%)	676 (31%)	19.025
twoSpiral4	20287 (0%)	70903 (5%)	900 (41%)	25.345
twoSpiral5	10393 (0%)	93953 (7%)	1332 (61%)	14.937
twoSpiral6	17051 (0%)	223580 (18%)	2160 (100%)	18.481

number of weights in the full FPNNs. While links in the reduced FPNNs contain usually up to three weights, the links in full FPNNs can dispose of tens of weights. Reduced FPNNs also can generally operate on higher clock frequencies.

VI. CONCLUSIONS AND FUTURE RESEARCH

In this paper, the formal definitions of the FPNA/FPNN concept were presented. The definitions of the new derived types were introduced. The process of direct transformation of the trained neural network to FPNN and the related algorithms were described. The diagrams of the FPGA implementation were presented. The experiments determining the approximation capabilities of different FPNNs of the reduced and the full type were run and their results were presented in this paper. The results show that in some cases, the reduced FPNN type is capable of good approximation performance. However, this depends on the concrete neural network and its weight values and their combinations. Therefore, the reduced FPNN are not suitable for all neural network implementations.

One of the main ideas of this paper was to show the possible trade-off between neural network approximation accuracy and the FPGA resources consumption. The experiments showed that reduced FPNNs consume significantly less resources than full FPNNs and that they are faster as well. On the other hand, the reduced FPNNs offer limited approximation accuracy compared to the accurate full FPNNs.

During the future research, we are going to perform experiments using our more advanced mapping techniques to increase the usability of reduced FPNNs as well as develop new methods and optimizations. Also we are going to include more types of neural networks into our research.

ACKNOWLEDGMENT

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602, ARTEMIS JU under grant agreement no 641439 (ALMARVI) and BUT project FIT-S-14-2297.

REFERENCES

[1] Girau, B.: FPNA: Concepts and Properties. In *FPGA Implementations of Neural Networks*, A. R. Omondi; J. C. Rajapakse, Springer US, 2006, ISBN 978-0-387-28487-3, pp. 71–123, http://dx.doi.org/10.1007/0-387-28487-7-3

[2] Girau, B.: Digital hardware implementation of 2D compatible neural networks. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, 2000, ISSN 1098-7576, pp. 506–511 vol.3

[3] Girau, B.: On-chip learning of FPGA-inspired neural nets. In *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on*, 2001, ISSN 1098-7576, pp. 222–227 vol.1

[4] Prechelt, L. P.; Informatik, F. F.: — A Set of Neural Network Benchmark Problems and Benchmarking Rules. Technical report, Universitat Karlsruhe; 76128 Karlsruhe, Germany, 1994.

[5] Krcma, M.: *The neural networks acceleration in FPGA*. Master's thesis, Faculty of Information Technology, Brno University of Technology; Brno, 2014. https://wis.fit.vutbr.cz/FIT/st/rp.php/rp/2013/DP/15754.pdf

[6] KRCMA Martin, KASTIL Jan a KOTASEK Zdenek: *Mapping trained neural networks to FPNNs.* In: IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems. Belgrade: IEEE Computer Society, 2015, pp. 157–160. ISBN 978-1-4799-6779-7.

[7] Krcma, M.; Kotasek, Z.; Kastil, J.: Fault tolerant Field Programmable Neural Networks. In *Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC), 2015*, Oct 2015, pp. 1–4, 10.1109/NORCHIP.2015.7364381.

[8] Holt, J.; Baker, T.: Back propagation simulations using limited precision calculations. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume II, July 1991, pp. 121 –126 vol.2.

[9] Munakata, T.: Neural Networks: Fundamentals and the Backpropagation Model. In *Fundamentals of the New Artificial Intelligence*, editace T. Munakata, Texts in Computer Science, Springer London, 2007, ISBN 978-1-84628-839-5, pp. 7–36, http://dx.doi.org/10.1007/978-1-84628-839-5--2

[10] Bohrn, M.; Fujcik, L.; Vrba, R.: Field Programmable Neural Array for feed-forward neural networks. In *2013 36th International Conference on Telecommunications and Signal Processing (TSP)*, 2013, pp. 727–731

[11] Harkin, J.; McDaid, L.; Hall, S.: Programmable architectures for large-scale implementations of Spiking Neural Networks. In *IET Irish Signals and Systems Conference (ISSC 2008)*, June 2008, ISSN 0537-9989, pp. 374–379

[12] Harkin, J.; Morgan, F.; Hall, S.: Reconfigurable platforms and the challenges for large-scale implementations of spiking neural networks. In *2008 International Conference on Field Programmable Logic and Applications*, Sept 2008, ISSN 1946-147X, pp. 483–486

[13] Kwan, H.: Simple sigmoid-like activation function suitable for digital hardware implementation. *Electronics Letters*, 1992: pp. 1379–1380. http://link.aip.org/link/?ELL/28/1379/1

Paper C

Comparison of FPNNs Approximation Capabilities

M. Krcma, J. Kastil, Z. Kotásek and J. Lojda, „Comparison of FPNNs Approximation Capabilities,“ *Proceedings of the Work in progress Session held in connection with DSD 2016*, 2016, pp. 1-2, ISBN:978-3-902457-46-2.

Comparison of FPNNs Approximation Capabilities

Martin Krcma, Zdenek Kotasek, Jakub Lojda, Jan Kastil
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
Email: ikrcma@fit.vutbr.cz, kotasek@fit.vutbr.cz, ilojda@fit.vutbr.cz, ikastil@fit.vutbr.cz

I. Introduction

The artificial neural networks [5] are one of the important models of softcomputing and artificial intelligence. They are structures composed of *neurons* interconnected by weighted *synapses*. Basically, the goal of the networks is to learn the relation between two sets of data vectors, to generalize the relation, to determine its features and to use it for the determining the relation of the unknown vectors belonging to the same problem. This capability can be used for classification tasks, for time series and functional prediction, to control tasks, to image recognition, clustering and other tasks.

The implementation of neural networks is challenged with two great neural networks complexities - space complexity and time complexity. The usual solution of both is to use a powerful hardware, such as graphical processor units or processor clusters, which suffer from a high power consumption. For some networks, FPGAs can be one of the possible solutions if a lower power consumption is desired. In this case, the time complexity is solvable by parallelism which is easy to achieve in both FPGAs and neural networks since both are parallel by their nature. The space complexity is bigger problem since an FPGA has limited resources. Thus, there is a need for such designs that exploit the neural networks parallel character for fast computations and save the FPGA resources as well. A Field Programmable Neural Networks (FPNN) concept can be seen as one of the possible solutions. The goal of this paper is to describe the types of FPNNs and compare their capabilities.

II. Field Programmable Neural Networks

The concept of FPNNs [1] is meant to simplify the implementation of artificial neural networks in FPGAs by adjusting their properties to be more suitable for implementation into them. The simplification originates from its main feature - a highly customizable structure which makes it possible to establish resource sharing between the original synaptic connections of the neural network. The FPNNs are composed of dedicated interconnected units called neural resources which approximate the original neurons and synaptic interconnections. The units of the first type are called *activators* and represent the original neural network neurons. The other units are called *links* and serve as an approximation of the original synaptic interconnection. Every link disposes of a set of affine operators serving as an approximation of the original synaptic weights.

An example of a grid FPNN can be seen in Fig. 1. The circles in the figure represent activators, wide arrows represent links and the thin arrows represent data interconnections. The orientation of the connection arrows shows the way of the passing data. The straight wide dashed/dotted arrows represent the original neural networks synapses. The thin dashed/dotted arrows represent the sequences of links approximating the particular synapses. The synapses and the particular sequences are drawn with the same line and arrow styles.

The FPNNs are not the same structures as neural networks, although they can be constructed in that way. The FPNNs represent a different model which can structurally differ from the implemented neural network. They can also have different capabilities which means that they are not only an implementation of the neural networks, they are an approximation of neural networks as well - with different structure and properties, they can provide similar results as the networks. The accuracy is the main problem here.

The approximation capabilities depend on the number of affine operators belonging to links. This number depends on the FPNN structure directly. However, the model can be altered to dispose of different number of affine operators. Two different models with different approximation capabilities exist. The original model disposes of as many affine operators as the number of directly connected preceding units. These operators are shared between groups of synapses approximated by the particular preceding units. This type of an FPNN is called *Standard FPNN*. We derived a stronger model that has the number of affine operators that allows it to reach the precise approximation accuracy. This type of an FPNN is called *Full FPNN*. In case of a full FPNN, every link disposes of dedicated affine operator for every synapse it approximates. There is no sharing of affine operators between synapses, therefore the accurate approximation is ensured. Although, this type of FPNN demands more FPGA resources.

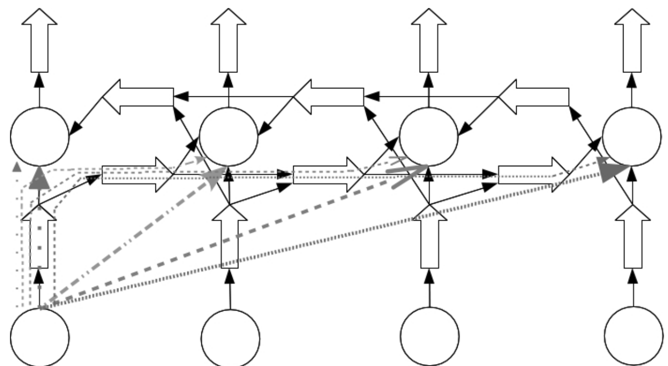

Fig. 1. Synapses approximation in a grid FPNN

III. Experimental results

We experimented with the presented models and algorithms, the experiments and results will be now described and summarized. The experiments were focused on the approximation capabilities of the standard and full FPNNs model. The goal of the experiments was to show and compare the capabilities of both models and their space complexity.

The base of the experiments was a set of neural networks, which were transformed (using our algorithm described in [3]) to FPNNs of both types. Both, the trained neural network and the particular FPNN were tested on a set of testing input vectors and their outputs were compared to each other to determine how the FPNN output differs from the reference neural network output. Since the FPNN serves as an approximation of the network, the match between their outputs is the essential information. We worked with 15 neural networks of different structures trained for three classification tasks (*Diabetes*, *Thyroid* and *Two Spiral*) originating in the *Proben1* neural networks benchmark tasks set [2].

Table I contains the information about the reference neural networks. The columns contain the network name and its structure (numbers of neurons in each layer), the numbers of neurons and synapses of the network and the number of links of the FPNN. The number of activators is equal to the numbers of neurons.

The experiments were run ten times and the best and the worst results of the approximation are presented in table II. The *Standard best* and *Standard worst* columns contain the percentual rate of match of the reference network and the standard type FPNN output. It is the rate of the identically classified input vectors by both the network and the FPNN. The *Full* column contains the rate of match of the Full FPNN type.

Network name	Structure	Neurons	Synapses	Links
diabetes1	8-64-2	74	640	200
diabetes2	8-32-32-32-2	106	2368	292
diabetes3	8-96-2	106	960	296
diabetes4	8-16-32-64-2	122	2816	340
diabetes5	8-16-32-16-32-16-2	122	2208	336
thyroid1	21-42-3	66	1008	149
thyroid2	21-84-3	108	2016	275
thyroid3	21-21-42-21-3	108	2268	271
thyroid4	21-63-21-3	108	2709	273
thyroid5	21-10-63-10-3	107	1500	268
twoSpiral1	2-64-1	67	192	192
twoSpiral2	2-128-1	131	384	384
twoSpiral3	2-16-32-16-1	67	1072	188
twoSpiral4	2-16-32-64-32-16-1	163	5168	472
twoSpiral5	2-16-32-48-1	99	2128	284

TABLE I. The list of the neural networks and their properties

As the table II shows, the standard FPNNs reached different levels of the approximation capabilities. Some of the standard FPNNs reached an accuracy higher than 90 %. However, some FPNNs did not cross the level of 50 % accuracy. The worst case results, which were in most cases very different from the best case results, were few times closer to the 0% accuracy. These findings show that in some cases the mapping to a standard FPNN can be successful, in other cases it is not possible. However, the results can be potentially improved by methods and optimizations described in [3].

FPNN name	Standard best [%]	Standard worst [%]	Full [%]
diabetes1	72.062	67.885	100
diabetes2	71.279	31.331	100
diabetes3	70.496	33.942	100
diabetes4	60.835	26.370	100
diabetes5	56.919	26.631	100
thyroid1	93.498	18.644	100
thyroid2	93.136	47.513	100
thyroid3	73.214	3.111	100
thyroid4	91.386	2.472	100
thyroid5	75.715	1.972	100
twoSpiral1	54.166	51.562	100
twoSpiral2	53.645	53.645	100
twoSpiral3	52.604	48.437	100
twoSpiral4	49.479	33.333	100
twoSpiral5	74.479	63.541	100

TABLE II. The list of the FPNNs approximation results

IV. Conclusions and future research

In this paper, the FPNN concept was described. The original (standard) and the derived new (full) type were presented. The experiments determining the approximation capabilities of different FPNNs of the standard and the full type were run and their results were presented in this paper. The results show that in some cases, the standard FPNN type is capable of good approximation performance. However, this depends on the concrete neural network and its weights values.

During the future research, we are going to include more types of neural networks into our research and we are going to perform more hardware oriented experiments. We are also going to devote to the comparison of our results and methodologies with other approaches.

Acknowledgment

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602. This work was also supported by Brno University of Technology under number FIT-S-14-2297 and by ARTEMIS JU under grant agreement no 641439 (ALMARVI).

References

[1] Girau, B.: FPNA: Concepts and Properties. In *FPGA Implementations of Neural Networks*, editted by A. R. Omondi; J. C. Rajapakse, Springer US, 2006, ISBN 978-0-387-28487-3, p. 71–123, 10.1007/0-387-28487-7-3.
http://dx.doi.org/10.1007/0-387-28487-7-3

[2] Prechelt, L. P.; Informatik, F. F.: — A Set of Neural Network Benchmark Problems and Benchmarking Rules. Technical report, Universitat Karlsruhe; 76128 Karlsruhe, Germany, 1994.

[3] KRCMA Martin, KASTIL Jan a KOTASEK Zdenek: *Mapping trained neural networks to FPNNs.* In: IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems. Belgrade: IEEE Computer Society, 2015, pp. 157–160. ISBN 978-1-4799-6779-7.

[4] Krcma, M.; Kotasek, Z.; Kastil, J.: Fault tolerant Field Programmable Neural Networks. In *Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC), 2015*, Oct 2015, s. 1–4, 10.1109/NORCHIP.2015.7364381.

[5] Munakata, T.: Neural Networks: Fundamentals and the Backpropagation Model. In *Fundamentals of the New Artificial Intelligence*, editace T. Munakata, Texts in Computer Science, Springer London, 2007, ISBN 978-1-84628-839-5, s. 7–36, 10.1007/978-1-84628-839-5–2.
http://dx.doi.org/10.1007/978-1-84628-839-5--2

Paper D

Detecting hard synapses faults in artificial neural networks

M. Krcma, Z. Kotasek and J. Lojda, „Detecting hard synapses faults in artificial neural networks,“ *2019 IEEE Latin American Test Symposium (LATS)*, 2019, pp. 1-6, doi: 10.1109/LATW.2019.8704637.

Detecting hard synapses faults in artificial neural networks

Martin Krcma, Zdenek Kotasek, Jakub Lojda
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence
Bozetechova 2, 612 66 Brno, Czech Republic
Email: ikrcma@fit.vutbr.cz, kotasek@fit.vutbr.cz, ilojda@fit.vutbr.cz

***Abstract*—This paper presents the concepts of detecting hard faults in artificial neural network synapses using the modification of the neural network settings. The core of this work is based on weights values modification and inserting the chosen testing data when comparing the neural network output to the known valid results. The paper also discusses the problem of neural networks output saturation and provides experiments regarding an influence of the neural network settings to the problem.**

I. Introduction

The artificial neural networks [7] are one of the important models of soft-computing and artificial intelligence. Their structure is inspired by the structure of the human brain and they dispose of a high capability of learning and memorizing to solve various types of tasks. Basically, the goal of the artificial neural network is to learn the relation between two sets of data vectors, to generalize the relation, to determine its features and to use it for the determining the relation of the unknown vectors belonging to the same problem. This capability can be used for classification tasks, for time-series and functional prediction, for control tasks, image recognition, clustering and other tasks.

The networks have been implemented in various kinds of devices starting from analog computers to the most modern processors, VLSI units, graphical processing units and FPGAs. In the hardware implementation there is a chance that a fault occurs in the device influencing its computation. The fault can be transient, temporary which can be solved by numerous ways. If the fault is hard and permanent however, it may not be possible to fix it. In this case, the detecting of the fault is even more important than in the case of temporary faults because the computation of the device and the data it produces are permanently affected by the fault. This paper deals with one of the possible ways how to detect hard faults in neural network synapses.

II. Artificial Neural Networks

Neural networks are composed of a set of *neurons*. A neuron is a simple unit which computes an *activation function* (2) over a result of a *basis function* which is often a *weighted sum* (1) of the neuron inputs. A neuron is illustrated in Fig. 1. The neurons are interconnected with the weighted connections called *synapses*. The learning of the neural network is basically a process of setting the weights.

978-1-7281-1756-0/19/$31.00 ©2019 IEEE

The value θ in equation (2) represents the neuron *threshold*. The threshold allows us to affect the shape of the neuron activation function (its position on the x axis) which increases the power of the network and the efficiency of its learning.

The neurons are often organized into the layered structure composed of an input layer, output layer and a number of hidden layers. This type of structure is illustrated in Fig. 2. Sometimes, the neural network is composed of only one or two layers or of layers with different neuron types (i.e. neurons with different basis and activation functions) or interconnection structures.

$$net = \sum_{i=1}^{n} x_i w_i \tag{1}$$

$$y = f(net + \theta) \tag{2}$$

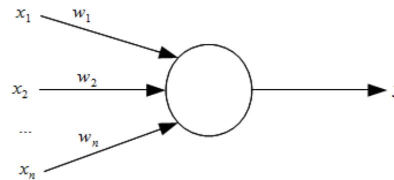

Fig. 1. The neuron.

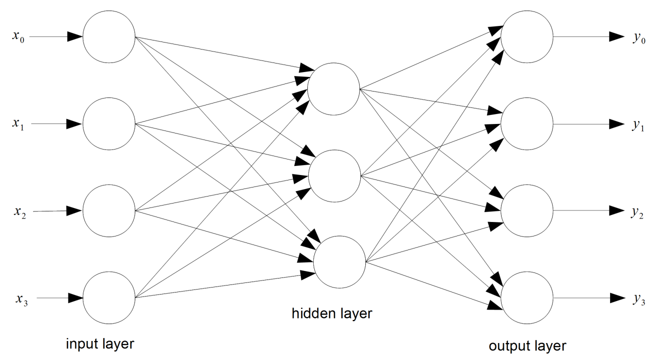

Fig. 2. The neural network layered structure.

A. Activation functions approximations

A number of different activation functions which are used in neural networks exists. One of the most used activation

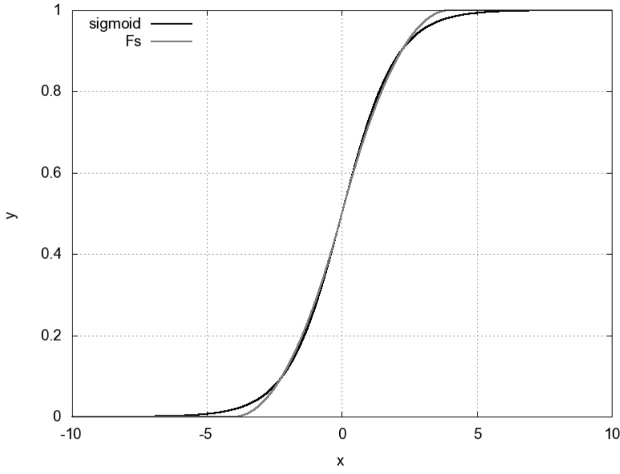

Fig. 3. The sigmoid function and its approximation.

functions in classical neural networks is a *sigmoid* function (3). It is a growing differentiable function, the features of which are important for gradient-descent based learning algorithms like the well known *backpropagation* algorithm [7]. However, this function uses operations of division and power which are not suitable for implementation in hardware. Therefore, it is appropriate to replace it with a more effective approximation with similar features. One of possible approximations is Fs function (7) which is based on equations (4),(5) and (6). Both the sigmoid function graphs and the Fs function are compared in Fig. 3.[8]

$$sigmoid(x) = \frac{1}{1 + \epsilon^{-\theta x}} \tag{3}$$

$$\theta = \tfrac{1}{L^2} \quad \beta = \tfrac{2}{L} \tag{4}$$

$$H_s(x) = \begin{cases} x\,(\beta + \theta x) & \text{for } x \in \langle -L, 0 \rangle \\ x\,(\beta - \theta x) & \text{for } x \in (0, L \rangle \end{cases} \tag{5}$$

$$G_s(x) = \begin{cases} -1 & \text{for } x \in (-\infty, -L \rangle \\ H_s(x) & \text{for } x \in (-L, L) \\ 1 & \text{for } x \in \langle L, \infty) \end{cases} \tag{6}$$

$$F_s(x) = \frac{1}{2} G_s(x) + \frac{1}{2} \tag{7}$$

III. Detecting the hard faults

Artificial neural networks are inherently massively parallel structures with a lot of redundancy. Even though this property makes them able to tolerate some faults, this fault tolerance reaches only a certain level and it is complicated to predict its quality. In order to enhance the fault tolerant properties of neural networks, several techniques are used. Some techniques are based on modifications of the neural networks training process to force the networks to learn to be fault tolerant [10], [11], [12]. Other techniques use retraining as a way of recovery from a fault [9], [18]. In some techniques, different modifications and restrictions of weights and neurons activation functions take place [13], [14], [15], [16], [17]. Also, techniques which utilize redundancy are commonly used. Either based on neurons replications [19], [21], [24], [25] or on the well known Triple Modular Redundancy (TMR) technique which is used for both faults detection and masking it in order to produce a correct output.

When implemented in hardware, the neural networks may face two types of faults. The soft (temporary) faults occur only temporarily and affect the computation only for some time. These faults are often caused by radiation generating a Single Event Upset (SEU) - flipping a bit in a memory. This type of fault may be often solved by rewriting the memory with correct data. Other type of faults - hard faults are persistent faults often caused by a physical condition or failure in the device. Most approaches of faults detection, masking and recovery are based on the TMR techniques [22], [23]. This technique, though proven reliable, may fail in some combinations of faults as shown in [20].

We propose a method of permanent faults detection that does not utilize redundancy or learning but it uses properties of neural networks computation instead. It utilizes modifications of basic neural network parameters - weights values and activation function shapes in order to detect a fault using the network output. As we present general principles and algorithm which may be used on any neural network platform and implementation which are flexible enough to allow the needed neural network parameters settings modification, we primarily intent this method to be used on reconfigurable hardware implementations, where a hard fault is more likely to occur and is harder to mask and recover from them in a purely software implementation. In our research, we intent to use these methods, while utilizing a dynamic reconfiguration, with our neural network *Field Programmable Gate Array* (FPGA) implementation. This platform is based on the concept of *Field Programmable Neural Network* (FPNN)[1] and we presented this platform in [4], [5].

A. Definitions

In order to describe the principles clearly, we declare a set of terms presented in the following list and equations. The terms describe the neural network structure and derive additional terms. Going through the list we define sets and functions containing neurons, synapses and their weights (N,S and W). Using those, we declare a set of neurons layers L, the input layer I and the output layer O. For all neurons we define two sets - ϕ and τ. The ϕ set contains all the sequences of synapses which connect the selected neuron to the input layer (to all its neurons). To the opposite, the τ set contains all sequences of synapses that connect the neuron to the output neurons.

Based on these sets we define a set π for all the neurons which contains all sequences of synapses connecting the input layer to the output layer through the selected neuron. It is important to have this set as we need to find a way to propagate the test data through the network in the sequence that includes

the neuron or synapse we want to test against the presence of a hard fault.

1) N is the set of all neurons.
2) $S \subseteq (N \times N)$ is the set of all synapses.
3) $W = S \to \mathbb{R}$ is the set of weights of all synapses.
4) $W = \{w_s \in \mathbb{R} | s \in S\}$ is the set of weights of all synapses.
5) $B = \{b_n : \mathbb{R}^m \to \mathbb{R} | n \in N \setminus I, m \in \mathbb{N}\}$ is the set of basis functions of all neurons except the input neurons. m is the number of the neuron n inputs.
6) L is set of network layers defined by equation (8).
7) I is the input layer defined by equation (9).
8) O is the output layer defined by equation (10).
9) $F = \{f_n : \mathbb{R} \to \mathbb{R} | n \in N \setminus I\}$ is the set of activation functions of all neurons except the input neurons.
10) ϕ_n is a set of sequences of synapses connecting the neuron n to the neurons in the input layer (11).
11) τ_n is a set of sequences of synapses connecting the neuron n to the neurons in the output layer (12).
12) $\forall n \in N : \pi_n = \phi_n \times \tau_n$ is a set of all sequences of synapses connecting the input layer to the output layer through the neuron n.
13) $\psi(s)$ is a sequence of all source neurons of the synapses in the synaptic sequence s (13).
14) $\chi(s)$ is a sequence of all target neurons of the synapses in the synaptic sequence s (14).
15) $\Omega \in \mathbb{R}$ is a chosen global value of weights to be used in further algorithms.
16) $d_a, d_o \in \mathbb{R}$ are chosen input data values for an active neuron (d_a) and for other neurons in the input layer (d_o).

$$L \subset N^X : \forall (n1, n2) \in S : n1 \in L_1 \wedge n2 \in L_2; \\ L_1, L_2 \in L \tag{8}$$

$$I \in L : \forall n \in I \wedge \exists (n, n_x) \in S : \nexists (n_y, n) \in S; \\ n_x, n_y \in N \tag{9}$$

$$O \in L : \forall n \in O \wedge \exists (n_x, n) \in S : \nexists (n, n_y) \in S; \\ n_x, n_y \in N \tag{10}$$

$$\forall n \in N \exists \phi_n \subset S^X : (n_x, n_{x+1}) \in \phi_n, x \in \{i..m\}; \\ i, m \in \mathbb{N}; n_i \in I; n_{m+1} = n \tag{11}$$

$$\forall n \in N \exists \tau_n \subset S^X : (n_x, n_{x+1}) \in \tau_n, x \in \{m..o\}; \\ m, o \in \mathbb{N}; n_m = n; n_{o+1} \in O \tag{12}$$

$$\forall s = s_1 s_2 .. s_m = (n1, n2)(n2, n3)...(n_{m-1}, n_m) \in S^m : \\ \psi(s) = n1, n2, ..., n_{m-1} \tag{13}$$

$$\forall s = s_1 s_2 .. s_m = (n1, n2)(n2, n3)...(n_{m-1}, n_m) \in S^m : \\ \chi(s) = n2, n3, ..., n_m \tag{14}$$

B. Detecting a fault in a synapse without affecting the activation functions

Using the previous definitions we declare an algorithm which utilizes the modifications of neural networks properties in order to detect the hard fault of a synapse. The principle of the algorithm is to check sequentially all the synapses by propagating the test data to the network and checking the network output while setting all the other synapses weights to 1 in order to omit them from the computation. Omitting the weight ensures that the passing test data are affected only by the weight of the tested synapse which makes the result easy to determine. The algorithm is as follows:

A) Declare a set $FS = \emptyset$ to store the faulty synapses.
B) For all synapses $s \in S$ execute:

1) To test a synapse $s = (n_1, n_2) \in S$ compute π_{n2}.
2) Select a sequence α out of π_{n2}. $\alpha = \beta\gamma, \beta \in \phi_{n2}, \gamma \in \tau_{n2}$.
3) Set all other weights to Ω: $\forall w \in W \setminus \{W(a) | a \in \alpha\} : w = \Omega$.
4) Set weights in α synapses to 1, leave the original value of the tested synapse - $W(s)$: $\forall w \in W(S \setminus \alpha \setminus s) : w = 1$.
5) Present the input data i to the input layer of the network in the form of a vector composed of d_a on the place of the neuron from the α sequence and d_o in places of others input neurons. Let the neural network compute the output $o(\alpha, s, i)$.
6) Compute an ω value. $\omega(\alpha, s, i) \in \mathbb{R}$ represents the expected output value for the selected synaptic sequence α, tested synapse (n_1, n_2) and the input data i. It is computed using equation (16) as a sequence of application of all activation functions of the neurons in the β sequence over the input data followed by multiplication with the tested synapse weight. Then, the sequence of all activation functions of the neurons in the γ sequence is applied.
7) Compute the difference ϵ between the expected and the actual output value:
$\epsilon(\alpha, s, i) = o(\alpha, s, i) - \omega(\alpha, s, i)$.
8) If the difference $\epsilon = 0$ (the output is not affected by a fault), return to the step A). Otherwise execute:
   a) Repeat the steps 1 to 8 for several other α sequences containing the synapse s.
   b) If all the ϵ values are not zero, the synapse s is affected by a fault. Add then the synapse s to the set FS.

$$\omega(\alpha, s, i) = f^{\gamma}\left(f^{\beta}(i_n) \times w_s\right); \\ n \in N, i_n \in \mathbb{R}, w_s \in S; \\ f^{\beta} = f_{n1} \circ ... \circ f_{n0}; \psi(\beta) = n0, ..., n1; \\ f^{\gamma} = f_{n1} \circ ... \circ f_{n0}; \chi(\gamma) = n0, ..., n1 \tag{15}$$

The general problem to deal with in this algorithm is a possibility of neurons outputs saturation preventing the fault detection. This problem has its origin in the input (the d_a, d_o values), used weights values (the Ω value) during the algorithm

and the activation functions. Regular sigmoid function has the range of $\langle 0, 1\rangle$ and the value of 0.5 as the function value of zero ($sigmoid(0) = 0.5$). When the weights of synapses outside an α sequence are set to zero, it causes all the neurons in the rest of the network to emit value of 0.5. This can affect the neurons in the α sequence as well as these values enter their basis functions. Together with data passing through the α sequence this can cause the neuron outputs to be saturated, i.e. to have the value of 1.0 or 0.0. When these are valid values for properly functioning network and there are no changes in them, this can cause a fault to be masked from detection. This effect can be straightened even more by values presented to the input neurons both in and outside the α sequence (the d_a, d_o values). It is necessary to choose all these values wisely to obtain as correct detection as possible.

Another problem related to the saturation problem is the problem of false positive detection. In the case that the fault causes the weight to have a high value, it may saturate the successive neuron itself causing saturation of neurons in higher level as well as the saturation of the network outputs. The saturated output then may be detected as fault even in case when other synapses than the faulty one is under the test. This problem may be solved by repetitive detection using different settings as well as using heuristics to obtain specific strategies of the test. This heuristics and methods will be part of the future research.

C. Detecting a fault in a synapse with affecting the activation functions

The hard faults detection becomes easier and less demanding if there is an option to change shapes of neurons activation functions. By changing the functions to the linear functions $f(x) = x$ we can prevent the saturation problem mentioned in the previous paragraph. However, the saturation problem is still present but it is far less likely as the only risk of saturation is reaching the upper or the lower boundary given by the used data-type and the bit width. Also, by changing the activation functions, we obtain a higher precision of the output computation and lesser influence of neurons faults to the output. The algorithm, derived from the previous algorithm, which utilizes the change of activation functions is as follows:

A) Declare a set $FS = \emptyset$ to store the faulty synapses.

B) For all synapses $s \in S$ perform:

1) Perform 1) - 4) steps of the section B algorithm.
5) Set the activation functions of the neurons in the α sequence to linear function:
   $\forall n \in \psi(\beta) \cup \chi(\gamma) : f_n(x) = x; f_n(x) \in F$. When the activation function is approximated using the function (7), the modification can be done using constants modifications according to the (17) equation.
6) Perform 5) step of the section B algorithm.
7) Compute an ω value. $\omega(\alpha, s, i) \in \mathbb{R}$ represents the expected output value for the selected synaptic sequence α, tested synapse (n_1, n_2) and the input data i. It is computed using equation (16) as a sequence of applications of all activation functions of the neurons in the β sequence. In this case, the activation functions are linear, therefore the applications are in principle function of identity. The output data of the activation functions sequence is data followed by multiplication with the tested synapse weight. Then the sequence of all activation functions of the neurons in the γ sequence is applied, again as a sequence of identities.
8) Perform 7) - 8) steps of the section B algorithm.

$$\begin{aligned} &\omega(\alpha, s, i) = i_n \times w_s; \\ &n \in N, i_n \in \mathbb{R}, w_s \in S; \end{aligned} \tag{16}$$

D. Activation functions modifications

If the implementation uses the Fs function as the activation function approximation, it can be simply forced to behave like a linear function in order to pass the neuron input data directly to the output without affecting them by the activation function. In the case of the Fs function, it can be done using constants modifications and input data propagation to the multiplexers realizing the Gs function. The modifications of the functions and the constants are as follows:

$$\begin{aligned}
&\theta = 0; \quad \beta = 1 \\
&H_s(x) = \begin{cases} x\,(\beta + \theta x) = x & \text{for } x \in \langle -L, 0\rangle \\ x\,(\beta - \theta x) = x & \text{for } x \in (0, L\rangle \end{cases} \\
&G_s(x) = \begin{cases} x & \text{for } x \in (-\infty, -L\rangle \\ H_s(x) = x & \text{for } x \in (-L, L) \\ x & \text{for } x \in \langle L, \infty) \end{cases} \\
&F_s(x) = 1 \times G_s(x) + 0 = G_s(x)
\end{aligned} \tag{17}$$

IV. Experiments

We have experimented with the first algorithm which does not utilize the activation functions modifications in order to determine the influence of the d_a, d_o and Ω values to the saturation problem and therefore to the quality of faults detection.

The used neural network was composed of 8 neurons in the input layer, 2 neurons in the output layer and of 64 and 16 neurons in two hidden layers. The sigmoid function was used as the activation function of all the neurons and the function of the weighted sum was used as basis functions. The experiments were implemented using a *FANN library* [3] using 32-bit floating point arithmetic. The neural network was trained to solve the *Diabetes* classification task from the *Proben* [2] set of neural networks benchmark tasks. Every experiment used two identical neural networks, one as a golden model, the second to inject fault and perform the algorithm. Only one fault per test was injected randomly into a synapse and the algorithm was executed to detect the fault.

With each set of d_a, d_o and Ω values, 100 tests were run and the number of successful detections was measured as a result. The d_o values were chosen in the $\langle -10, 0\rangle$ as we expected that low values around zero may help to prevent the saturation problem as well as the negative values. We expect

these values cause the neurons to emit low values as well which may help to prevent the basis function to generate high values which would saturate the neurons outputs in the higher layers making the neural network output to be saturated as well. The Ω values were chosen to be the same for the same reasons as it may help to lower the high values emitted by neurons and thus lower the risk of saturation in higher layers. On the other hand, the d_i values are the most important as they enter the computation in the α sequence. In order to explore their influence on the detection quality, we chose them to be in the $\langle -10, 10 \rangle$ interval.

Tables I - III illustrate the results of the experiments. In each table, the Ω value is the same for all listed experiments and it is declared on the top row of the table. The values of d_o are declared in the third rows of the tables (the first numerical rows) and the d_i values are listed in the first columns of the tables. The cells contain the experiments results illustrating how many detections out of 100 were successful with the Ω, d_i and d_o set according to the position in the table.

TABLE I
THE EXPERIMENTS RESULTS WHEN $\Omega = 0.0$

$\Omega = 0.0$					
	d_o				
d_i	**-10.0**	**-1.0**	**-0.1**	**-0.01**	**0.0**
-10.0	21	21	56	21	24
-1.0	20	99	32	58	78
-0.1	66	99	72	1	97
-0.01	81	100	0	0	71
0.01	70	96	79	58	0
0.1	79	100	75	63	85
1.0	86	100	83	100	82
10.0	100	100	1	63	91

TABLE II
THE EXPERIMENTS RESULTS WHEN $\Omega = -0.01$

$\Omega = -0.01$					
	d_o				
d_i	**-10.0**	**-1.0**	**-0.1**	**-0.01**	**0.0**
-10.0	97	97	98	92	75
-1.0	98	98	98	100	81
-0.1	99	99	100	99	70
-0.01	100	96	97	96	64
0.01	96	98	100	99	61
0.1	100	100	100	98	68
1.0	99	100	100	97	72
10.0	100	100	100	96	80

As you can see in the tables, the most of the experiments resulted in the high ratio of detected faults. If we are going to identify the general trends of the Ω, d_i and d_o values influence on the results, we can see that higher values of d_i often led to better results. This can be also stated in general about influence of the higher negative values of d_o. As we expected, the negative values of d_o helped the detection by lowing the risk of saturation and allowing the fault detection by doing so. On the other hand, the 0.0 value of d_o proved to provide generally worse results. As it was said before, the value of 0.0 as an input to the neuron causes to emit the value of 0.5 as a result in case it uses the sigmoid function as an activation function, which is the case of these experiments. Also, the high negative values of d_i provided worse results as they probably increased the saturation problem in case the weights values were high in the α sequence or the injected fault had a high value.

TABLE III
THE EXPERIMENTS RESULTS WHEN $\Omega = -10.0$

$\Omega = -10.0$					
	d_o				
d_i	**-10.0**	**-1.0**	**-0.1**	**-0.01**	**0.0**
-10.0	75	55	53	54	60
-1.0	99	84	84	88	64
-0.1	84	91	92	100	58
-0.01	88	85	93	87	43
0.01	74	85	92	95	64
0.1	81	89	94	99	60
1.0	81	86	89	89	55
10.0	100	89	100	98	78

TABLE IV
THE EXPERIMENTS RESULTS WHEN $\Omega = -0.1$

$\Omega = -0.1$					
	d_o				
d_i	**-10.0**	**-1.0**	**-0.1**	**-0.01**	**0.0**
-10.0	97	97	98	93	82
-1.0	99	98	99	98	85
-0.1	99	99	100	100	73
-0.01	99	98	98	99	61
0.01	96	100	100	100	64
0.1	100	100	100	99	77
1.0	99	100	99	100	83
10.0	100	100	100	97	77

TABLE V
THE EXPERIMENTS RESULTS WHEN $\Omega = -1.0$

$\Omega = -1.0$					
	d_o				
d_i	**-10.0**	**-1.0**	**-0.1**	**-0.01**	**0.0**
-10.0	21	24	56	21	24
-1.0	20	58	32	58	66
-0.1	66	97	72	1	97
-0.01	81	71	0	0	71
0.01	70	91	79	58	0
0.1	79	85	75	63	85
1.0	86	100	83	100	100
10.0	100	91	1	63	91

As the tables II and IV illustrate, the low negative values of Ω has positive influence on the results. These values reduced the neurons output to higher layers helping to prevent the saturation problem. The assumption of false positive detection during the experiments was also confirmed, however this aspect of the problem is beyond the range of this paper.

V. Conclusions and Future Research

In this paper, we described basis of neural networks as well of formal basis of two algorithms which are the core of this research. The algorithms offer the way to detect hard fault in neural network synapses. Both algorithms are based on the

principle of setting the synapses weights to some chosen value, then creating an interconnected sequence from input layer to an output layer. One of the synapses is then set with its original weight and chosen testing data are passed trough the network and the output is compared to the pre-calculated valid result. The difference between the outputs indicates a fault.

One of the algorithms uses the change of activation functions to linear ones preventing the problem with an output saturation which may occur with the other algorithm. The experimental part of this work focuses on this problem and shows the effect of the chosen values of the input data and the weights to the quality of faults detection. The results show that combination of low negative Ω values with high values of d_o and negative values of d_o led in general to the best results as they were the most successful preventing the saturation of the network output.

In the future research, more extensive experiments with both algorithms will be done as well as an optimization of the algorithms based on a test strategy selection heuristics. The heuristics are needed to achieve higher speed and better precision of detection as well as saving of resources. Also, the heuristics will help to prevent false positive detection which may occur in case of the algorithm which does not utilize the activation function modification. In addition, experiments with limited precision will be performed, as in case of classical neural networks, 16-bit fixed point precision was proven to be sufficient [6]. We have also designed modifications for both algorithm to be used to detect fault in both neurons basis functions and their activation functions.

Acknowledgment

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science – LQ1602, the BUT project FIT-S-17-3994 and the JU ECSEL Project SECREDAS (Product Security for Cross Domain Reliable Dependable Automated Systems), Grant agreement No. 783119.

References

[1] Girau, B.: FPNA: Concepts and Properties. In *FPGA Implementations of Neural Networks*, A. R. Omondi; J. C. Rajapakse, Springer US, 2006, ISBN 978-0-387-28487-3, pp. 71–123, http://dx.doi.org/10.1007/0-387-28487-7-3

[2] Prechelt, L. P.; Informatik, F. F.: — A Set of Neural Network Benchmark Problems and Benchmarking Rules. Technical report, Universitat Karlsruhe; 76128 Karlsruhe, Germany, 1994.

[3] Fast Artificial Neural Network Library (FANN). http://leenissen.dk/fann/wp/

[4] KRCMA Martin, KASTIL Jan a KOTASEK Zdenek: *Mapping trained neural networks to FPNNs.* In: IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems. Belgrade: IEEE Computer Society, 2015, pp. 157–160. ISBN 978-1-4799-6779-7.

[5] Krcma, M.; Kotasek, Z.; Kastil, J.: Fault tolerant Field Programmable Neural Networks. In *Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC), 2015*, Oct 2015, pp. 1–4, 10.1109/NORCHIP.2015.7364381.

[6] Holt, J.; Baker, T.: Back propagation simulations using limited precision calculations. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume II, July 1991, pp. 121 –126.

[7] Munakata, T.: Neural Networks: Fundamentals and the Backpropagation Model. In *Fundamentals of the New Artificial Intelligence*, editace T. Munakata, Texts in Computer Science, Springer London, 2007, ISBN 978-1-84628-839-5, pp. 7–36, http://dx.doi.org/10.1007/978-1-84628-839-5--2

[8] Kwan, H.: Simple sigmoid-like activation function suitable for digital hardware implementation. *Electronics Letters*, 1992: pp. 1379–1380. http://link.aip.org/link/?ELL/28/1379/1

[9] Deng, J.; Rang, Y.; Du, Z.; aj.: Retraining-based timing error mitigation for hardware neural networks. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, March 2015, s. 593–596.

[10] Elsimary, H.; Mashali, S.; Shaheen, S.: Generalization ability of fault tolerant feedforward neural nets. In *Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century., IEEE International Conference on*, Issue 1, Oct 1995, pp. 30–34 vol.1, 10.1109/ICSMC.1995.537728.

[11] Arad, B. S.; El-Amawy, A.: On Fault Tolerant Training of Feedforward Neural Networks. *Neural Networks*, Issue 10, vol. 3, 1997: pp. 539 – 553, ISSN 0893-6080, http://dx.doi.org/10.1016/S0893-6080(96)00089-5. http://www.sciencedirect.com/science/article/pii/S0893608096000895

[12] Ito, T.; Takanami, I.: On fault injection approaches for fault tolerance of feedforward neural networks. In *Test Symposium, 1997. (ATS '97) Proceedings., Sixth Asian*, Nov 1997, ISSN 1081-7735, pp. 88–93, 10.1109/ATS.1997.643927.

[13] Haruhiko, T.; Hidehiko, K.; Terumine, H.: Partially weight minimization approach for fault tolerant multilayer neural networks. In *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, vol. 2, 2002, ISSN 1098-7576, pp. 1092–1096, 10.1109/IJCNN.2002.1007646.

[14] Haruhiko, T.; Hidehiko, K.; Terumine, H.: Fault tolerant training algorithm for multi-layer neural networks focused on hidden unit activities. In *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, 2006, pp. 1540–1545, 10.1109/IJCNN.2006.246616.

[15] Hammadi, N. C.; Ito, H.: A Learning Algorithm for Fault Tolerant Feedforward Neural Networks. *IEICE Trans. Information and Systems*, Issue 80, 1996: pp. 21–27.

[16] Rusiecki, A.: Fault tolerant feedforward neural network with median neuron input function. *Electronics Letters*, Issue 41, vol. 10, May 2005: pp. 603–605, ISSN 0013-5194, 10.1049/el:20058169.

[17] Kamiura, N.; Taniguchi, Y.; Isokawa, T.; and col.: An improvement in weight-fault tolerance of feedforward neural networks. In *Test Symposium, 2001. Proceedings. 10th Asian*, 2001, ISSN 1081-7735, pp. 359–364, 10.1109/ATS.2001.990309.

[18] Sequin, C.; Clay, R.: Fault tolerance in artificial neural networks. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, June 1990, pp. 703–708 vol.1, 10.1109/IJCNN.1990.137651.

[19] Phatak, D.; Koren, I.: Complete and partial fault tolerance of feedforward neural nets. *Neural Networks, IEEE Transactions on*, Issue 6, vol. 2, Mar 1995: pp. 446–456, ISSN 1045-9227, 10.1109/72.363479.

[20] Tohma, Y.; Koyanagi, Y.: Fault-tolerant design of neural networks for solving optimization problems. *Computers, IEEE Transactions on*, Issue 45, vol. 12, Dec 1996: pp. 1450–1455, ISSN 0018-9340, 10.1109/12.545976.

[21] Zhou, Z.-H.; Chen, S.-F.; Chen, Z.-Q.: Improving tolerance of neural networks against multi-node open fault. In *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on*, Issue 3, 2001, ISSN 1098-7576, pp. 1687–1692 vol.3, 10.1109/IJCNN.2001.938415.

[22] Mahdiani, H. R.; Fakhraie, S. M.; Lucas, C.: Relaxed Fault-Tolerant Hardware Implementation of Neural Networks in the Presence of Multiple Transient Errors. *IEEE Transactions on Neural Networks and Learning Systems*, Issue 23, vol. 8, Aug 2012: pp. 1215–1228, ISSN 2162-237X, 10.1109/TNNLS.2012.2199517.

[23] Latif-Shabgahi, G.; Hirst, A.; Bennett, S.: A novel family of weighted average voters for fault-tolerant computer control systems. In *European Control Conference (ECC), 2003*, Sept 2003, pp. 642–646.

[24] Emmerson, M.; Damper, R.: Determining and improving the fault tolerance of multilayer perceptrons in a pattern-recognition application. *Neural Networks, IEEE Transactions on*, Issue 4, vol. 5, Sep 1993: pp. 788–793, ISSN 1045-9227, 10.1109/72.248456.

[25] Ahmadi, A.; Sargolzaie, M. H.; Fakhraie, S. M.; aj.: A Low-Cost Fault-Tolerant Approach for Hardware Implementation of Artificial Neural Networks. In *Computer Engineering and Technology, 2009. ICCET '09. International Conference on*, Issue 2, Jan 2009, pp. 93–97, 10.1109/ICCET.2009.204.

Paper E

Fault tolerant Field Programmable Neural Networks

M. Krcma, Z. Kotasek and J. Kastil, „Fault tolerant Field Programmable Neural Networks,“ *2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)*, 2015, pp. 1-4, doi: 10.1109/NORCHIP.2015.7364381.

Fault Tolerant Field Programmable Neural Networks

Martin Krcma, Zdenek Kotasek, Jan Kastil
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
Email: ikrcma@fit.vutbr.cz, kotasek@fit.vutbr.cz, ikastil@fit.vutbr.cz

***Abstract*—This paper describes a concept of Field Programmable Neural Networks (FPNNs) for artificial neural networks implementation in FPGAs, presents a model of fault tolerant FPNNs and different fault tolerance improving techniques based on the model. It describes an experiment based on one of these techniques and presents its results.**

I. INTRODUCTION

In the area of fault tolerant system design, three clearly distinguished groups of methodologies can be identified: a) the methodologies of constructing fault tolerant systems which guarantee that the system behaves as fault tolerant [4][7], b) the methodologies for detecting erroneous behavior of the system [6], c) the methodologies which allow the system to be recovered from the fault and reestablish its correct function [5].

In our previous research we dealt with all above mentioned methodologies [3]. Anyway, in the past we concentrated primarily on developing these methodologies for classical digital systems, while now we deal also with the design of fault tolerant neuron nets. It is a very specialized area which requires unique approaches to be used. The principles developed by us in this area so far belong to the area which is mentioned above as c). It means, we expect that the neuron net is designed as fault tolerant (e.g. as TMR system) and our methodology brings the TMR faulty component back to correct operation.

In this paper we focus on one of the possible implementations of neural networks in FPGAs - on Field Programmable Neural Networks (FPNNs) and ways how to bring a fault tolerance into design based on them. The paper is structured as follows - first we will describe the concept of FPNNs, next we will introduce a model of fault tolerant FPNNs and techniques based on this model and eventually we will present the results of one of our experiments with these techniques.

II. FPNNS

The concept of FPNNs [1] in design is meant to simplify the implementation of artificial neural networks in FPGAs by adjusting its properties to be more suitable for implementation into them. The simplification originates from its main feature - a highly customizable structure which makes it possible to establish resource sharing between the original synaptic connections of the neural network. This is done by using its customizability to simplify the interconnection model.

FPNAs are one of the possible implementations of neural networks in FPGAs. And just like others, they are vulnerable to various kinds of faults, the Single Event Upset (a change of a bit value caused by a ionizing particle) is the most impending one. The vulnerability is even higher since FPNNs are composed of a set of interconnected and interdependent dedicated units. On the other hand, this allows us to use a plenty of fault tolerance improving techniques. Some of them are presented in this paper.

The author originally defined the FPNNs formally [1] but now we shall describe it mostly in natural language in this paper. However, we will introduce a formal model, which we have based on the original definitions, in order to support our effort to bring the fault tolerance into FPNNs. Our definitions specify special derived types of units the FPNNs are composed of. They gave us proper instruments for further research leading to the design of fault tolerant FPGA artificial neural network architectures based on the FPNN concept as well.

A. FPNN

The FPNN is defined [1] as an oriented graph (N, E). Nodes (the set N) and edges (the set E) of this graph represent two different types of units. Nodes are called *activators* and represent original neural network neurons. Edges are called *links* and serve as an approximation of an original synaptic interconnection. Both types of units (together called *neural resources*) specify operators responsible for computation. Activators dispose of two operators. The first operator is an iterative operator i - a binary operator on the set of real numbers. This operator serves for iteration computing of an activator inner potential (similarly to a neuron potential). The first input of the operator is an activator input and the second input is an inner accumulator variable. The second activator operator is a function operator f - an unary operator on the set of real numbers. The operator performs an activator output computation - it computes an activation function (similarly to a neuron). The input of this operator is the value of the inner accumulator variable. The concrete functions of the operators are not predefined, only the conditions that they have to be binary and unary real functions. Then, there is a set of affine operators for every link serving as an approximation of original synaptic weights. There is one affine operator for every activator predecessor. The links dispose of a set of affine operators (1) performing a transformation of an input x using two real constants W and T for the original neural network weight approximation. Weights are approximated by the sequence of one or more affine transformations performed by affine operators in a sequence of connected links. In case, that all $T_n(p) = 0$, the approximation is done by a sequence of multiplications. In [2] we have described one of the possible ways of transformation of neural network weights to affine operators.

$$\alpha_{(p,n)} = W_n(p) \times x + T_n(p); W_n(p), T_n(p) \in \mathbb{R};$$
$$p, n \in N; (p, n) \in E \quad (1)$$

Also, the data interconnection between neural resources are specified using binary flags determining interconnections between particular neural resources. We are allowed to connect not only links to activators and activator to links, but even to connect links to other links. This is a new property regarding the neural networks, the possibility of connecting links to other links and to construct sequences of interconnected links is the core feature allowing us to construct FPNNs with very various structures. Using this we are able to fit more effectively an FPNN structure for implementation in FPGAs.

If only a graph (N, E), iteration, function and affine operators are specified and data interconnection structure is left unspecified, the result object is called Field Programmable Neural Array (FPNA) [1]. This object specifies the whole class of possible FPNNs. Adding the structure specification and other details like default value of the inner accumulator variable and number of i operator iterations allows us to create FPNNs with different configurations.

Since an FPNN could be structurally different object from the original neural network, it can have different parameters and it generally can be constructed as an object of lower power than the original network, we can generally say that the FPNN is an approximation of the original network. The approximation can have a different accuracy [2].

B. Grid FPNN

A special type of FPNN can now be defined using previous definitions. *Grid FPNN* is an FPNN with an enforced limitation of the structure causing it to form a grid shape. The reason for this is to make an FPNN suitable for implementation in FPGAs due to the similarity of the grid FPNNs structure and FPGAs interconnection bus.

An example of a grid FPNN can be seen in Fig. 1. The circles on the figure represent activators, wide arrows represent links and the thin arrows represent data interconnections. The orientation of the connection arrows show the way of the passing data. As the picture illustrates, there is only one link on the output of every activator which realizes the connection to another layer. It is directly connected to one successive activator in the next layer. The connection to the other activators goes through two sequence of links going the opposite ways within the whole layer.

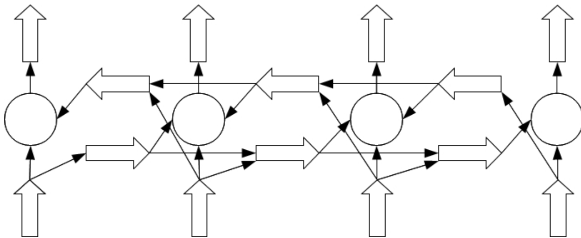

Fig. 1. A grid FPNN

C. FPNN operation

The communication and computation model [1] of the FPNNs used in this paper is asynchronous (synchronous FPNNs have been designed [1] as well). It is based on the request-acknowledgement model. The neural resources within a whole FPNN generate requests for all the connected succeeding units once they finish the output value computation. Then they wait until all the successors process the requests and send the acknowledgements. All acknowledgements have to be obtained before the new iteration of computation starts. So, all the neural resources work as follows:

1) Wait for the request from the predecessors.
2) Select a request from all the waiting requests.
3) Process the request and compute the new output.
4) Generate requests for all successors.
5) Wait for the acknowledgements from all the successors.
6) Go back to 1.

This model makes FPNNs to be flexible, easily constructable and extensible. However, it also brings an overhead in space and time complexity due to resources needed for an implementation of the communication.

III. FAULT TOLERANCE

In this paper we would like to present a formal model of fault tolerant neural resources allowing us to use different techniques to ensure a fault tolerance of an FPNN.

Definition III.1 (Fault tolerant link)**.** is a set $rLink^n = \{S, s, R, U, I, m, D\}$, where:

- S is a set of link settings (sets of parameters)
- $R = \{link_1, link_2, .., link_n\}$ is a set of n identical links with settings $s \in S$
- $U = \{0, 1\}^n$ is a set of binary flags determining the links activity
- I is an identity operator (x is a link input): $I(x) = x; x \in \mathbb{R}$
- m is a majority level
- $D \in \{f, r, i\}$ is a link mode:
  - f - output is taken from the first active link
  - r - output is taken as a majority of the first m active links
  - i - output is taken from the identity operator

Definition III.2 (Fault tolerant activator)**.** is a set $rLink^n = \{S, s, R, U, C, m, c, D\}$, where:

- S is a set of activator settings (sets of parameters)
- $R = \{link_1, link_2, .., link_n\}$ is a set of n identical activators with settings $s \in S$
- $U = \{0, 1\}^n$ is a set of binary flags determining the activators activity
- C is a constant operator (x is a activator input): $C(x) = c; x, c \in \mathbb{R}$
- m is a majority level

- $D \in \{f, r, i, t\}$ is an activator mode:
  - f - output is taken from the first active activator
  - r - output is taken as a majority of the first m active activators
  - t - output is taken from the constant operator

$$TMRLink = \{\{s\}, s, \{l_1, l_2, l_3\}, \{1, 1, 1\}, \emptyset, 2, r\} \quad (2)$$

The definitions allows us to use different techniques to secure neural resources against faults. It allows us to use unit-based redundancy techniques (such as TMR) and techniques without unit-based redundancy. The neural resources based on the definitions are composed of a set R of n identical neural resources with identical settings s. The s is a set of all neural resources parameters (W and T values for links, f, i, a, θ for activators). The set of binary flags U determines the activity of the particular resources in the R set. The positive flags determine the active resources. Using the set U, we are able both to switch between resources and have resources working concurrently. The way, how the final neural resource output is constructed from the active resources in the R is based on a neural resource mode. The mode is specified by the tuple last member. Both fault tolerant link and activator have f and r mode. In the f mode, the output is taken from the first active resource in the R set. So, in this mode we are able to use the resources in the R set as backup units and switch between them in case of a fault. In the mode r, the neural resource is taken as a majority of m (the majority level) first active links in the R set. So, in this mode we are able to realize techniques such as TMR. The example of TMR secured link is in equation (2). So, the combinations of usage of R and U sets and f and r modes allow us to use unit-based redundancy techniques.

Techniques without unit-based redundancy are based on the different principles. The first principle is switching between neural resources settings. The settings are determined by the member $s \in S$. The set S contains possible neural resource settings. It allows us to switch between different settings in case of a fault. For example, if one of the resource output bits is hardly set due to a fault, we can compensate it by a change of the settings. We can also use this principle to compensate other neural resources faults. So, in case of our example, we may be able to compensate the error even more by a change of a successive link settings. Basically, the s and S is a dynamic reconfiguration formalization.

The second unit redundancy free technique is based on two new operators - an identity and constant operator. These operators allows us to secure an FPNN against the unpredictable effect of a faulty neural resource. It allows us to stabilize the state of the FPNN. The main idea is to force some value to the faulty neural resource output instead leaving its output affected by the fault. So this technique does not solve the fault but it offers a way, how to lower the fault impact until the recover is done. For this purposes, the fault tolerant links dispose of the identity operator which is activated in the i mode. An identity operator makes a resource to be transparent to the passing data, i.e. it turns the resource in to a register. This however negatively affects the passing data, since the link W parameter is removed from the multiplication sequence. In some cases (according to the weights distribution) we are able to lower this negative effect by switching the settings of the other neural resources. And, if we anticipate a failure of some particular link, we might be able to intercorporate this prediction into the final FPNN during the mapping of an original neural network into it - we can map it as if the secured link was already gone (transparent) and final FPNN will be configured to work without it. We have described the process of mapping of neural networks to FPNNs in [2]. According to the weights distribution, this step can negatively influence the approximation accuracy in both the faulty and faulty free state. So, this method is usable only in some cases. The experimental results of this technique are presented in the next section.

Fault tolerant activators dispose of the second operator, the constant operator. This operator is activated in the t mode, and it forces the value of c to the activator output. So, it turns it to a constant register. In this case we can use statistics of the activator output and assign for example the most common value (or average of the most common values, or limit values of an activation function) to the c to ensure that et least in some cases, the data will not be affected by the fault and the constant operator.

In case of all presented techniques, the other advantage of the asynchronous communication model comes to place. That is, that in case of the possible usage of the technique to solve or compensate some fault, the mid-state during the recover does not harm the the computation of other neural resources neither the synchronization of the FPNN. The only negative effect is that other neural resources have to wait until the recover is done. This delays the FPNN computation by the time of the recover, but the result of the computation shall be correct, not affected by the recover itself.

IV. Experimental results

In this section we present an experiment dealing with an influence of mapping neural network to an FPNN [2] in order to decrease a negative influence of identity operator activation according to the principle presented in the previous section. We have experimented with the very basic neural network task which allows us, due to its simplicity, to perform an experiment with many combinations of faults and security. All links dispose of only one affine operator in this FPNN.

The task is a logical exclusive addition - XOR. The original neural network and derived grid FPNN have 2 inputs, 3 neurons in a hidden layer and 1 output. As a first step, we measured the influence of link failures to the correctness of the FPNN classification of all four input vectors. Table I summarizes the results. The first column contains the name of faulty links, the next two columns contain the numbers of correctly and incorrectly classified inputs. The next column shows the percentage rate of correctness. The last column indicates if the link can be secured by the technique. The faulty links were supposed to be treated as transparent (i.e. their identity operators were activated). The link names are derived from the names of activators ($n1 ... n6$) they connect, as $(source, destination)$, for example $(n4, n6)$.

As the table shows, two link failures ($(n4, n6)$,$(n4, n3)$) did not have an effect on the FPNN output. But the remaining seven failures caused an output error. It can be seen that links $(n1, n3)$,$(n2, n5)$ in the first layer caused the highest error.

Missing resource	Correct	Incorrect	Match [%]	Possible to secure
-	4	0	100	-
(n4,n6)	4	0	100	No
(n4,n3)	4	0	100	Yes
(n3,n4)	3	1	75	Yes
(n4,n5)	3	1	75	Yes
(n3,n6)	3	1	75	No
(n5,n4)	3	1	75	Yes
(n1,n3)	2	2	50	No
(n2,n5)	2	2	50	No
(n5,n6)	2	2	50	No

TABLE I. FAILURES EFFECT ON THE XOR FPNN

This is expected since they lay at the beginning of the FPNN and thus have a high influence on the rest of it. Also, one of the links in the last layer caused the same error. However, all these links cannot be secured by mapping since they have no link predecessors [2]. Other links caused a smaller error and three of them ($(n3, n4)$,$(n4, n5)$,$(n5, n4)$) can be secured using the presented technique.

We tried to secure these links. First, only one link at the time was secured. Next, we tried to secure combinations of two links and finally the combination of all three links. In all experiments we ran the FPNN with and without failure of the secured links in all possible combinations. Table II summarizes the results of these experiments.

Secured resources	Faulty resources	Correct	Incorrect	Match [%]
(n4,n5)	-	4	0	100
(n4,n5)	(n4,n5)	4	0	100
(n3,n4)	-	3	1	75
(n3,n4)	(n3,n4)	2	2	50
(n5,n4)	-	3	1	75
(n5,n4)	(n5,n4)	3	1	75
(n4,n5),(n3,n4)	-	4	0	100
(n4,n5),(n3,n4)	(n4,n5)	4	0	100
(n4,n5),(n3,n4)	(n3,n4)	4	0	100
(n4,n5),(n3,n4)	(n4,n5),(n3,n4)	4	0	100
(n4,n5),(n5,n4)	-	3	1	75
(n4,n5),(n5,n4)	(n4,n5)	4	0	100
(n4,n5),(n5,n4)	(n5,n4)	3	1	75
(n4,n5),(n5,n4)	(n4,n5),(n5,n4)	3	1	75
(n3,n4),(n5,n4)	-	3	1	75
(n3,n4),(n5,n4)	(n3,n4)	2	2	50
(n3,n4),(n5,n4)	(n5,n4)	3	1	75
(n3,n4),(n5,n4)	(n3,n4),(n5,n4)	2	2	50
(n3,n4),(n4,n5), (n5,n4)	-	3	1	75
(n3,n4),(n4,n5), (n5,n4)	(n3,n4)	2	2	50
(n3,n4),(n4,n5), (n5,n4)	(n4,n5)	3	1	75
(n3,n4),(n4,n5), (n5,n4)	(n5,n4)	3	1	75
(n3,n4),(n4,n5), (n5,n4)	(n3,n4),(n4,n5)	2	2	50
(n3,n4),(n4,n5), (n5,n4)	(n4,n5),(n5,n4)	3	1	75
(n3,n4),(n4,n5), (n5,n4)	(n3,n4),(n5,n4)	2	2	50
(n3,n4),(n4,n5), (n5,n4)	(n3,n4),(n4,n5), (n5,n4)	2	2	50

TABLE II. FAILURES EFFECT ON THE XOR FPNN

As the table shows, in five cases the technique really increased the influence of the identity operators activation. In some cases the technique did not help and in some cases it led to an even higher error. It also caused new errors which occurred in the faulty free state. Both findings were expected since the result of this technique depends on the original weight distribution as mentioned above.

V. CONCLUSIONS AND FUTURE RESEARCH

In this paper we have followed the FPNNs author's original formal model and derived the model of the fault tolerant neural resources. We have presented the techniques of fault tolerance ensurance based on the described models. It allow us to use unit based redundancy using fault tolerant techniques such as TMR or backup copies. The techniques based on the identity and the constant operator do not use the unit based redundancy and offer the way how to potentially lower the influence of a fault. The presented experiment showed, that in some cases, the usage of link identity operator may lead to increase of fault tolerance if we perform the mapping of a neural network to an FPNN with assumption of the fault. It also proved that this technique, because its usability depends on the weight distribution, can have a negative effect on the FPNN in both a fault-free and faulty state.

In future work, we are going to perform more practical experiments with presented models and techniques and to develop them. These efforts should lead to a design of a fault tolerant neural network architecture implemented in FPGAs using FPNNs and offering suitable features for implementing deep neural networks.

ACKNOWLEDGMENT

This work was supported by the following projects: National COST LD12036 - "Methodologies for Fault Tolerant Systems Design Development, Implementation and Verification", project Centrum Excelence IT4Innovations (ED1.1.00/02.0070), EU COST Action IC1103 - MEDIAN - Manufacturable and Dependable multIcore Architectures at Nanoscale and BUT project FIT-S-14-2297.

REFERENCES

[1] Girau, B.: FPNA: Concepts and Properties. In *FPGA Implementations of Neural Networks*, editace A. R. Omondi; J. C. Rajapakse, Springer US, 2006, ISBN 978-0-387-28487-3, p. 63–136, 10.1007/0-387-28487-7-3. http://dx.doi.org/10.1007/0-387-28487-7-3

[2] KRCMA Martin, KASTIL Jan a KOTASEK Zdenek: *Mapping trained neural networks to FPNNs.* In: IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems. Belgrade: IEEE Computer Society, 2015, pp. 157–160. ISBN 978-1-4799-6779-7.

[3] STRAKA Martin, KASTIL Jan, KOTASEK Zdenek a MICULKA Lukas: *Fault Tolerant System Design and SEU Injection Based Testing*. Microprocessors and Microsystems. Amsterdam: Elsevier Science, 2013, issue 37, pp. 155–173. ISSN 0141-9331.

[4] J. A. Cheatham, J. M. Emmert, and S. Baumgart: *A survey of fault tolerant methodologies for fpgas*, ACM Trans. Des. Autom. Electron. Syst., vol. 11, no. 2, pp. 501–533, 2006.

[5] M. G. Gericota, L. F. Lemos, G. R. Alves, and J. M. Ferreira: *Online self-healing of circuits implemented on reconfigurable fpgas*, in IOLTS '07: Proceedings of the 13th IEEE International On-Line Testing Symposium. Washington, DC, USA: IEEE Computer Society, 2007, pp. 217–222.

[6] G. Yan, Y. Han, and X. Li: *SVFD: A Versatile Online Fault Detection Scheme via Checking of Stability Violation*, IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 19, pp. 1627–1640, Sept. 2011.

[7] U. Sharma, *Fault tolerant techniques for reconfigurable platforms*, in A2CWiC '10: Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India. New York, NY, USA: ACM, 2010, pp. 1–4.

Paper F

Triple modular redundancy used in field programmable neural networks

M. Krcma, Z. Kotasek and J. Lojda, „Triple modular redundancy used in field programmable neural networks," *2017 IEEE East-West Design & Test Symposium (EWDTS)*, 2017, pp. 1-6, doi: 10.1109/EWDTS.2017.8110128.

Triple Modular Redundancy Used in Field Programmable Neural Networks

Martin Krcma, Zdenek Kotasek and Jakub Lojda
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
ikrcma@fit.vutbr.cz, kotasek@fit.vutbr.cz, ilojda@fit.vutbr.cz

Abstract

This paper presents the concepts of FPNA and FPNN, used for the approximation of artificial neural networks in FPGAs and discusses the usage of TMR technique in order to reach a fault tolerance. The schemes of the FPGA implementation are presented. The results of experiments determining the FPGA resources utilization with different usage of the TMR technique are provided.

1 Introduction

The artificial neural networks [10] are one of the important models of softcomputing and artificial intelligence. They are structure inspired by the human brain with high capability of learning and memorizing to solve various types of tasks. Basically, the goal of the artificial neural network is to learn the relation between two sets of data vectors, to generalize the relation, to determine its features and use it for the determining the relation of the unknown vectors belonging to the same problem. This capability can be used for classification tasks, for timeseries and functional prediction, to control tasks, to image recognition, clustering and other tasks.

Neural networks are composed of a set of *neurons* computing the *activation function* over the *basis* function (often the weighted sum) of their inputs. The neurons are interconnected with the weighted connections called *synapses*. The learning of the neural network is basically a process of setting the weights.

The networks have been implemented in various kinds of devices starting from analog computers to the most modern processors, VLSIs, graphical processing units and FPGAs. This paper deals with one of the possible implementations of artificial neural networks in FPGAs - Field Programmable Neural Arrays/Networks (FPNAs/FPNNs).

In our paper published at NORCAS 2015 conference [8], we described the concept of Field Programmable Neural Networks for artificial neural networks implementation in FPGA. We also presented a model of fault tolerant FPNNs and various fault tolerance improving techniques based on the model. Experimental results were also provided.

This paper is organised as follows - the first section introduces the FPNA/FPNN concept. The second section describes the implementation of FPNNs into FPGAs. The third section deals with fault tolerance techniques. The fourth section presents the experimental results and the last section summarizes the whole paper.

2 Field Programmable Neural Networks

The concept of FPNNs [4] is meant to simplify the implementation of artificial neural networks in FPGAs by adjusting their properties to be more suitable for implementation into them. The simplification originates from its main feature - a highly customizable structure which makes it possible to establish resource sharing between the original synaptic connections of the neural network and to simplify the interconnection model. The FPNNs are composed of dedicated interconnected units called neural resources which approximate the original neurons and synaptic interconnections. The units of the first type are called *activators* and represent the original neural network neurons. The other units are called *links* and serve as an approximation of the original synaptic interconnection. Every link disposes of a set of weights serving as an approximation

of the original synaptic weights.

An example of a grid FPNN can be seen in Fig. 1. The circles in the figure represent activators, wide arrows represent links and the thin arrows represent data interconnections. The orientation of the connection arrows shows the way of the passing data. The straight wide dashed/dotted arrows represent the original neural networks synapses. The thin dashed/dotted arrows represent the sequences of links approximating the particular synapses. The synapses and the particular sequences are drawn with the same line and arrow styles.

The FPNNs are not the same structures as neural networks, although they can be constructed in that way. The FPNNs represent a different model which can structurally differ from the implemented neural network. They can also have different capabilities which means that they are not only an implementation of the neural networks, they are an approximation of neural networks as well - with different structure and properties, they can provide similar results as the networks. The accuracy is the main problem here. Since the FPNNs can be constructed in various ways and types, the approximation accuracy can be different. We dealt with the approximation accuracy in [7].

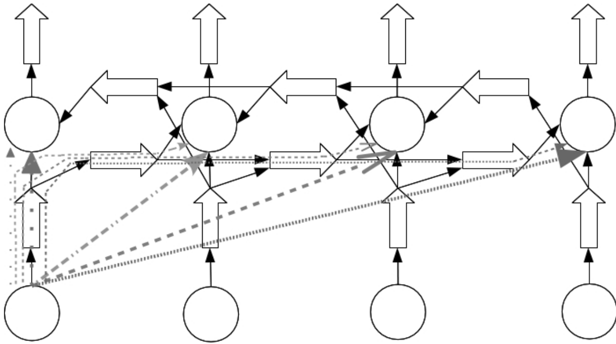

Figure 1. Synapses approximation in a grid FPNN

2.1 Implementation of FPNNs into FPGAs

The VHDL implementation of both types was created according to the original design and schematic [4]. Both, activators and links were designed as separated units communicating with signals. The communication is based on the asynchronous *request - acknowledgement* model. Every neural resource generates requests for all units directly connected to its output (successors) when its computation is done. Once a successor starts to process the request, it sends the acknowledgement back to the original resource. When the original resource receives acknowledgements from all successors, it selects a new input request to process, sends the acknowledgement and begins the computation. The activators also send a *flag* together with the requests. The flag is a constant activator number and it is used in links to select the proper weight to multiply the input data width. The links then propagate the flag to all connected links.

The implementations of both types of neural resources are similar, however they differ in used computational units. The scheme of standard link implementation is illustrated in Fig. 2 and the scheme of the activator in Fig. 3. Both types are composed of a multiplexor, demultiplexor, register, computation units and units for processing requests. The meaning of common units is described bellow:

- **SELECT** selects one of the active requests for processing using the *Round&Robin* algorithm. The requests from preceding neural resources are indicated by the set bits on its input. When the request is selected, it sets the *start* signal up.

- **MUX** is an input data vectors multiplexer. It is controlled by the SEL unit.

- **REG** is a register storing the selected data vector.

- **ACK_DEMUX** delivers an acknowledgement (generated by the *start* signal) to the proper predecessor. It is controlled by the SEL unit.

These units are present in both links and activators. They serve for input requests processing and delivery of the input data to the computation part of the unit. Computation part of links and activators is composed of different units:

- **MULT_ADD** applies the weights to the data. The key to select the proper weight is the flag associated with the request. The flag is selected from all of the flags at the input $FLAG_IN$ by the value at the input s.

- **ITER** iteratively computes the sum of all input data (simulates the neuron basis function). After a predefined number of iterations, it transmits the result to the TRANS unit and activates it using the fin signal. After every iteration it activates the *next* signal which starts the processing of another request.

- **TRANS** computes the activation function (the output of the activator). The input is gained from

the ITER unit. The activation function were sigmoid like function suitable for hardware implementation [9].

All computation units take the input data from the register REG, perform the computation of the result and transmit it to the neural resource output. They also activate the signal *ready* which is an input of the output requests generators:

- **LINK_REQ_GEN** generates the requests to the connected successors when the *ready* signal is set. It also receives the acknowledgements from the successors. Using the *free* signal it controls the SEL block - it enables (when all acknowledgements are received) or disables (new request was selected - *start* signal is up) its function.
- **ACT_REQ_GEN** is similar to the LINK_REQ_GEN, but it allows to activate the *free* signal using the *next_req* signal without the requests generation.

These units are responsible for the control of the neural resource. When the processing of the selected request is started, they block the SEL unit preventing it from selecting another request before the actual one is processed. After the computation is done, they generate output requests and hold the entire neural resource inactive until all requests are successfully received by the successors.

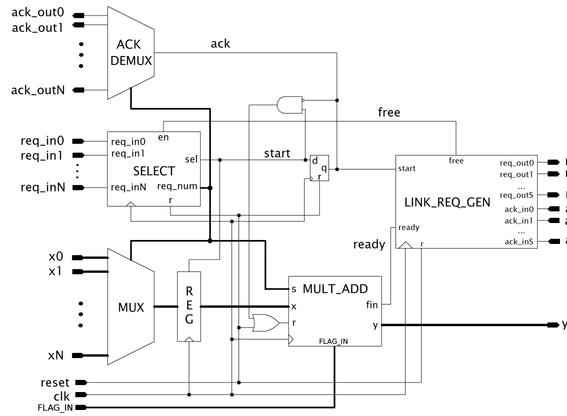

Figure 2. Scheme of a link implementation - the interconnection of the inner units

3 Fault tolerant FPNN using TMR

The neural networks are parallel structures with lot of redundancy performing an approximate (soft) computing. Therefore they dispose of inherent fault tolerant properties which differ with every network though. There are a number of techniques designed to increase the fault tolerant properties. Some do it using an additional redundancy on different levels (for example TMR on the level of the whole network [13], or adding redundant neurons [1]) in order to build the network to be fault tolerant. Others modify the process of training [2] in order to train the network to be fault tolerant (for example [5] uses weight minimization during the training) or use retraining after a fault occurs [3]. Others modify the basis function [11] or activation function [12]. All approaches are combined as well.

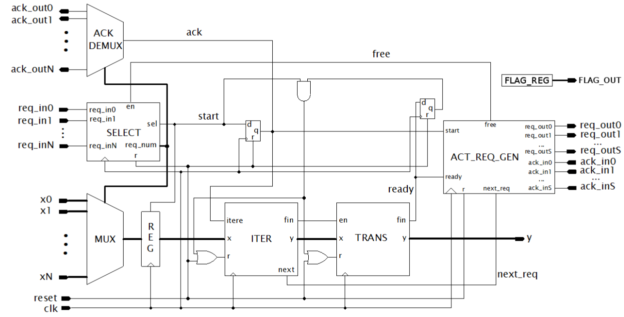

Figure 3. Scheme of an activator implementation - the interconnection of the inner units

Our approach based on extension of the implemented neural network fault tolerance (or substitution if no fault tolerant technique was used on the network). Different approaches can be used to make an FPNN fault tolerant. The approaches can utilize replication or can use other principles. We introduced a fault tolerance technique which does not use replication in [8]. The homogeneous structure of FPNNs is suitable for using replication based techniques as well as for the recovery using the online reconfiguration. The asynchronous model of communication is suitable for this type of recovery as well since the FPNN can be simply put on hold until the recovery is finished and then resume its function without a need of resetting the whole FPNN.

TMR is a well known fault tolerant technique based on triple replication of the secured unit and comparison of the triplet output data in order to determine the major result which is then used as the output of the whole triplet. This technique can be used on different levels with FPNNs. In this paper we focus on two levels - the level of inner units (we will refer to this type as *type A*) and the level of the whole neural resources (*type B*).

On the level of inner units (type A), there are six (link) or seven (activator) main units which can be secured using the TMR technique. Using the technique on this level has several advantages. It allows us to choose which units (if not all) will be secured. Therefore it allows us to adjust the security/overhead ratio. Also, if we focus on the smaller inner units, the recovery from fault using the dynamic reconfiguration will be easier and faster than in the case of reconfiguration of the whole neural resources. However, on this level the interconnection between units will not be secured. This will occur on the level of the whole neural resources (type B). On this level, the fault tolerance will be generally higher because of duplication the whole resources but overhead will be higher as well. Also, recovery from fault using dynamic reconfiguration will be more complicated and slower due to larger reconfigured area.

We decided to compare these two levels in the meaning of area utilization in order to have a base for decision which level of TMR will be used. There are other criteria to evaluate the fault tolerant techniques. The power consumption, maximum clock frequency and latency belong to the most important. However, in this paper we deal with area (resource utilization) only, with other criteria we shall deal with in our future research.

4 Experimental results

In order to determine the area usage (in the number of slice registers and LUTs) of the neural resources and their inner units in both the unsecured and the TMR versions, we implemented them in VHDL and synthesized them using the Xilinx ISE 14.7 tool. The target FPGA was the Xilinx Virtex-6 device *xc6vlx240t-1-ff1156*. All computations were implemented in fixed point form with 8 bits of the integer part and 8 bits of the fractional part [6]. The voters were implemented using bit operations, therefore the voting was performed on the level of bits. All neural resources were implemented to be connected with three predecessors and two successors. The number of connected neural resources affects the size of the communication units. The link has three weights with real values. The use of DSP blocks was switched off. The optimization level was left on default but the *Equivalent registers removal* option was switched off to avoid the drop of the duplicated units. All results were provided by the synthesis only.

The resources utilization of the unsecured units are shown in Table 1. In the table, the *Unit* column identifies the units by their name. The columns *Slice Regs.* and *Slice-LUTs* contain the resources utilization. The columns *LUTs of act.* and *LUTs of link* compare the LUTs utilization of the unit with the utilization of the whole neural resources in order to illustrate the area portion of the inner units.

Table 1. Utilization of unsecured blocks

Unit	Slice Regs.	Slice-LUTs	LUTs of act.	LUTs of link
SELECT	8	17	1%	1%
ITER	49	104	6%	0%
TRANS	1	1478	86%	0%
MULT-_ADD	0	1389	0%	88%
REQ-_GEN	2	5	0.3%	0.3%
ACTI-VATOR	126	1723	100%	0%
LINK	57	1582	0%	100%

As the table illustrates the most significant portion of FPGA resources are utilized in the computation units MULT_ADD and TRANS. The communication and control units utilize around one percent of resources and around 5%-10% of resources are utilized by the units interconnection. This shows that the computation units are the best candidates for using the TMR technique as the probability of failure is the highest with them. On the other hand, the communication and control units are essential for the data flow, therefore for the functionality of the whole FPNN and the failure in these units could stop the operation of the whole FPNN, while the failure in the computing unit could only cause the degraded precision. Moreover according to their low resources utilization, it could be suitable to apply the TMR technique to secure them.

Table 2 summarizes the resource utilization of the building blocks secured using the TMR technique. The columns *Registers* and *LUTs* have the same meaning as in Table 1, the other columns contain the percent increase of the resources utilization. As expected, the resource utilization has increased approximately three times or less in most of the units. The neural resources are marked by the used TMR type. The utilization of the type A activator (all units were TMR secured although not all of them are listed in the table) has increased around 2.7 times and the utilization of the type A link around 2.9 times (registers) and 2.3 times (LUTs). The type B resources LUTs utilization has increased even more but their registers utilization has increased less than in case of type A resources.

Table 3 compares the utilization of neural resources

Table 2. Utilization of TMR-secured blocks

Secured unit	Registers	LUTs	Increase of Regs.	Increase of LUTs
SELECT	28	40	250%	135%
ITER	179	331	265%	218%
TRANS	83	4291	83	190%
MULT-_ADD	83	3641	83	162%
REQ-_GEN	6	6	200%	20%
ACTIV-ATOR_-TYPE_A	346	4699	174%	173%
LINK_-TYPE_A	166	3737	191%	136%
ACTIV-ATOR_-TYPE_B	311	5611	147%	226%
ACTIV-ATOR_-TYPE_B	163	4299	185%	172%

Table 3. Comparison of different TMR levels

Neural resource (secured using type B)	Increase of registers utilization vs. type A	Increase of LUTs utilization vs. type A
Link	-2%	15%
Activator	-11%	19%

secured by the both TMR types. As the table illustrates, the type B neural resources consumes less registers than neural resources of type A. This is due to the number of voters consuming the registers in the type A resources. However, the type B neural resources consume more LUTs. This is due to interconnection included into the duplicates. It is needed to consider that there is around twice more registers than LUTs available in the FPGA. From this point of view the type A neural resources seems to be more resource and area efficient than type B resources. However, the type B resources are secured including the interconnection between inner units, therefore their fault tolerance should be higher.

5 Conclusions and future research

In this paper we briefly described the concept of FPNN serving for the implementation of artificial neural networks in FPGAs. We also described the implementation using the schematics and explained the construction of neural resources and the communication model. The fault tolerance techniques were considered and two levels of application of the TMR technique on the neural resources were discussed.

The application of the TMR technique on the inner units of the neural resources (type A) has proven to be less consuming in the meaning of the number of consumed LUTs, although this type consumed more registers. However, the registers are more available than LUTs, so this type seems to be more resource efficient. Due to smaller areas secured using TMR it might be more effective to use the dynamic reconfiguration in order to recover from fault.

The type of the TMR that secures the whole neural resources (type B) consumes less registers but more LUTs. In the meaning of the available resources, this type is less resource effective. Also, using the dynamic reconfiguration to recover from fault might be less effective and slower due to larger area needed to be reconfigured.

In our future research we will deal with other fault tolerance techniques. Especially with techniques which do not use the replication but they are based on a change of parameters and on the robustness of the FPNN which we designed. We shall also perform experiments with fault injection. We shall measure how the techniques affect the consumption and the value of the frequency on which the system works as well.

Acknowledgement

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602, ARTEMIS JU under grant agreement no 641439 (ALMARVI) and BUT project FIT-S-14-2297.

References

[1] A. Ahmadi, M. H. Sargolzaie, S. M. Fakhraie, C. Lucas, and S. Vakili. A low-cost fault-tolerant approach for hardware implementation of artificial neural networks. In *Computer Engineering and Technology, 2009. ICCET '09. International Conference on*, volume 2, pages 93–97, Jan 2009.

[2] B. S. Arad and A. El-Amawy. On fault tolerant training of feedforward neural networks. *Neural Networks*, 10(3):539 – 553, 1997.

[3] J. Deng, Y. Rang, Z. Du, Y. Wang, H. Li, O. Temam, P. Ienne, D. Novo, X. Li, Y. Chen, and C. Wu. Retraining-based timing error mitigation for hardware neural networks. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, pages 593–596, March 2015.

[4] B. Girau. Fpna: Concepts and properties. In A. R. Omondi and J. C. Rajapakse, editors, *FPGA Implementations of Neural Networks*, pages 63–101. Springer US, 2006. 10.1007/0-387-28487-7-3.

[5] T. Haruhiko, K. Hidehiko, and H. Terumine. Fault tolerant training algorithm for multi-layer neural networks focused on hidden unit activities. In *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, pages 1540–1545, 2006.

[6] J. Holt and T. Baker. Back propagation simulations using limited precision calculations. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume ii, pages 121 –126 vol.2, jul 1991.

[7] M. Krcma, J. Kastil, and Z. Kotasek. Mapping trained neural networks to fpnns. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2015 IEEE 18th International Symposium on*, pages 157–160, April 2015.

[8] M. Krcma, Z. Kotasek, and J. Kastil. Fault tolerant field programmable neural networks. In *Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC), 2015*, pages 1–4, Oct 2015.

[9] H. Kwan. Simple sigmoid-like activation function suitable for digital hardware implementation. *Electronics Letters*, 28(15):1379–1380, 1992.

[10] T. Munakata. Neural networks: Fundamentals and the backpropagation model. In T. Munakata, editor, *Fundamentals of the New Artificial Intelligence*, Texts in Computer Science, pages 7–36. Springer London, 2007. 10.1007/978-1-84628-839-5–2.

[11] A. Rusiecki. Fault tolerant feedforward neural network with median neuron input function. *Electronics Letters*, 41(10):603–605, May 2005.

[12] Y. Taniguchi, N. Kamiura, Y. Hata, and N. Matsui. Activation function manipulation for fault tolerant feedforward neural networks. In *Test Symposium, 1999. (ATS '99) Proceedings. Eighth Asian*, pages 203–208, 1999.

[13] F. Zarafshan, G. Latif-Shabgahi, and A. Karimi. Notice of retraction a novel weighted voting algorithm based on neural networks for fault-tolerant systems. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 9, pages 135–139, July 2010.

Paper G

Fault tolerant Field Programmable Neural Networks

M. Krcma, Z. Kotasek and J. Kastil, „Fault tolerant Field Programmable Neural Networks,“ *2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)*, 2015, pp. 1-4, doi: 10.1109/NORCHIP.2015.7364381.

Light grid FPNNs and Fault Tolerant Mapping

Martin Krcma, Jan Kastil, Zdenek Kotasek
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
Email: ikrcma@fit.vutbr.cz, ikastil@fit.vutbr.cz, kotasek@fit.vutbr.cz

***Abstract*—This paper presents formal definitions of FPNA and FPNN concepts and introduces new types of FPNN derived and used by author. The process of mapping a trained artificial neural network to FPNNs is described. Techniques of redundancy free fault tolerance of the selected parts of FPNN are introduced, the derived algorithms are presented and the experimental results of this algorithm are summarized.**

I. Introduction

The concept of the Field Programmable Neural Arrays (FPNAs) [1] is in design meant to simplify the implementation of artificial neural networks in FPGAs by adjusting its properties to be more suitable for implementation in them. The simplification originates from its main feature - highly customizable structure which makes possible to establish a resource sharing between the original synaptic connections of the neural network. This is done by using its customizability to simplify the interconnection model.

FPNAs are one of the possible implementation of neural networks in FPGAs. And just like others, they are vulnerable to various kinds of faults, the SEU is the most impending one. The vulnerability is even higher since FPNNs are composed of a set of interconnected and interdependent dedicated units. On the other hand, this allows us to use a plenty of fault tolerance improving techniques Since the main goal of FPNN is the FPGAs resources savings, it is appropriate to avoid the use of redundancy techniques and exploit other techniques preserving this core property. One of the possible methods is presented in this paper.

The author originally hase defined the FPNNs in quite formal way in his work [1]. I have reformulated the original definitions (see definiton I.1 and I.2) to suit us further definitions which we have based on them. These definitions, we are introducing in this paper, specify special types of FPNNs and allow us to easily describe the algorithms of fault tolerant mapping we are presenting. It also gave us proper instruments for further research leading to the design of fault tolerant FPGA artificial neural network architectures based on the FPNA/FPNN concept.

A. FPNA

The FPNA is defined (definition I.1) [1] as an oriented graph. Nodes and edges of this graph represent two different types of unit. Nodes are called *activators* and represent original neural network neurons. Edges are called *links* and serve as an approximation of an original synaptic interconnection. Both types of units (together called *neural resources*) specify operators responsible for computation. Activators dispose of an iterative operator i serving for activator potential computing (similarly to a neuron potential). The Second activator operator is a function operator f performing an activator output computation, i.e. the computation of an activation function over the potential. Then, there is a set of affine operators for every link serving as an approximation of original synaptic weights. There is one affine operator for every activator predecessor.

Definition I.1 (FPNA [1])**.** Let N be a set of nodes (*activators*) and E a set of oriented edges (*links* denoted as $(m, n); m, n \in N$). We say that graph (N, E) is an *FPNA* if the following statements hold:

1) Every node has a set of predecessors: $\forall n \in N : \exists Pred(n) = \{p \in N \mid (p, n) \in E\}$
2) Every node has a set of successors: $\forall n \in N : \exists Succ(n) = \{s \in N \mid (n, s) \in E\}$
3) A set of input nodes exists: $\exists N_i = \{n \in n \mid Pred(n) = \emptyset\}$
4) Every link has an affine operator (x is an input data): $\forall p, n \in N \wedge p \in Pred(n) : \exists \alpha_{(p,n)} = W_n(p) \times x + T_n(p)$
5) Every non-input activator has an iterative operator: $\forall n \in N \setminus N_i : \exists i_n : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
6) Every non-input activator has a function operator: $\forall n \in N \setminus N_i : \exists f_n : \mathbb{R} \to \mathbb{R}$

The definition specifies the existence of neural resources, declare their operators and determines the topology. However, it does not declare the affine operator parameters, starting value of an iteration operators and it does not specify the full structure of the resulting object from the data connections point of view. Thus, the FPNA specifies a whole class of possible designs, and to obtain a fully specified and implementable object there is need for something more. The remaining specification is offered by a Field Programmable Neural Network - FPNN.

B. FPNN

An FPNN (definition I.2) [1] is one of the possible FPNA instances. It defines concrete values of W and T parameters of affine operators. It also defines a starting value θ of an activator iteration operator and a number of iterations a. For every input node it gives a number of input batches c. That is because input vectors can be separated to parts and fed to an FPNN part by part saving the number of necessary inputs in this way.

FPNN also specifies the data interconnection between neural resources. It uses four types of binary flags to do that. The nterconnections between links and successive activators

is determined by r flags. The s flags define interconnections between activators and successive links. Similarly, the S flags determine interconnections between input nodes and successive links.

The last type of flag, R, specifies interconnection between two links. This is a new property regarding the neural networks, the possibility of connecting links to other links, to construct sequences of interconnected links. This is the core feature allowing to construct FPNNs with very various structures, thus structures suitable for implementation in FPGAs.

Definition I.2 (FPNN [1])**.** Let N be a set of nodes (*activators*) and E a set of oriented edges (*links* denoted as $(m, n); m, n \in N$). We say that graph (N, E) is an *FPNN* if the following statements hold:

1) (N, E) is an FPNA
2) A default value of an activators iteration variable exists: $\forall n \in N : \exists \theta_n \in \mathbb{R}$
3) A number of iterations is defined:
$\forall n \in N : \exists a_n \in \mathbb{N}$
4) Concrete values of affine operators parameters are defined:
$\forall \alpha_{(p,n)} : W_n(p), T_n(p) \in \mathbb{R}$
5) Number of inputs for every input node is defined:
$\forall n \in N_i : \exists c_n \in \mathbb{N}$
6) For every activator n a binary flag determining interconnection with a link (p, n) exists:
$\forall p, n \in N \wedge p \in Pred(n) : \exists r_n(p) \in \{0, 1\}$
7) For every activator n a binary flag determining interconnection with a link (n, s) exists:
$\forall n, s \in N \wedge s \in Succ(n) : \exists s_n(s) \in \{0, 1\}$
8) For every link (p, n) a binary flag determining interconnection with a link (n, s) exists:
$\forall n, p, s \in N \wedge p \in Pred(n) \wedge s \in Succ(n) : \exists R_n(p, s) \in \{0, 1\}$
9) For every input node n a binary flag determining interconnection with a link (n, s) exists:
$\forall n \in N_i, \forall s \in N \wedge s \in Succ(n)) : \exists S_n(s) \in \{0, 1\}$

C. Grid FPNN

For further definition purposes some auxiliary variables are defined (definition I.3). These variables contain information about numbers of connected neural resource to a particular resource the variable is related to.

Definition I.3 (FPNN structural variables)**.** Structural variables are variables related to individual neural resources and containing numbers of connected resources derived from the number of the positive structural flags (i.e. $r_n(p), s_n(s), R_n(p, s)$ and $S_n(s)$ flags). Four types of structural variables are defined as follows:

1) num_r_n and num_s_n variables for every activator:
$\forall n \in N : \exists num_r_n = |\{r_n(p)|r_n(p) = 1 \wedge p \in Pred(n)\}|$
$\forall n \in N : \exists num_s_n = |\{s_n(s)|s_n(s) = 1 \wedge s \in Succ(n)\}|$
2) $num_R_n(p)$ variable for every link:
$\forall (p, n) \in E : \exists num_R_n(p) = |\{R_n(p, s)|R_n(p, s) = 1 \wedge n \in Pred(s) \wedge (n, s) \in E\}|$
3) num_S_n variable for every input node:
$\forall n \in N_i : \exists num_S_n = |\{S_n(s)|S_n(s) = 1 \wedge s \in N\}|$

A special type of FPNN can now be defined using previous definitions. *Grid FPNN* (definition I.4) is an FPNN with enforced limitation of the structure causing it has to form a grid shape. The reason of this is to make FPNN suitable for implementation in FPGAs due to similarity of grid FPNNs structure and FPGAs interconnection bus.

Definition I.4 (Grid FPNN)**.** Let N be a set of nodes (*activators*) and E a set of oriented edges (*links* denoted as $(m, n); m, n \in N$). We say that graph (N, E) is n *grid FPNN* if the following statements hold:

1) (N, E) is an FPNN
2) A limited number of connected preceding links is defined:
$\forall n \in N : num_r_n \geq 2$
3) A limited number of connected successive links is defined:
$\forall n \in N : num_s_n \in \{0, 1\}$
$\forall n \in N : num_S_n \in \{0, 1\}$
$\forall (p, n) \in E : num_R_n(p) \in \{0, 1, 2\}$
4) A limited number of iterations is defined: $\forall n \in N : a_n \geq 2$
5) A limited number of inputs is defined: $\forall n \in N_i : c_n = 1$

The example of grid FPNN is in Fig. 1. The circles on the figure represent activators, wide arrow represent links and thin arrow represent data interconnections. The orientation of the connection arrows show the way of the passing data. As the picture illustrates, there is only one link on the output of every activator which realizes the connection to another layer. It is directly connected to one successive activator in the next layer. The connection to the other activators goes through the sequence of links within the whole layer. The are two sequences of the links going the opposite ways. They are called *Interconnection train* (definitions I.5-I.9). Every layer with more than one activator has an interconnection chain within.

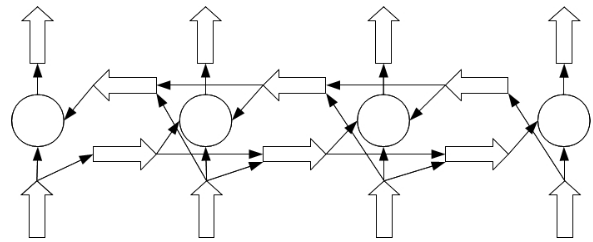

Fig. 1. A grid FPNN

Definition I.5. *Train of links* is generally a sequence of interconnected links.

Definition I.6. *Initial* is a link having no link predecessors. It has only an activator predecessor.

Definition I.7. *Terminal* is a link having no link successors. It has only an activator successor.

Definition I.8. *Chain of links* is a train of links bordered by initial (the beginning of the chain) and terminal (the end of the link) respectively.

Definition I.9. *Interconnection train* is a train of links interconnecting activators within a layer. It is composed of two trains going the opposite ways.

Before introducing another special type of FPNNs, the new type of links has to be established. The definition I.10 specifies a *light link*. The difference between a standard link and the light link is that the light link disposes of only one affine operator. This leads to another spare of FPGA resources because universal multiplier, which the standard link has to contain, can be replaced by a less expensive constant multiplier in the light link.

Definition I.10 (Light link)**.** Let (N, E) be an FPNN and $(p, n) \in E$ a link. We say that (p, n) is a *light link* if it dispose of only one affine operator which is common for all predecessors (x is an input data):
$\forall p, n \in N \wedge (p, n) \in E : \alpha_{(p,n)} = W \cdot x; W \in \mathbb{R}$

Now the core type of FPNN of this paper can be introduced. It is a special type of grid FPNN called a *Light grid FPNN* (definition I.11) and it is composed only of light links. This type is oriented on maximal spare of FPGAs resources using both principles described above.

Definition I.11 (Light grid FPNN)**.** Let N be a set of nodes (*activators*) and E a set of oriented edges (*links* denoted as $(m, n); m, n \in N$). We say that graph (N, E) is an *light grid FPNN* if the following statements hold:

1) (N, E) is a grid FPNN
2) all $e \in E$ are the light links

II. Mapping

Mapping is a process of direct transfer of an artificial neural network into an FPNN without using a training data set and without the need of learning. Mapping uses information obtainable from an original neural network such as weights, biases, activations functions and the network structure.

A. Theory

Considering the declarations in table I, the artificial neural network will be seen in this paper as a double composed of set of neurons and set of synapses. Every object in these sets represents an original object in the original network and its related properties such as weights, biases and activation functions (i.e. syn_{val}, neu_{act} and syn_{biases} in table I respectively).

In light grid FPNN, every original synapse is approximated with sequence of multiplications performed by some chain of links (1). The chain of links is determined by $approx$ function (table I).

$$\forall s \in Synapses : s_{apx} = \prod_{e \in approx(s)} W_e \quad (1)$$

There is a set of chains of different length approximating the synapses. Every chain has to be fully specified (i.e every multiplicands in a sequence has to be known) for equation (1) to hold. For computation, it is possible to exploit the fact that in the light grid FPNN the links can be part of more than one chain, ad thus we are able to compute the last multiplicand in chain multiplication sequence using one link shorter chain. In this case only a chain terminal (syn_{last}) has to be computed (2). So, if mapping begins on the chains with length 1, which are present in every grid FPNN, the mapping process can be effectively performed with one link longer chain in every step leading to fully mapped FPNN.

$$\forall s \in Synapses : s_{apx} = \left(\prod_{e \in approx(s) \setminus syn_{last}} W_e \right) \times syn_{last} \quad (2)$$

Since the multiplication values of the rest of the chain are known, the syn_{last} can be computed as the division of the original synapse weight and the product of all the previous multiplicands in a sequence (3).

$$\forall s \in Synapses : syn_{last} = \frac{s_{val}}{\prod_{e \in approx(s) \setminus syn_{last}} W_e} \quad (3)$$

Since the light link can approximate more than one synapse ($\forall e \in E : |apprSyns_e| \geq 1$), there is more than one syn_{last} values for it. However, the light links dispose of only one W parameter, so there is a need for compromise. This compromise can be computed in different ways [3], in this paper the arithmetic average will be used.

B. Fault tolerance

There are many popular ways to enhance a fault tolerance of various types of technology. In this paper we are proposing a method increasing the light grid FPNNs fault tolerance without using redundancy. The new type of link is defined (definition II.1) for the purposes of this method. It adds new operators to the standard or light link.

Definition II.1 (Fault tolerant link)**.** Let (N, E) be an FPNN and $e \in E$ a link. We say that e is a *Fault tolerant link* if the following statements hold:

1) it dispose of an identity operator (x_e is e input data): $\exists I_e; I_e : \mathbb{R} \to \mathbb{R}; I_e(x_e) = x_e$
2) there is a binary flag $useI_e \in \{0, 1\}$ determining the activity of the identity operator (y_e is e output data): $y_e = I_e(x_e) \Leftrightarrow useI_e = 1$

The new operator is an identity operator. If this operator is active it replaces a link affine operator on a computation of an output data and it simply copies the link input data to its output. Using this technique the identity operator makes the link to be transparent for the passing data. So, if the link is faulty, the activation of this operator ensures the propagating data against unpredictable effect of the defective link.

Even if we secure the propagating data against unknown defects caused by faulty link, a problem of invalid data (a chain is broken and one of the elements of a multiplication sequence is missing) passing to the rest of the FPNN remains. To ensure as correct FPNN computation results as possible we need to

settle this problem in some way. Since we do not intend to use redundancy, we have no other choice than to use other neural resources to compensate the missing (transparent) link. The best candidates for this are the direct link predecessors of the faulty link. If we anticipate the possible failure of some particular link, or if we decide that one of links is more important than others for some reason and we want to ensure the FPNN against its failure, we might be able to delegate the link function to its predecessors in advance. Thus compute the failure of the link to the parameters of its predecessors. This can be done by considering the link just not be there. Then the predecessors are forced to approximate the link synapses instead of it. So, basically the predecessors take a part of a link $apprSyns$ set and intercorporate it into their own $apprSyns$ sets. Then, during the process of mapping, the approximated synapses of the selected neuron are approximated by its predecessors. But since these synapses are together with the original predecessor synapses, the approximation accuracy suffers from a decrease caused by this sharing. And the accuracy of the approximation of the original synapses is degraded as well. So this method should be applied only if the fault tolerance and FPGA resources savings is so important to the user that a decrease of the accuracy is acceptable. Also, it is applicable only on link having link predecessors, i.e. not initials.

Another problem is how to pick the link to be ensured with this method. We can use different metrics. For example we can consider links approximating synapses with high weights to be more important then links approximating lower weighted synapses. Or link being placed closer to the beginnings of chains can be seen as more important as the higher number of successive links depends on them compared to terminals for example. Also links having more predecessors can be regarded as more important, and so on. The next section introduces an algorithm of selecting links for fault tolerance ensuring as well as the mapping algorithm using the previously described technique to mapping neural network to FPNN with securing selected links against faults.

C. Algorithms

The presented principles are implemented in the following algorithms which perform a fault tolerant mapping. Algorithm 1 is one of the possible algorithm for link fault tolerant importance ranking. The resulting ranking is the addition of a sum of all link approximated synapses weights and an inverse proportion sum of all chains the link is part of. This ranking method comes from an idea that a link is more important if more successive links depend on it, i.e. the longer parts of the chains rests on it. And also link is more important if it approximates a higher weights, i.e. has a bigger impact in the result.

After the ranking is done, the initialization must be done (Algorithm 2). A graph of the link interconnection is constructed making links to be new nodes and interconnecting those nodes with new edges according to original link to link connections ($R_n(p,s)$ flags). This graph is separated to its components, where every component represents link interconnection within one layer. Next, the links intended to be secured are selected and stored in a SEL set. Then all the synapse variables are initialized, val variable gets and original synapse weight, $prod$ and $aprx$ gets 1.0 (multiplication neutral element).

As a next step, the $apprSyns$ sets are actualized (Algorithm 3). The approximated synapses of links in SEL set are copied to their chain predecessors $apprSyns$ sets.

Finally, the mapping is done (algorithm 5). It it performed layer by layer. In every layer the set of chains is determined (algorithm 4). Links are divided into two sets with respect to their input and output degree - links with zero input degree (no predecessors) are the first links in chains and are stored in the $first$ set and the links with a zero output degree (no successors) are the last links in chains and stored in the $last$ set. Next, all chains existing between all the links in $first$ and $last$ sets are determined and stored in $chains$ set together with all their subsets (chains between links from the sets are the longest chains within a layer. Thus, their subsets make all possible chains. At least the chains are sorted by length in the ascending order to make possible to start the mapping with the shortest chains.

Algorithm then iterate over all the chains. Every chain is fully mapped before the algorithm moves to the next one. First link of every chain is picked and all its syn_{last} are computed. Then a compromise is find and used as new value for link W parameter. The illustrated algorithm uses an arithmetic average (11th line) as the compromise. When W is known, it is used for actualization of partial products values ($prod$ variables) of synapses passing trough the link, i.e. longer chains. Finally the computed link is removed from the chain and the algorithm can continue with another link in the chain.

It is needed to be mentioned, that the mapping algorithm is the universal algorithm for mapping neural networks to FPNNs. It only performs the mapping using the input sets and variables bud it does not improve the fault tolerance itself. That is done by the algorithm 3 which perform expansion of secured links predecessors approximated synapses sets.

The presented mapping algorithm can be easily used even for the standard FPNNs mapping with only two changes needed to be done. The equation on the 11th line of the algorithm has to be replaced with equation 4. These equations does not use any compromise and assign the value directly since there is affine operator for every predecessor. Second, the equation on the 13th line of the algorithm (actualization of the synapses product variables) has to be switched to the equation 5. The algorithm 3 can be used without any changes.

$$W_n(p) = syn_{last} \Leftrightarrow srcNeuron(syn) = p \wedge dstNeuron(syn) = n \quad (4)$$

$$syn_{prod} = syn_{prod} \times W_n(p) \Leftrightarrow srcNeuron(syn) = p \wedge dstNeuron(syn) = n \quad (5)$$

III. Experiments

We have experimented with the presented algorithms and in this paper we are going to present results from experiments over the very basic neural networks task which allows us, due to its simplicity, to perform an experiment with many combinations of faults and security. The task is logical exclusive

Declaration	Description
$NN = (Neurons, Synapses)$	an input neural network
(N, E)	light an grid FPNN
$NeuToAct : Neurons \rightarrow N$	Mapping neurons to activators.
$ActToNeu = NeuToAct^{-1}$	Inverse mapping.
$srcNeuron : Synapses \rightarrow Neurons$	Source neuron of a synapse.
$dstNeuron : Synapses \rightarrow Neurons$	Destination neuron of a synapse.
$approx : Synapses \rightarrow E^n$	Determination of chain of links apprixmating a synapse.
$sortByLength : E^n \rightarrow E$	Sorting by path length.
$findChain : E \times E \rightarrow E^n, n \geq 1$	Find path.
$firstNodeOf : E^n \rightarrow E$	Chain's first node.
$\forall u \in E : \exists apprSyns_u \subset Synapses$	Set of synapses ending in the link.
$\forall u \in E : \exists passSyns_u \subset Synapses$	Set of synapses passing trough the link.
$\forall u \in E : \exists connPred_u \subset N$	Set of connected preceding activators.
$\forall neu \in Neurons : \exists neu_{act} : \mathbb{R} \rightarrow \mathbb{R}$	Neurons activation functions.
$\forall neu \in Neurons : \exists neu_{bias} \in \mathbb{R}$	Neurons biases.
$\forall syn \in Synapses : \exists syn_{val} \in \mathbb{R}$	Synapses weight.
$\forall syn \in Synapses : \exists syn_{prod} \in \mathbb{R}$	Partial product approximating the synapse.
$\forall syn \in Synapses : \exists syn_{apx} \in \mathbb{R}$	The value of synapse approximation.
$\forall syn \in Synapses : \exists syn_{last} \in \mathbb{R}$	Last multiplicand of approximation product.
$chains \subset E^*$	an ordered collection of all chains.
$belongTo : E \rightarrow \{E^n\}^m$	Set of chains the link belongs to.
$lengthOfChain\,(E^n) = n$	Length of chain function.
$\forall link \in E : \exists rank_{link} \in \mathbb{R}$	Link's fault tolerant ranking.

TABLE I. DECLARATIONS

1: **procedure** RANKLINKS($NN, FPNN$)
2: **for all** $\forall link \in E$ **do**
3: $rank_1 \leftarrow \sum^{syn \in apprSyns_{link}} syn_{val}$
4: $rank_2 \leftarrow \sum^{c \in belongTo(link)} lengthOfChain(c)$
5: $rank_{link} \leftarrow rank_1 + \frac{1}{rank_2}$
6: **end for**
7: **end procedure**

Algorithm 1. Ranking algorithm

addition - XOR. As the first step, we measured the influence of link failures to the correctness of the FPNN classification of all four input vectors. Table II summarizes the results. The first column contains the name of faulty links, the next two columns contain the numbers of correctly and incorrectly classified vectors. The next column shows the percentage rate of correctness. The last column says if the link can be secured by fault tolerant mapping. The faulty links were supposed to be treated as transparent, i.e. their identity operator were activated.

Missing resource	Correct	Incorrect	Match [%]	Possible to secure
-	4	0	100	-
(n4,n6)	4	0	100	No
(n4,n3)	4	0	100	Yes
(n3,n4)	3	1	75	Yes
(n4,n5)	3	1	75	Yes
(n3,n6)	3	1	75	No
(n5,n4)	3	1	75	Yes
(n1,n3)	2	2	50	No
(n2,n5)	2	2	50	No
(n5,n6)	2	2	50	No

TABLE II. FAILURES EFFECT ON THE XOR FPNN

As the table shows, two link failures ((n4,n6),(n4,n3)) did not have an effect on the FPNN output. But the remaining seven failures caused the output error. It can be seen that initials in the first layer ((n1,n3),(n2,n5)) caused the highest error. This is expected since they lay in the beginning of the FPNN and thus have a big influence on the rest of it. Also one of the links in the last layer caused the same error. However, all these link cannot be secured by mapping since they have no link predecessors. Another links caused a smaller error and three of them ((n3,n4),(n4,n5),(n5,n4)) can be secured using the presented algorithms.

We tried to secure these links. First, we secured only one

1: **procedure** INITIALIZE($NN, FPNN$)
2: Construct a graph of the link connection
3: Separate components of the graph and store them in a $CONN$ set.
4: Select the links intended to have a non-redundant security and store them in a SEL set.
5: **for all** $\forall syn \in Synapses$ **do**
6: $syn_{val} \leftarrow original_weight$
7: $syn_{prod} \leftarrow 1.0$
8: $syn_{apx} \leftarrow 1.0$
9: **end for**
10: **end procedure**

Algorithm 2. Initialization algorithm

1: **procedure** EXPANSE($NN, FPNN$)
/* Expansion of predecessors in the $apprSyns$ sets of all selected links: */
2: **for all** $(U, V) \in CONN$ **do**
3: **for all** $(n, s) \in SEL \wedge (n, s) \in U$ **do**
4: **for all** $(p, n) \in Pred(n) \wedge R_n(p, s)$ **do**
5: **for all** $\forall syn \in apprSyns_{(n,s)})$ **do**
6: **if** $\exists act \in connPred_{(p,n)} \wedge ActToNeu(act) = srcNeuron(syn)$ **then**
7: $apprSyns_{(p,n)} \leftarrow apprSyns_{(p,n)} \cup syn$
8: **end if**
9: **end for**
10: **end for**
11: **end for**
12: **end for**
13: **end procedure**

Algorithm 3. Expansion algorithm

1: **function** DETERMINECHAINS((U, V))
2: $first \leftarrow \{u \in U | deg^+(u) = 0\}$
3: $last \leftarrow \{u \in U | deg^+(u) = 0\}$
4: $chains \leftarrow \emptyset$
5: **for all** $\forall u \in first$ **do**
6: **for all** $\forall v \in last$ **do**
7: $p \leftarrow findChain(u, v)$,
8: $chains \leftarrow chains \cup 2^p$,
9: $sortByLength(chains))$
10: **end for**
11: **end forreturn** $chains$
12: **end function**

Algorithm 4. Chain determination algorithm

1: **procedure** MAPLIGHTFPNN($NN, FPNN$)
2: INITIALIZE()
3: EXPANSE()
4: **for all** $(U, V) \in CONN$ **do**
5: $chains \leftarrow$ DETERMINECHAINS((U, V))
`/* Mapping path by path: */`
6: **for all** $\forall r \in chains$ **do**
7: $(p, n) \leftarrow firstNodeOf(r)$
`/* Multiplicands computation: */`
8: **for all** $\forall syn \in apprSyns_{(p,n)}$ **do**
9: $syn_{last} \leftarrow \frac{syn_{val}}{syn_{prod}}$
10: **end for**
`/* Computing a link W: */`
11: $W_{(p,n)} \leftarrow \frac{\sum^{syn \in apprSyns_{(p,n)}} syn_{last}}{|apprSyns_{(p,n)}|}$
`/* Updating the products: */`
12: **for all** $syn \in passSyn_{(p,n)}$ **do**
13: $syn_{prod} \leftarrow syn_{prod} * W_{(p,n)}$
14: **end for**
`/* Deleting finished node from the path: */`
15: $r \leftarrow r \setminus \{(p, n)\}$
16: $chains \leftarrow \{p | p \in chains \wedge p \neq \emptyset\}$
17: **end for**
18: **end for**
19: **end procedure**

Algorithm 5. Light FPNN mapping algorithm

singular link at the time. Next, we tried to secure combinations of two links and finally we tried the combination of all three links being secured. In all experiments we ran the FPNN with and without failure of the secured links in all possible combinations. Table III summarizes the results of these experiments.

Secured resources	Faulty resources	Correct	Incorrect	Match [%]
(n4,n5)	-	4	0	100
(n4,n5)	(n4,n5)	4	0	100
(n3,n4)	-	3	1	75
(n3,n4)	(n3,n4)	2	2	50
(n5,n4)	-	3	1	75
(n5,n4)	(n5,n4)	3	1	75
(n4,n5),(n3,n4)	-	4	0	100
(n4,n5),(n3,n4)	(n4,n5)	4	0	100
(n4,n5),(n3,n4)	(n3,n4)	4	0	100
(n4,n5),(n3,n4)	(n4,n5),(n3,n4)	4	0	100
(n4,n5),(n5,n4)	-	3	1	75
(n4,n5),(n5,n4)	(n4,n5)	4	0	100
(n4,n5),(n5,n4)	(n5,n4)	3	1	75
(n4,n5),(n5,n4)	(n4,n5),(n5,n4)	3	1	75
(n3,n4),(n5,n4)	-	3	1	75
(n3,n4),(n5,n4)	(n3,n4)	2	2	50
(n3,n4),(n5,n4)	(n5,n4)	3	1	75
(n3,n4),(n5,n4)	(n3,n4),(n5,n4)	2	2	50
(n3,n4),(n4,n5),(n5,n4)	-	3	1	75
(n3,n4),(n4,n5),(n5,n4)	(n3,n4)	2	2	50
(n3,n4),(n4,n5),(n5,n4)	(n4,n5)	3	1	75
(n3,n4),(n4,n5),(n5,n4)	(n5,n4)	3	1	75
(n3,n4),(n4,n5),(n5,n4)	(n3,n4),(n4,n5)	2	2	50
(n3,n4),(n4,n5),(n5,n4)	(n4,n5),(n5,n4)	3	1	75
(n3,n4),(n4,n5),(n5,n4)	(n3,n4),(n5,n4)	2	2	50
(n3,n4),(n4,n5),(n5,n4)	(n3,n4),(n4,n5),(n5,n4)	2	2	50

TABLE III. FAILURES EFFECT ON THE XOR FPNN

As the table shows in five cases the mapping really increased the tolerance of the secured links faults. In some other case the mapping did not help and in some cases it led to even higher error. Also it caused new errors showing in the state without failure. Both findings are expectable since the FPNN is the light grid FPNN suffering of sharing affine operators between multiple synapses. However, the results show that fault tolerant mapping can lead in some cases to real increase of the fault tolerance.

IV. CONCLUSION AND FUTURE RESEARCH

In this paper we have followed the FPNNs author's original formal model and derived the model of the new type of FPNN - the light grid FPNN. We have based algorithms on this model serving for the mapping trained neural networks to the standard or the light grid FPNNs. Beside mapping, these algorithm are able to secure an FPNN against failure of some link only using other links and without the usage of any redundancy.

This method was the core of the experiments presented in this paper. During these experiments we tried to secure the light grid FPNN implementing the logical exclusive addition against failure of almost all securable links. In some cases we observed an actual increase of fault tolerance, in other cases it remained the same or even getting worse. In some cases the usage of the method led to worse functionality of the FPNN in the failure free state. This was expexted due to the fact the FPNN is the light grid FPNN which have smaller strength than standard FPNN. However, the presented method proved to be able to increase an FPNN fault tolerance in some cases and thus, it can be used for this purpose.

In the future work, we are going to perform more experiments with bigger FPNNs and the standard FPNNs as well. Also we are going to design other redundancy free methods of securing the neural networks and FPNN against failures. We are going to examine method using redundancy as well. These efforts should lead to a design of a fault tolerant neural network architecture implemented in FPGAs using FPNNs and offering suitable features for implementing deep neural networks.

ACKNOWLEDGMENT

This work was supported by the following projects: National COST LD12036 - "Methodologies for Fault Tolerant Systems Design Development, Implementation and Verification", project Centrum excelence IT4Innovations (ED1.1.00/02.0070), EU COST Action IC1103 - MEDIAN - Manufacturable and Dependable multIcore Architectures at Nanoscale and BUT project FIT-S-14-2297.

REFERENCES

[1] Girau, B.: FPNA: Concepts and Properties. In *FPGA Implementations of Neural Networks*, editace A. R. Omondi; J. C. Rajapakse, Springer US, 2006, ISBN 978-0-387-28487-3, p. 63–136, 10.1007/0-387-28487-7-3. http://dx.doi.org/10.1007/0-387-28487-7-3

[2] Krcma, M.: *The neural networks acceleration in FPGA*. Master's thesis, Faculty of Information Technology, Brno University of Technology; Brno, 2014. https://wis.fit.vutbr.cz/FIT/st/rp.php/rp/2013/DP/15754.pdf

[3] Krcma, M.: *Mapping trained neural networks to FPNNs*. Paper accepted on DDECS 2015 conference, Faculty of Information Technology, Brno University of Technology; Brno, 2014.

Paper H

Implementation of fault tolerant techniques into FPNNs

M. Krcma, Z. Kotasek and J. Lojda, „Implementation of fault tolerant techniques into FPNNs,“ *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 297-298, doi: 10.1109/FPT.2016.7929559.

Implementation of Fault Tolerant Techniques into FPNNs

Martin Krcma, Zdenek Kotasek, Jakub Lojda
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
Email: ikrcma@fit.vutbr.cz, kotasek@fit.vutbr.cz, ilojda@fit.vutbr.cz

***Abstract*—This paper presents concepts of FPNN which can be used for the implementation of artificial neural networks in FPGAs and introduces fault tolerant techniques applied on this concept that are developed by the authors.**

I. Introduction

The artificial neural networks [4] are one of the important models of softcomputing and artificial intelligence. They are structures composed of *neurons* interconnected by weighted *synapses*. Basically, the goal of the networks is to learn the relation between two sets of data vectors, to generalize the relation, to determine its features and to use it for the determining the relation of the unknown vectors belonging to the same problem. This capability can be used for classification tasks, for time series and functional prediction, to control tasks, to image recognition, clustering and other tasks.

The implementation of neural networks is challenged with two great neural networks complexities - space complexity and time complexity. The usual solution of both is to use a powerful hardware, such as graphical processor units or processor clusters, which suffer from a high power consumption. For some networks, FPGAs can be one of the possible solutions if a lower power consumption is required. In this case, the time complexity is solvable by parallelism which is easy to be achieved in both FPGAs and neural networks since both are parallel by their nature. The space complexity is bigger problem since an FPGA has limited resources. Thus, there is a need for such designs that exploit the neural networks parallel character for fast computations and save the FPGA resources as well. A Field Programmable Neural Networks (FPNN) concept can be seen as one of the possible solutions. The goal of this paper is to describe the types of FPNNs and compare their capabilities.

II. Field Programmable Neural Networks

The concept of FPNNs [1] is meant to simplify the implementation of artificial neural networks in FPGAs by adjusting their properties to be more suitable for implementation into them. The simplification originates from its main feature - a highly customizable structure which makes it possible to establish resource sharing between the original synaptic connections of the neural network. The FPNNs are composed of dedicated interconnected units called neural resources which approximate the original neurons and synaptic interconnections. The units of the first type are called *activators* and represent the original neural network neurons. The other units are called *links* and serve as an approximation of the original synaptic interconnection. Every link disposes of a set of affine operators serving as an approximation of the original synaptic weights.

An example of a grid FPNN can be seen in Fig. 1. The circles in the figure represent activators, wide arrows represent links and the thin arrows represent data interconnections. The orientation of the connection arrows shows the way of the passing data. The straight wide dashed/dotted arrows represent the original neural networks synapses. The thin dashed/dotted arrows represent the sequences of links approximating the particular synapses. The synapses and the particular sequences are drawn with the same line and arrow styles.

The FPNNs are not the same structures as neural networks, although they can be constructed in that way [2]. The FPNNs represent a different model which can structurally differ from the implemented neural network. They can also have different capabilities which means that they are not only an implementation of the neural networks, they are an approximation of neural networks as well - with different structure and properties, they can provide similar results as the networks. The accuracy is the main problem here.

The approximation capabilities depend on the number of affine operators belonging to links. This number depends on the FPNN structure directly. However, the model can be altered to dispose of different number of affine operators. Two different models with different approximation capabilities exist. The original model disposes of as many affine operators as the number of directly connected preceding units. These operators are shared between groups of synapses approximated by the particular preceding units. This type of an FPNN is called *Standard FPNN*. A stronger model was derived that has the number of affine operators that allows it to reach the precise approximation accuracy. This type of an FPNN is called *Full FPNN*. In case of a full FPNN, every link disposes of dedicated affine operator for every synapse it approximates. There is no sharing of affine operators between synapses, therefore the accurate approximation is ensured. Although, this type of FPNN demands more FPGA resources.

III. Fault tolerance

The present research is dealing with a fault tolerance of neural networks implemented using the FPNN concept and

978-1-5090-5602-6/16/$31.00 © 2016 IEEE

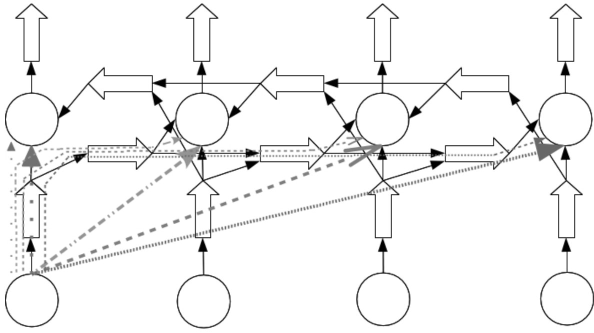

Fig. 1. Synapses approximation in a grid FPNN

developing new methods to improve the fault tolerance. The well known redundancy based techniques such as Triple Modular Redundancy (TMR) are considered. These techniques are well usable in many levels - on the level of neural resources, on the level of their inner implementation or on the level of the FPNN itself. However, at present the research focuses on methods that do not use this kind of redundancy (replication). Instead the goal is to use the FPNN parameters to partially or completely mask faults. In this paper, the methods are briefly described.

The first method relies on the inherit robustness of FPNN. It uses two operators. The *identity operator* has a simple function - it turns a faulty neural resource to the transparent register. The dataflow and the synchronization are restored and the computation continues. However, the neural resource is missing from the computation sequence. Depending on the missing resource parameters and the parameters of other resources in the sequence, the approximation accuracy of the original network is decreased. In some cases, the decrease is only marginal, in some other cases, the impact could be critical. The method decreasing the negative influence of identity operators usage on links was developed [3].

The other operator is a *constant operator*. It turns the neural resource to a constant register (the synchronization signals pass transparently). This operator takes advantage from the fact that in many neural networks neurons which have the similar output in the majority cases exist. In case of fault if the neuron (activator) is switched to the constant register with the most frequent value, the network will compute properly in the major cases (related to the value). The principle of switching neural resource is illustrated in Fig. 2.

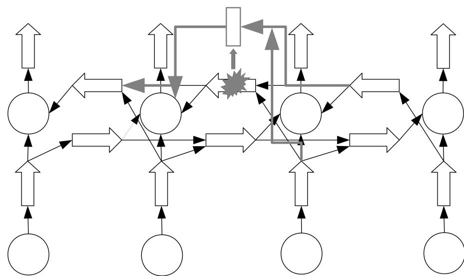

Fig. 2. Illustration of the identity operator principle

The second technique is based on changing the settings of neural resources laying in the same sequence as the faulty resource (especially resources lying before the faulty one). With the proper change it might be possible to mask or partially compensate the fault impact (or the impact of operators usage). However, there are many possible modification of parameters and according algorithms is under development. The principle is illustrated in Fig. 3. In the figure, the gray resources are the resources preceding the faulty one in the sequence i.e. the most probable candidates to apply changes to.

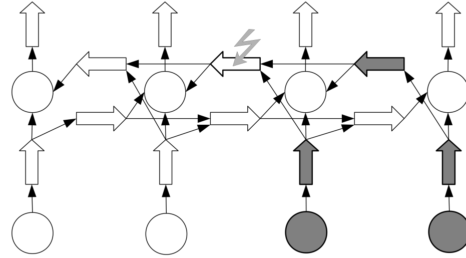

Fig. 3. Illustration of the parameters change principle

IV. Conclusions and Future Research

In this paper, the FPNN concept was described. It is the concept of resource saving implementation of neural networks in FPGAs which can serve as an approximation as well. The techniques of increasing the fault tolerance of FPNNs were also described. The first technique uses two operators - the identity operator and the constant operator. These operators turn selected neural resource to the transparent or constant register. These techniques exploit the FPNN robustness - that it can withstand the loss of a neural resource. The second idea is that neural resource (activator specifically) can be replaced by a constant register with the median of its values to make the FPNN computation correct at least in some cases. These techniques serve as temporal partial masking of the fault. The second technique uses the changes of FPNN resources parameters to compensate the fault impact. Both techniques are under development and they are the core of the future research.

Acknowledgment

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602, ARTEMIS JU under grant agreement no 621439 (ALMARVI) and BUT project FIT-S-14-2297.

References

[1] Girau, B.: FPNA: Concepts and Properties. In *FPGA Implementations of Neural Networks*, editted by A. R. Omondi; J. C. Rajapakse, Springer US, 2006, ISBN 978-0-387-28487-3, p. 71–123, 10.1007/0-387-28487-7-3. http://dx.doi.org/10.1007/0-387-28487-7-3

[2] Krcma, M.; Kotasek, Z. and Kastil, J.: *Mapping trained neural networks to FPNNs.* In: IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits and Systems. Belgrade: IEEE Computer Society, 2015, pp. 157–160. ISBN 978-1-4799-6779-7.

[3] Krcma, M.; Kotasek, Z. and Kastil, J.: Fault tolerant Field Programmable Neural Networks. In *Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC), 2015*, Oct 2015, s. 1–4, 10.1109/NORCHIP.2015.7364381.

[4] Munakata, T.: Neural Networks: Fundamentals and the Backpropagation Model. In *Fundamentals of the New Artificial Intelligence*, editace T. Munakata, Texts in Computer Science, Springer London, 2007, ISBN 978-1-84628-839-5, s. 7–36, 10.1007/978-1-84628-839-5–2. http://dx.doi.org/10.1007/978-1-84628-839-5--2

Related Unpublished Papers

Paper I

Fault tolerance of different Field Programmable Neural Networks types

M. Krcma and Z. Kotasek, unpublished paper

Fault tolerance of different Field Programmable Neural Networks Types

Martin Krcma, Zdenek Kotasek
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence
Bozetechova 2, 612 66 Brno, Czech Republic
Email: ikrcma@fit.vutbr.cz, kotasek@fit.vutbr.cz

***Abstract*—This paper presents deals with fault tolerance properties of Field Programmable Neural Networks (FPNNs). The paper describes the concept of FPNNs in context of our works and presents experiments we did to get insight to the concept robustness against Single Event Upset (SEU) induced errors. The experiments were based on simulating SEUs by injecting bit-flips into the FPNNs data representing the weights. Different FPNNs of three different essential types were evaluated. The experimental results were presented.**

I. Introduction

The artificial neural networks are one of the important models of soft-computing and artificial intelligence and they popularity rises. Their structure is inspired by the structure of the human brain. They try to emulate the brain's capability of learning and memorizing in order to solve various types of tasks in an intelligent way. Basically, the goal of the artificial neural networks is to learn the relation between two sets of data vectors (known data of the selected problem), to generalize the relation, to determine its features and then to use it to estimate the relation between previously unseen vectors belonging to the same problem. This capability can be used for classification tasks, for time-series and functional prediction, to control tasks, to image recognition, clustering and other tasks.

Neural networks are composed of a set of *neurons* computing the *activation function* over the *weighted sum* (their potential) of their inputs. The neurons are interconnected with the connections called *synapses*. Each synapse has its own *weight* which represents the strength of the connection similarly to the synapses in the human brain. The weights represent the neural network knowledge. The learning of the neural network is basically a process of calculating the weights.

The fault tolerance properties of artificial neural networks have been researched since the very beginning of the field's development. Many techniques were suggested to enhance the robustness of neural networks in order to harden them enough to be used in real world applications as controllers for various systems.

Many method utilize fault injection in the training phase to harden the network. The faults can be injected into neurons in the hidden layers during some of the iterations of the selected learning algorithm [1]. It is possible to harden the network against multiple faults or even against a particular set of faults selected by the network's designer [2]. These methods were proven to potentially increase the generalization ability of neural networks [5].

978-1-7281-1756-0/19/$31.00 ©2019 IEEE

Many methods are based on increasing neural networks redundancy. They are inherently highly redundant structures which grands them some level of built-in fault tolerance, it is possible however to harden the networks even more by adding redundancy to address possible faults specifically. It is possible to determine the most valuable neurons (in therm having the most measurable impact on the output data) in the network and then replicate them to harden the networks against their failure [3].

The neural networks have been implemented in various kinds of devices starting from analog computers to the most modern processors, Very large Scale Integrated circuits (VLSIs), graphical processing units and Field Programmable Gate Arrays (FPGAs). This paper deals with one of the possible implementations of artificial neural networks in FPGAs - FPNA/FPNN.

The goal of this paper is to describe the types of FPNNs and compare the fault tolerant properties of these types.

II. FPNNs

The concept of Field Programmable Neural Arrays/Networks (FPNAs/FPNNs) [6] is designed to enable a resource efficient implementation of artificial neural networks in FPGAs by adjusting the networks properties in order to make them more suitable for the FPGAs structure. The efficiency comes from the FPNN's main feature - a highly customizable structure which enables the designer to build it in a way that allows sharing the FPGA's resources between synaptic connections of the original neural network by simplifying its interconnection structure. FPNNs were used for implementing large scale spiking networks [8].

The FPNNs have not the same structures as neural networks, although they can be constructed that way. They are based on different model that can be structurally different from the original neural network. This also means that the FPNN can differ in its capabilities. In principle, the FPNNs are not straightforward implementation of neural networks but rather their approximation designed in an FPGA friendly way. Since the FPNNs can be constructed in various ways and types, the approximation accuracy can be different.

For purposes of our research we developed a new definition of an FPNN (see Definition II.1, original definition by B. Girau [6]) in order to support various FPNN types and algorithms developed to map the neural networks, enhance the approximation accuracy, detect hard faults and other tasks. We define an FPNN to be a structure composed of two types of units (together called *neural resources*). The set N contains the first type of units called *activators*. The activators directly represent the original neural network neurons. They perform the same actions the original neurons do - they gather input data into *potential* and apply an activation function to compute the activator's output. The activation function is represented by the *function operator* "f" and the *iteration operator* "i" substitutes the potential computation and is responsible for input data processing to provide the input to the function operator.

The activators are interconnected by the other type of neural resources called *links* (in the set L). The links approximate the original synaptic interconnection of the network. Unlike the synapses in neural networks link do not only transmit data between neurons (activators), they also perform the weight multiplication. This allows the weighting of the transmitted data to be computed in parallel across the FPNNs and leaves the activators the only duty to apply the iteration operator "i (usually but not necessarily a simple addition) to finish the weighted sum that forms the activator's potential.

The interconnection of the neural resources is described by an oriented graph (N, E), where E is a set of valued edges interconnecting the activators. Every edge is usually split up to a sequence of links which allows us to construct various structures. The more we split the edges into links, the more flexibility we obtain.

Definition II.1 (FPNN [6])**.** We say that structure (N, L, E, ϕ, ω) is an *FPNN* if the following statements hold true:

1) N is a set of units called *activators* that dispose of:
   a) An iterative variable t_n: $\forall n \in N : \exists t_n \in \mathbb{R}$
   b) A default value of t_n:
      $\forall n \in N : \exists o_n \in \mathbb{R}; t_{n_0} = o_n$
   c) A number of iterations: $\forall n \in N : \exists a_n \in \mathbb{N}$
   d) An iterative operator (x_n is an input data):
      $\forall n \in N : \exists i_n : \mathbb{R} \times \mathbb{R} \to \mathbb{R};$
      $t_{n_a} = i_n(t_{n_{a-1}}, x_n); a = 1..a_n$
   e) A function operator: $\forall n \in N : \exists f_n : \mathbb{R} \to \mathbb{R}$
2) L is a set of units called *links* that dispose of:
   a) A set of *link operators* $\forall l \in L : \exists A_l$:
      $A_l = \{\alpha_n(x) | \alpha_n(x) = W_n \times x; W_n \in \mathbb{R}; n = 1..c\}$
3) E is a set of valued oriented edges: $(m, n) \in E; m, n \in N)$.
   The edge value is defined: $\forall e \in E : \exists W_e \in \mathbb{R}$
4) (N, E) is an oriented graph denoting the interconnection between activators.
5) ϕ is a function $E \to L^+$, so that:
   $\forall e \in E : \phi(e) = (l_1..l_n); l_1..l_n \in L; n > 0$
6) ω is a function $E \to L^+$, so that:
   $\forall e \in E : \phi(e) = (l_1..l_n); l_1..l_n \in L; \omega(e) \subseteq \phi(e); 0 < n \leq |\phi(e)|$
7) Edge-to-operator functions $\sigma_l : E \to A_l; l \in L$:
   $\forall e \in E \wedge \forall l \in \phi(e) : \sigma_l(e) = \alpha_l^x; \alpha_l^x \in A_l$
8) Operator determining $\psi_l : E^+ \to A_l, l \in L$:
   $\forall l \in L : \psi_l(e_1..e_n) = \alpha_x \Leftrightarrow \alpha_x \in A_l \wedge l \in \omega(e_1) \wedge .. \wedge l \in \omega(e_n) \wedge \sigma_l(e_1) = .. = \sigma_l(e_n) = \alpha_x$
9) A set of input nodes exists:
   $\exists N_i = \{n \in N \mid deg^+(n) = 0\}$
   $\forall n \in N_i : i_n = \emptyset; f_n(x) = x$

A. Grid FPNN

For our research purposes we developed a special type of FPNN based on the above provided definitions. *Grid FPNN* (definition II.2) is an FPNN with an enforced limitation of the structure causing it to form a grid shape. The reason for this is to make an FPNN suitable for the implementation in FPGAs due to the similarity of the grid FPNNs structure and FPGAs interconnection bus and the sharing of resources in links.

Definition II.2 (Grid FPNN)**.** We say that FPNN is the *grid FPNN* if the the following statements hold true:

1) The activators are organized into layers.
2) The two chains of interconnected links exist in all layers composed of more than one activator. The number of links in every chain is one less than the number of activators in the layer. The output of every link is connected to the input of the nearest activator. The chains go in the opposite ways.
3) The output of every activator is connected only to a single link which provides the connection to the next layer. The output of the link is connected to the nearest activator and to the nearest links of one or both link chains in the layer (which realizes connection to all other activators).

An example of a grid FPNN can be seen in Fig. 1. In the figure, the circles represent activators, wide arrows represent links and the thin arrows represent data interconnections. The orientation of the connection arrows shows the way of the passing data. The straight wide dashed/dotted arrows represent the original neural networks synapses. The thin dashed/dotted arrows represent the chains of links approximating the particular synapses. The synapses and the particular chains are drawn with the same line and arrow styles. As the picture illustrates, there is only one link on the output of every activator which provides the connection to the following layer. It is directly connected to one successive activator in the following layer. The connection to the other activators goes through the chain of links within the whole layer. Two chains of the links are going the opposite ways. They are called *Interconnection chain* (definitions II.3-II.4). Every layer with more than one activator has an interconnection chain within.

Definition II.3. *A chain of links* is generally a sequence of *directly interconnected* links.

Definition II.4. *An interconnection chain* is a chain of links interconnecting activators within a layer. It is composed of two chains going the opposite ways. The input of every link is connected to one or two preceding links, the output is always connected to the nearest activator and to the succeeding link in the chain (if exists).

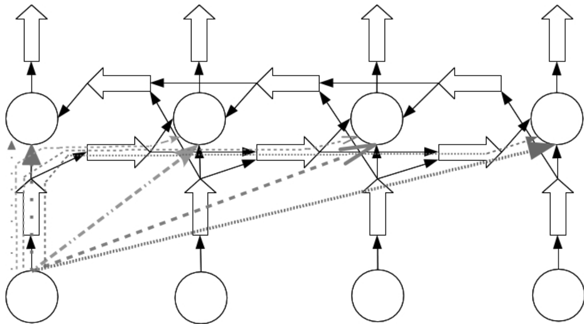

Fig. 1. Synapses (edges) approximation in a grid FPNN

Definition II.5 (Light FPNN)**.** We say that FPNN is a *light FPNN* if the following statement holds true: $\forall l \in L : |A_l| = 1$.

Definition II.6 (Full FPNN)**.** We say that FPNN is a *full FPNN* if the following statement holds true:
$\forall l \in L \wedge \forall e \in E : |A_l| = |\{e|e \in \omega^{-1}(l)\}|$.

Definition II.7 (Reduced FPNN)**.** We say that FPNN is a *reduced FPNN* if the following statements hold true:

1) The edge equivalence is defined: $\forall e_1, e_2 \in E; l \in L : e_1 \equiv_l e_2 \Leftrightarrow \phi(e_1) = l_1^1..l_x l..l_n \wedge \phi(e_2) = l_1^2..l_y l..l_m; l_x = l_y$
2) $\forall l \in L$ the size of A_l is equal to the number of the equivalence classes generated by the $\equiv_l$.

In this paper we focus on determining the FPNNs robustness against SEU introduced errors. We already presented a FPNN implementation hardened by Triple Module Redundancy on multiple levels of the concept in [10] and evaluated the resource penalty of this approach.

III. Experiments

In this work we focused on measuring effects of faulty links. Therefore we we were injecting faults into links weights to simulate possible SEU causing errors in the computation. The injected faults were generated randomly and in the range determined by the used datatype bit-length. The bit-length were chosen to be 16 bits with fixed point arithmetic. The fraction as well as the integer part have both 8 bits. It has been demonstrated that this particular setup is good enough for most neural networks computations in hardware [7].

We experimented with all three of the FPNNs types to find out the differences in their robustness as well as the general fault tolerant properties of the FPNN concept. The FPNNs were generated from neural networks trained to perform selected tasks from the [4] set of benchmarks intended to test neural networks. The input data used in the experiments were the test data from this set.

A. Injecting faults

The faults were injected by flitting a random bit in a weight. Given the different sizes and types of FPNN, we decided to consequently inject faults into all of the FPNN's weight rather than to inject a predefined number of faults. In that case, larger FPNNs would have been advantaged and their inherently higher fault tolerance (in comparison to smaller FPNNs) might have been illusionary increased. Therefore we rather injected faults into all the FPNNs weights to put them under higher and better determined stress.

B. Injecting faults

The experiment procedure for all of the tested FPNNs is as follows:

1) Test the FPNN using a test data set and save the results.
2) Save the original state of the FPNN.
3) Repeat until all weights have been tested:
    a) Select a weight that has not been tested yet.
    b) Inject a fault into the selected weight.
    c) Re-test the FPNN with the same data-set and compare the current results with the saved results of the original FPNN. Calculate the number of matching results.
    d) Restore the FPNN into its original state.

The FPNN is tested with all the test data vectors after each fault injection. Every iteration of this algorithm goes through the full testing data-set then.

IV. Recovery from faults using θ parameter modification

We have also tried to experiment with a method of recovery from an error in an affine operator by modifying the FPNN's parameters. The idea was to recover the FPNN without utilizing techniques based on redundancy, relearning, or any other complex mechanisms. Instead, the already existing resources and settings would be used. We have not expected the method to prove universally useful but rather to be another option to choose from, similarly to the method published in [9]. The method modified θ of the faulty link's succeeding activators using the fault's value. The fault values were known at the time of injection and could also be determined using the method described in [11]. The Table I shows modifiers used to change the θ values. The methods 7..12 used the same modifiers in the same order but applied them only when the faulty link had an activator among its predecessors. The idea is that the presence of the activators would indicate that the faulty link is at the beginning of some link of chains. Therefore its fault would impact all the following links in the chain, amplifying its effect. Therefore a fault in such a link would be a candidate for recovery. The methods $13-15$ added an additional modifier with a value of 0.5.

TABLE I
THE RECOVERY MODIFIERS

Method	θ modifier
1	+ fault
2	- fault
3	+ fault/10
4	- fault/10
5	$\times$ fault
6	+ fault/10

V. EXPERIMENTAL RESULTS - ROBUSTNESS

We have used six different FPNNs structures constructed to solve two benchmarks (Diabetes and Thyroid) from the Proben set. All six FPNNs were used in all three varieties of types, so the total number of FPNNs that went through experiments is eighteen. The structures of all the six basic FPNNs are listed in first columns of tables II and III. The structures are written as numbers of activators in all layers separated by the character x. The second column identify their types. The last three columns list the minimum, maximum and average percentages of matching original-faulty FPNN output vectors through all iterations of the experiment.

TABLE II
THE EXPERIMENTS RESULTS OF THE DIABETES-TASK FPNNS

	Diabetes				
Structure	**Type**	**Faults**	**Min.**[%]	**Max.**[%]	**Avg.**[%]
8x16x8x2	light	78	64.8	100	99.7
8x16x8x2	reduced	124	5.9	98.7	94.4
8x16x8x2	full	384	36.2	100	99.7
8x16x16x2	light	102	64.8	100	99.5
8x16x16x2	reduced	164	30.5	96.4	85.1
8x16x16x2	full	576	11.2	91.7	87.5
8x64x2	light	200	36.2	100	97.1
8x64x2	reduced	328	38.8	100	94.5
8x64x2	full	4160	53.4	97.4	95.6
8x32x32x2	light	198	64.8	100	99.6
8x32x32x2	reduced	324	100	100	100
8x32x32x2	full	2112	0	100	98.8

TABLE III
THE EXPERIMENTS RESULTS OF THE THYROID-TASK FPNNS

	Thyroid				
Structure	**Type**	**Faults**	**Min.**[%]	**Max.**[%]	**Avg.**[%]
21x21x3	light	86	98.6	100	99.9
21x21x3	reduced	130	95	100	99.7
21x21x3	full	882	96.3	98.8	96.3
21x63x3	light	212	93.8	98.8	96.3
21x63x3	reduced	340	86.2	90	87.5
21x63x3	full	4410	10	75	60.1

As you can see in the tables, the light FPNNs types proved to be the most robust against bit-flipping faults. This is probably due to their reduced approximation capabilities of the original neural network that that already limits the approximation precision and therefore the injected faults had less disrupting potential as opposed of the more complex types of FPNNs that hold more computing power than the light FPNNs. The high redundancy of full FPNNs on the other hand provides them with some inherent robustness makes them perform decently in the Diabetes task as opposed to reduced FPNNs which are more prone to errors as their redundancy is lower as well as their computing and approximation power.

The Thyroid task results show an interesting trend. Larger the FPNN gets, more prone to errors it seems. That goes against the idea that larger FPNNs would have more redundancy and therefore would be more robust. It is due to the structure if these particular FPNNs that introduce this trend. The FPNNs have relatively small number of layers (as opposed to the Diabetes task FPNNs) with relatively high number of activator in the input and hidden layers. These large layers mean that their interconnection chain is rather long. Any introduced error in these chains (especially at their beginnings) would go trough a number of the consequent links and get potentially worse as it participates in the following computations in the chain. The longer the interconnection chain is the higher the chance of a fault to introduce an increasingly influential error into the computation. We can see this effect even with the Diabetes FPNN with 64 activators in the hidden layer.

VI. EXPERIMENTAL RESULTS - RECOVERY

We have experimented with the recovery method applied to both Diabetes and Thyroid tasks and their particular FPNNs listed in the previous section. We have also generated faulty FPNNs using the method described in the previous section. We used all the fifteen techniques described in Section . Given the number of combinations of these settings and the quantity of resulting data, we could not include them in detail in this paper. Instead, we decided to include average rates between all the FPNNs performing the given task regarding the particular recovery method. Table IV contains the results of the Diabetes FPNNs, and Table V contains the results of the Thyroid FPNNs. The first column *Mtd.* identifies the recovery method by its number. Then each table contains three triplets of columns that list the data of the three FPNN types implementing the given task. In each triplet, the first column denoted by *F* contains the average results of the FPNNs after injecting a fault. Like in the previous experiments, the results are in terms of average correspondence between the original FPNNs' and the *Faulty* FPNNs' results. The *R* column shows the average correspondence between the original FPNNs and the *Recovered* FPNN. The last column, the *I* column, shows the *Improvement* of the recovered FPNN over the faulty FPNN; therefore, how closer to the original FPNNs the recovered FPNNs' results got.

Unfortunately, the method proved to be more disrupting than repairing. Even though there were instances of improving the results of the recovered FPNNs as compared to the faulty FPNNs, there is not a particular method that would show a general pattern of improvements. Most results show that the θ parameter modifications led to even worse results than the faulty FPNN. What these results show, however, is that FPNNs

TABLE IV
THE RECOVERY RESULTS OF THE DIABETES-TASK FPNNS

	Diabetes								
	Light			Reduced			Full		
Mtd	**F**	**R**	**I**	**F**	**R**	**I**	**F**	**R**	**I**
1	47.5	47.4	-0.1	57.9	66.8	8.9	55.7	54.7	-1.0
2	47.5	51.3	3.8	57.9	56.7	-1.2	55.7	56.1	0.4
3	47.5	47.4	-0.1	57.9	57.6	-0.3	55.7	55.5	-0.2
4	47.5	47.5	0	57.9	55.2	-2.7	55.7	49.3	-6.4
5	47.5	47.4	-0.1	57.9	57.6	-0.3	55.7	56.9	1.2
6	47.5	47.4	-0.1	57.9	57.6	-0.3	55.7	49.4	-6.3
7	47.5	47.4	-0.1	57.9	66.8	8.9	55.7	55.7	0
8	47.5	51.3	3.8	57.9	56.7	-1.2	55.7	55.8	0.1
9	47.5	47.4	-0.1	57.9	57.6	-0.3	55.7	55.7	0
10	47.5	47.5	0	57.9	55.2	-2.7	55.7	49.3	-6.4
11	47.5	47.4	-0.1	57.9	57.6	-0.3	55.7	55.7	0.0
12	47.5	47.4	-0.1	57.9	57.6	-0.3	55.7	49.3	-6.4
13	47.5	47.4	-0.1	57.9	62.2	4.3	55.7	55.7	0
14	47.5	47.5	0	57.9	56.0	-1.9	55.7	49.3	-6.4
15	47.5	47.4	-0.1	57.9	57.6	-0.3	55.7	49.3	-6.4

TABLE V
THE RECOVERY RESULTS OF THE THYROID-TASK FPNNS

	Thyroid								
	Light			Reduced			Full		
Mtd	**F**	**R**	**I**	**F**	**R**	**I**	**F**	**R**	**I**
1	74.6	74.6	0	97.6	97.6	0	71.1	66.4	-4.7
2	74.6	90.9	16.3	97.6	97.6	0	71.1	65.5	-5.6
3	74.6	74.6	0	97.6	97.6	0	71.1	70.9	-0.2
4	74.6	74.6	0	97.6	97.6	0	71.1	70.7	-0.4
5	74.6	61.4	-13.2	97.6	97.6	0	71.1	69.6	-1.5
6	74.6	74.6	0	97.6	97.6	0	71.1	70.7	-0.4
7	74.6	74.6	0	97.6	97.6	0	71.1	66.4	-4.7
8	74.6	90.9	16.3	97.6	97.6	0	71.1	65.6	-5.5
9	74.6	74.6	0	97.6	97.6	0	71.1	70.9	-0.2
10	74.6	74.6	0	97.6	97.6	0	71.1	70.7	-0.4
11	74.6	61.4	-13.2	97.6	97.6	0	71.1	70.2	-0.9
12	74.6	74.6	0	97.6	97.6	0	71.1	70.7	-0.4
13	74.6	74.6	0	97.6	97.6	0	71.1	66.4	-4.7
14	74.6	90.9	16.3	97.6	97.6	0	71.1	65.9	-5.2
15	74.6	74.6	0	97.6	97.6	0	71.1	70.7	-0.4

are particularly sensitive to the θ parameters modification, which can also be seen as errors in activators.

VII. CONCLUSION

In this paper we focused on robustness of different FPNN types against SEU introduced errors. We presented experimental results showing that light FPNNs are relatively robust against faults due to their limited computing power while full FPNN proved to be more robust in the Diabetes task. In the Thyroid task, we also demonstrated that robustness of FPNN depends on its structure. It is shown that FPNNs with large hidden layers featuring a long interconnection chain are more prone to errors due to cumulative effect of an error going trough the long chain of links. The experiments with recovery from a fault using activator's θ parameters shown the vulnerability of the FPNNs to changes in these parameeters.

In the future work we focus on more experiments with more FPNNs as well on methods to harden the FPNNs using other activators parameters modifications as well as using the weight re-computation without a need of retraining and remapping the FPNNs.

REFERENCES

[1] C. Sequin and R. Clay, "Fault tolerance in artificial neural networks," in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, Jun. 1990, 703–708 vol.1. DOI: 10.1109/IJCNN.1990.137651.

[2] Y. Tan and T. Nanya, "Fault-tolerant back-propagation model and its generalization ability," in *Neural Networks, 1993. IJCNN '93-Nagoya. Proceedings of 1993 International Joint Conference on*, vol. 3, Oct. 1993, 2516–2519 vol.3. DOI: 10.1109/IJCNN.1993.714236.

[3] C.-T. Chin, K. Mehrotra, C. Mohan, and S. Rankat, "Training techniques to obtain fault-tolerant neural networks," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, Jun. 1994, pp. 360–369. DOI: 10.1109/FTCS.1994.315624.

[4] L. P. Prechelt and F. F. Informatik, "-– a set of neural network benchmark problems and benchmarking rules," Universitat Karlsruhe; 76128 Karlsruhe, Germany, Tech. Rep., 1994.

[5] H. Elsimary, S. Mashali, and S. Shaheen, "Generalization ability of fault tolerant feedforward neural nets," in *Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century., IEEE International Conference on*, vol. 1, Oct. 1995, 30–34 vol.1. DOI: 10.1109/ICSMC.1995.537728.

[6] B. Girau, "Fpna: Concepts and properties," in *FPGA Implementations of Neural Networks*, A. R. Omondi and J. C. Rajapakse, Eds., 10.1007/0-387-28487-7-3, Springer US, 2006, pp. 63–101, ISBN: 978-0-387-28487-3. [Online]. Available: http://dx.doi.org/10.1007/0-387-28487-7-3.

[7] M. Moussa, S. Areibi, and K. Nichols, "On the arithmetic precision for implementing back-propagation networks on fpga: A case study," in *FPGA Implementations of Neural Networks*, A. R. Omondi and J. C. Rajapakse, Eds., 10.1007/0-387-28487-7$_2$, Springer US, 2006, pp. 37–61, ISBN: 978-0-387-28487-3. [Online]. Available: http://dx.doi.org/10.1007/0-387-28487-7_2.

[8] J. Harkin, F. Morgan, S. Hall, P. Dudek, T. Dowrick, and L. McDaid, "Reconfigurable platforms and the challenges for large-scale implementations of spiking neural networks," in *2008 International Conference on Field Programmable Logic and Applications*, 2008, pp. 483–486. DOI: 10.1109/FPL.2008.4629989.

[9] M. Krcma, Z. Kotasek, and J. Kastil, "Fault tolerant field programmable neural networks," in *Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC), 2015*,

Oct. 2015, pp. 1–4. DOI: 10.1109/NORCHIP.2015.7364381.

[10] M. Krcma, Z. Kotasek, and J. Lojda, "Triple modular redundancy used in field programmable neural networks," in *2017 IEEE East-West Design Test Symposium (EWDTS)*, 2017, pp. 1–6. DOI: 10.1109/EWDTS.2017.8110128.

[11] ——, "Detecting hard synapses faults in artificial neural networks," in *2019 IEEE Latin American Test Symposium (LATS)*, 2019, pp. 1–6. DOI: 10.1109/LATW.2019.8704637.

Appendices

Publications cited by other authors

- M. Krcma, Z. Kotasek and J. Lojda, „Triple modular redundancy used in field programmable neural networks,“ *2017 IEEE East-West Design & Test Symposium (EWDTS)*, 2017, pp. 1-6, doi: 10.1109/EWDTS.2017.8110128.
  - T. Fruehling et al., „Architectural Safety Perspectives Considerations Regarding the AI-based AV Domain Controller,“ *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*, 2019, pp. 1-10, doi: 10.1109/ICCVE45908.2019.8965197.
  - Sapozhnikov, V. & Sapozhnikov, Vl & Efanov, Dmitry. (2020). Signal correction for combinational automation devices on the basis of Boolean complement with control of calculations by parity. Informatics. 17. 71-85. 10.37661/1816-0301-2020-17-2-71-85.
  - Efanov, Dmitry & Sapozhnikov, V. & Sapozhnikov, Vl. (2021). Boolean-Complement Based Fault-Tolerant Electronic Device Architectures. *Automation and Remote Control.* 82. 1403-1417. 10.1134/S0005117921080075.
  - V. Sapozhnikov, V. Sapozhnikov and D. Efanov, „The Structures of the Fault-Tolerant Automation and Computing Devices Based on the Boolean Complement,“ *2021 IEEE East-West Design Test Symposium (EWDTS)*, 2021, pp. 1-10, doi: 10.1109/EWDTS52692.2021.9581037.
  - Sapozhnikov, V. & Sapozhnikov, Vl & Efanov, Dmitry. (2022). Duplication of Boolean Complements for Synthesis of Fault-Tolerant Digital Devices and Systems. *Automatic Control and Computer Sciences.* 56. 1-9. 10.3103/S0146411622010096.
- M. Krcma, Z. Kotasek and J. Lojda, „Implementation of fault tolerant techniques into FPNNs,“ *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 297-298, doi: 10.1109/FPT.2016.7929559.
  - Johnson, Anju & Liu, Junxiu & Millard, Alan & Karim, Shvan & Tyrrell, Andy & Harkin, Jim & Timmis, Jon & McDaid, Liam & Halliday, David. (2017). Homeostatic fault tolerance in spiking neural networks utilizing dynamic partial reconfiguration of FPGAs. 195-198. 10.1109/FPT.2017.8280139.

- M. Krcma, Z. Kotasek and J. Kastil, „Fault tolerant Field Programmable Neural Networks,“ *2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)*, 2015, pp. 1-4, doi: 10.1109/NORCHIP.2015.7364381.
  - I. C. Lopes, F. L. Kastensmidt and A. A. Susin, „SEU susceptibility analysis of a feedforward neural network implemented in a SRAM-based FPGA,“ *2017 18th IEEE Latin American Test Symposium (LATS)*, 2017, pp. 1-6, doi: 10.1109/LATW.2017.7906770.