Hindawi Publishing Corporation
VLSI Design
Volume 2010, Article ID 251210, 25 pages
doi:10.1155/2010/251210

*Research Article*

# Evolvable Block-Based Neural Network Design for Applications in Dynamic Environments

**Saumil G. Merchant[1] and Gregory D. Peterson[2]**

[1] *Department of Electrical and Computer Engineering, George Washington University, 20101 Academic Way, Ashburn, VA 20147-2604, USA*
[2] *Department of Electrical Engineering and Computer Science, University of Tennessee, 414 Ferris Hall, Knoxville, TN 37996-2100, USA*

Correspondence should be addressed to Saumil G. Merchant, smerchan@gwu.edu

Received 7 June 2009; Accepted 2 November 2009

Academic Editor: Ethan Farquhar



Dedicated hardware implementations of artificial neural networks promise to provide faster, lower-power operation when compared to software implementations executing on microprocessors, but rarely do these implementations have the flexibility to adapt and train online under dynamic conditions. A typical design process for artificial neural networks involves offline training using software simulations and synthesis and hardware implementation of the obtained network offline. This paper presents a design of block-based neural networks (BbNNs) on FPGAs capable of dynamic adaptation and online training. Specifically the network structure and the internal parameters, the two pieces of the multiparametric evolution of the BbNNs, can be adapted intrinsically, in-field under the control of the training algorithm. This ability enables deployment of the platform in dynamic environments, thereby significantly expanding the range of target applications, deployment lifetimes, and system reliability. The potential and functionality of the platform are demonstrated using several case studies.

## 1. Introduction

Artificial Neural Networks (ANNs) are popular among the machine intelligence community and have been widely applied to problems such as classification, prediction, and approximation. These are fully or partially interconnected networks of computational elements called artificial neurons. An artificial neuron is the basic processing unit of the ANN that computes an output function of the weighted sum of its inputs and a bias. Depending on the interconnection topology and the dataflow through the network, ANNs can be classified as feedforward or recurrent. The design process with ANNs involves a training phase during which the network structure and synaptic weights are iteratively tuned under the control of a training algorithm to identify and learn the characteristics of the training data patterns. The obtained network is then tested on previously unseen data. ANNs are effective at identifying and learning nonlinear relationships between input and output data patterns and have been successfully applied in diverse application areas such as signal processing, data mining, and finance.

Explicit computational parallelism in these networks has produced significant interest in accelerating their execution using custom neural hardware circuitry implemented on technologies such as application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs). The design process for ANNs typically involves offline training using software simulations and the obtained network is designed and deployed offline. Hence the training and design processes have to be repeated offline for every new application of ANNs. This paper is an extended version of [1] and presents an FPGA design of block-based neural networks (BbNNs) that can be adapted and trained online, on-chip under the control of an evolutionary algorithm. On-chip evolution under the control of evolutionary algorithms is called intrinsic evolution in the evolvable hardware community [2]. The capability of intrinsic evolution expands the potential applications of these networks to dynamic

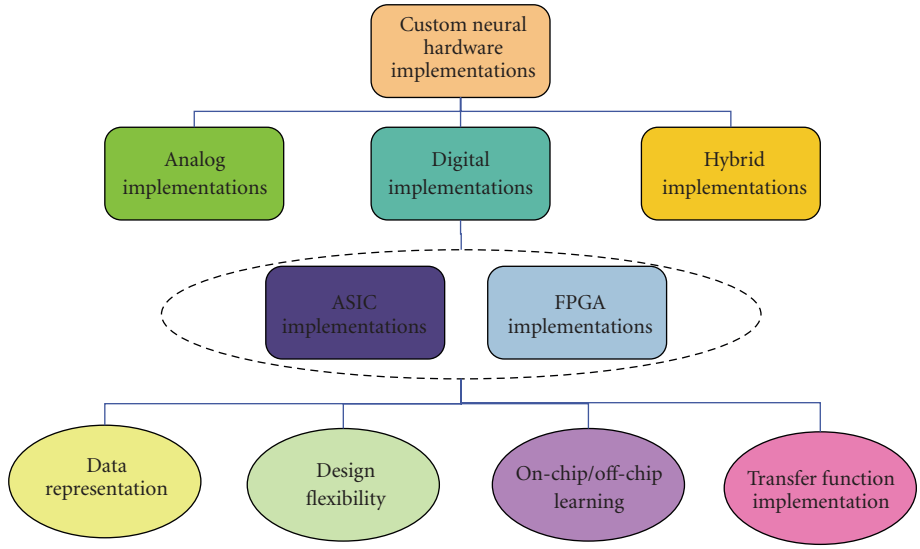



Figure 1: Neural network hardware classification.

environments and provides increased deployment lifetimes and improved system reliability. The paper also provides a detailed survey of reported ANN hardware implementations.

The rest of the paper is organized as follows. Section 2 provides a review of custom ANN implementations reported in the literature. Section 3 introduces block-based neural networks and their training procedure. Section 4 gives the details on the FPGA design of the evolvable BbNN implementation. Section 5 demonstrates the capabilities of the design using several case studies and Section 6 concludes the paper.

## 2. Review of ANN Implementations

There has been significant interest in custom ANN implementations and many have been reported in the literature over the years. Dedicated hardware units for artificial neural networks are called neurochips or neurocomputers [3]. Due to limited commercial prospects and the required development and support resources, these chips have seen little commercial viability. Also, due to the existence of wide-ranging neural network architectures and a lack of a complete and comprehensive theoretical understanding of their capabilities, most commercial neurocomputer designs are dedicated implementations of popular networks such as multilayer perceptrons (MLPs), hopfield networks, or kohonen networks targeting particular applications. Several classification and overview studies of neural hardware have appeared in the literature [3–13]. Heemskerk has a detailed review of neural hardware implementations until about 1995 [3]. He classified the neural hardware according to their implementation technologies such as the neurocomputers built using general purpose processors, digital signal processors, or custom implementations using analog, digital, or mixed-signal design. Zhu et al. have surveyed ANN FPGA implementations, classifying them based on design features such as precision and flexibility [13]. The review presented here follows and extends these surveys. Figure 1 shows the classification structure used in this review. Broadly, the implementations have been classified into digital, analog, and hybrid implementations. The digital hardware implementations have been grouped into FPGA and ASIC implementations and classified according to design features such as data representation, design flexibility, on-chip/off-chip learning, and transfer function implementation. A brief overview of analog and hybrid implementations is also offered. A detailed survey of these is beyond the scope of this work.

*2.1. Digital Neural Network Implementations.* Digital neural network implementations offer high computational precision, reliability, and programmability. They are targeted to either ASICs or FPGAs. The synaptic weights and biases for these implementations are typically stored in digital memories or registers, either on- or off-chip dictated by the design tradeoffs between the speed and the circuit size. ASIC neurochips can achieve higher processing speeds, lower power, and more density than corresponding FPGA implementations but have significantly higher design and fabrication costs. FPGAs are COTS chips and can be reconfigured and reused for different ANN applications significantly lowering implementation costs for low volume productions. The last decade has seen a lot of advancement in reconfigurable hardware technology. FPGA chips with built-in RAMs, multipliers, gigabit transceivers, on-chip embedded processors, and faster clock speeds have attracted many neural network designers. As compared to analog, digital implementations have relatively larger circuit sizes and higher power consumption.

*2.1.1. Data Representation.* Digital neural implementations represent the real-valued data such as weights and biases using fixed point, floating point, or specialized representations such as pulse stream encoding. The choice of a

particular representation is a tradeoff between arithmetic circuit size and speed, data precision, and the available dynamic range for the real values. Floating point arithmetic units are slower, larger, and more complicated than their fixed point counterparts, which are faster, smaller, and easier to design.

Generally, floating point representations of real-valued data for neural networks are found in custom ASIC implementations. Aibe et al. [14] used floating point representation for their implementation of probabilistic neural networks (PNNs). In PNNs, the estimator of the probabilistic density functions is very sensitive to the smoothing parameter (the network parameter to be adjusted during neural network learning). Hence, high accuracy is needed for the smoothing parameter, making floating point implementations more attractive. Ayela et al. demonstrated an ASIC implementation of MLPs using a floating point representation for weights and biases [15]. They also support on-chip neural network training using the backpropagation algorithm and are listed also in Section 2.1.3. Ramacher et al. present a digital neurochip called SYNAPSE-1 [16]. It consists of a 2-dimensional systolic array of neural signal processors that directly implement parts of common neuron processing functions such as matrix-vector multiplication and finding a maximum. These processors can be programmed for specific neural networks. All the real values on this chip are represented using floating point representation.

For FPGA implementations the preferred implementation choice has been fixed point due to chip capacity restrictions, but advances in FPGA densities may make floating point representations practical. Moussa et al. demonstrate implementations of MLP on FPGAs using fixed and floating point representations [17]. Their results show that the MLP implementation using fixed point representation was over 12x greater in speed, over 13x smaller in area, and achieved far greater processing density as compared to the MLP implementation using floating point representation. The works in [17–21] present precision analysis of ANN implementations and conclude that it is possible to train ANNs with fixed point weights and biases. But there is a tradeoff between minimum precision, dynamic data range, and the area required for the implementation of arithmetic units. Higher precision has fewer quantization errors but require larger arithmetic units, whereas lower precision arithmetic units are smaller, faster, and more power efficient but may have larger quantization errors that can limit the ANN's capabilities to learn and solve a problem. Tradeoff between precision, circuit area, and speed necessitates numerical analysis to determine the minimum precision for an application. Holt and Baker [19], Holt and Hwang [20], and Holt and Hwang [21] investigated the minimum precision problem on a few ANN benchmark classification problems using simulations and found 16-bit data widths with 8-bit fractional parts as sufficient for networks to learn and correctly classify the input datasets. Ros et al. demonstrate a fixed point implementation of spiking neural networks on FPGAs [22]. Pormann et al. demonstrate fixed point implementations of neural associative memories, self-organizing feature maps, and basis function networks on FPGAs [23]. Some other reported implementations that used fixed point representations can be found in [24–32].

As a third alternative many have proposed specialized data encoding techniques that can simplify arithmetic circuit designs. Marchesi et al. proposed special training algorithms for multilayer perceptrons that use weight values that are powers of two [33]. The weight restriction eliminates the need for multipliers in the design which are replaced with simple shift registers. Other approaches encode real values in stochastic bit streams and implement the multipliers in bit-serial fashion using simple logic gates instead of complex arithmetic units [34–37]. The advantage is that the product of the two stochastic bit streams can be computed using a simple bitwise "xor." But the disadvantage is that for multiplications to be correct, the bit streams should be uncorrelated. To produce these would require independent random sources which require more chip resources to implement. Also the precision limitation may affect the ANNs capability to learn and solve a problem. Murray and Smith's implementation of ANNs [38] used pulse-stream encoding for real values which was later adopted by Lysaght et al. [39] for implementations on Atmel FPGAs. Salapura used delta encoded binary sequences to represent real values and used bit stream arithmetic to calculate a large number of parallel synaptic calculations [40].

Chujo et al. have proposed an iterative calculation algorithm for the perceptron type neuron model that is based on a multidimensional, binary search algorithm [41]. Since the binary search does not require any sum of products functionality, it eliminates the need for expensive multiplier circuitry in hardware. Guccione and Gonzalez used a vector-based data parallel approach to represent real values and compute the sum of products [42]. The distributed arithmetic (DA) approach of Mintzer [43] for implementing FIR filters on FPGAs was used by Szabo et al. [44] for their implementation of neural networks. They used Canonic Signed Digit Encoding (CSD) technique to improve the hardware efficiency of the multipliers. Noory and Groza also used the DA neural network approach and targeted their design for implementation on FPGAs [45]. Pasero and Perri use LUTs to store all the possible multiplication values in an SRAM to avoid implementing costly multiplier units in FPGA hardware [46]. It requires a microcontroller to precompute all the possible product values for the fixed weight and stores it in the SRAM.

*2.1.2. Design Flexibility.* An important design choice for neural network hardware implementations is the degree of structure adaptation and synaptic parameter flexibility. An implementation with fixed network structure and weights can only be used in the recall stage of each unique application, thus necessitating a redesign for different ANN applications. For ASIC implementations this could be quite expensive due to high fabrication costs. An advantage of FPGA ANN implementations is the capability of runtime reconfiguration to retarget the same FPGA device for any number of different ANN applications, substantially reducing the implementation costs. There are several different

motivations of using FPGAs for ANN implementations such as prototyping and simulation, density enhancement, and topology adaptation. The purpose of using FPGAs for prototyping and simulation is to thoroughly test a prototype of the final design for correctness and functionality before retargeting the design to an ASIC. This approach was used in [47]. Runtime reconfigurations in FPGAs can be used for density enhancement to implement larger network sizes via time folding that a single FPGA device may not be able to hold. This increases the amount of effective functionality per unit reconfigurable circuit area of the FPGAs. Eldredge et al. used this technique to implement the backpropagation training algorithm on FPGAs [48, 49]. The algorithm was divided temporally in three different executable stages and each stage was reconfigured on the FPGA using runtime reconfiguration. More details on this and other follow-up implementations to Eldredge's technique are covered in Section 2.1.3 for on-chip learning. The runtime reconfiguration in FPGAs can also be used for topology adaptation. ANNs with different structures and synaptic parameters targeting different applications can be loaded on the FPGA via runtime reconfiguration. One of the earliest implementations of artificial neural networks on FPGAs, the Ganglion connectionist classifier, used FPGA reconfigurations to load networks with different structures for each new application of the classifier [50]. Similar approaches of using runtime reconfiguration to retarget the FPGA for different ANN applications are found in [22, 25–31, 51, 52].

Runtime reconfiguration provides the flexibility to retarget the FPGA for different ANN designs but is impractical for use with dynamic adaptations required for online training. The overheads associated with runtime reconfiguration are on the order of milliseconds. Thus the overheads of repetitive reconfigurations required in the iterative training procedures may outweigh any benefits associated with online adaptations. The design presented in this paper is an online trainable ANN implementation on FPGAs that supports dynamic structure and parameter updates without requiring any FPGA reconfiguration.

ASIC implementations of flexible neural networks that can be retargeted for different applications have been reported in literature. The Neural Network Processor (NNP) from Accurate Automation Corp. was a commercial neural hardware chip that could be adapted online [53]. It had machine instructions for various neuron functions such as multiply and accumulate or transfer function calculation. Thus the network can be programmed using the NNP assembly instructions for different neural network implementations. Mathia and Clark compared performance of single and parallel (up to 4) multiprocessor NNPs against that of the Intel Paragon Supercomputer (up to 128 parallel processor nodes). Their results show that the NNP outperformed the Intel Paragon by a factor of 4 [54].

*2.1.3. On-Chip/Off-Chip Learning.* ANN training algorithms iteratively adapt network structure and synaptic parameters based on an error function between expected and actual outputs. Hence an on-chip trainable network design should have the flexibility to dynamically adapt its structure and synaptic parameters. Most reported ANN implementations use software simulations to train the network, and the obtained network is targeted to hardware offline [22–31, 55, 56]. However, few implementations have been reported that support an on-chip training of neural networks. Eldredge et al. demonstrate an implementation of the backpropagation algorithm on FPGAs by temporally dividing the algorithm into three sequentially executable stages of the feedforward, error backpropagation, and synaptic weight update [48, 49]. The feed-forward stage feeds in the inputs to the network and propagates the internal neuronal outputs to output nodes. The backpropagation stage calculates the mean squared output errors and propagates them backward in the network in order to find synaptic weight errors for neurons in the hidden layers. The update stage adjusts the synaptic weights and biases for the neurons using the activation and error values found in the previous stages. Hadley et al. improved the approach of Eldredge by using partial reconfiguration of FPGAs instead of full-chip runtime reconfiguration [57]. Gadea et al. demonstrate a pipelined implementation of the backpropagation algorithm in which the forward and backward passes of the algorithm can be processed in parallel on different training patterns, thus increasing the throughput [58]. Ayala et al. demonstrated an ASIC implementation of MLPs with on-chip backpropagation training using floating point representation for real values and corresponding dedicated floating point hardware [15]. The backpropagation algorithm implemented is similar to that by Eldredge et al. [48, 49]. A ring of 8 floating point processing units (PUs) are used to compute the intermediate weighted sums in the forward stage and the weight correction values in the weight update stage. The size of the memories in the PUs limits the number of neurons that can be simulated per hidden layer to 200. A more recent FPGA implementation of backpropagation algorithm can be found in [59]. Witkowski et al. demonstrate an implementation of hyper basis function networks for function approximation [60]. Both learning and recall stages of the network are implemented in hardware to achieve higher performance. The GRD (Genetic Reconfiguration of DSPs) chip by Murakawa et al. can perform on-chip online evolution of neural networks using genetic algorithms [61]. The GRD chip is a building block for the configuration of a scalable neural network hardware system. Both the topology and the hidden layer node functions of the neural network mapped on the GRD chips can be reconfigured using a genetic algorithm (GA). Thus, the most desirable network topology and choice of node functions (e.g., Gaussian or sigmoid function) for a given application can be determined adaptively. The GRD chip consists of a 32-bit RISC processor and fifteen 16-bit DSPs connected in a binary-tree network. The RISC processor executes the GA code and each of the DSPs can support computations of up to 84 neurons. Thus each GRD chip can support 1260 neurons. Multiple GRD chips can be connected for a scalable neural architecture. Two commercially available neurochips from the early 1990s, the CNAPS [62] and MY-NEUPOWER [63], support on-chip training. CNAPS was an SIMD array

of 64 processing elements per chip that are comparable to low precision DSPs and was marketed commercially by Adaptive solutions. The complete CNAPS system consisted of a CNAPS server that connected to a host workstation and Codenet software development tools. It supported Kohonen LVQ (linear vector quantization) and backpropagation networks. MY-NEUPOWER supported various learning algorithms such as backpropagation, Hopfield, and LVQ and contained 512 physical neurons. The chip performed as a neural computational engine for software package is called NEUROLIVE [63].

The following references discuss analog and hybrid implementations that support on-chip training. Zheng et al. have demonstrated a digital implementation of backpropagation learning algorithm along with an analog transconductance-model neural network [64]. A digitally controlled synapse circuit and an adaptation rule circuit with an R-2R ladder network, a simple control logic circuit, and an UP/DOWN counter are implemented to realize a modified technique for the backpropagation algorithm. Linares-Barranco et al. also show an on-chip trainable implementation of an analog transconductance-model neural network [65]. Field Programmable Neural Arrays (FPNAs), an analog neural equivalent of FPGAs, are a mesh of analog neural models interconnected via a configurable interconnect network [66–70]. Thus, different neural network structures can be created dynamically, enabling on-chip training. Intel's ETANN (Electronically Trainable Analog Neural Network) [71] and the Mod2 Neurocomputer [72] are other examples of on-chip trainable analog neural network implementations. Schmitz et al. use the embedded processor on the FPGA to implement genetic algorithm operators like selection, crossover, and mutation [73]. This FPGA is closely coupled as a coprocessor to a reconfigurable analog ANN ASIC chip on the same PCB. A host processor initializes this PCB and oversees the genetic evolution process.

*2.1.4. Activation Function Implementation.* Activation functions, or transfer functions, used in ANNs are typically nonlinear, monotonically increasing sigmoid functions. Examples include hyperbolic tangent and logistic sigmoid functions. Digital ANN implementations use piecewise linear approximations of these to implement in hardware either as a direct circuit implementation or as a look-up table. Omondi et al. show an implementation of piecewise linear approximation of activation functions using the CORDIC algorithm on FPGAs [74]. Krips et al. show an implementation of piecewise linear approximation of activation functions precomputed and stored in LUTs [30]. Direct circuit implementation of the activation function requires a redesign of hardware logic for every application that uses a different activation function. In such scenarios the LUT approach maybe more flexible as the function values can be precomputed and loaded in the LUT offline. But the LUTs may occupy significant higher chip area as compared to direct implementations. Each extra bit in the data width more than doubles the size of the LUT. These tradeoffs are further discussed in Section 4.

*2.2. Analog Neural Hardware Implementations.* Analog artificial neurons are more closely related to their biological counterparts. Many characteristics of analog electronics can be helpful for neural network implementations. Most analog neuron implementations use operational amplifiers to directly perform neuron-like computations, such as integration and sigmoid transfer functions. These can be modeled using physical processes such as summing of currents or charges. Also, the interface to the environment may be easier as no analog-to-digital and digital-to-analog conversions are required. Some of the earlier analog implementations used resistors for representing free network parameters such as synaptic weights [75]. These implementations using fixed weights are not adaptable and hence can only be used in the recall phase. Adaptable analog synaptic weight techniques represent weights using variable conductance [76, 77], voltage levels between floating gate CMOS transistors [71, 78–80], capacitive charges [81, 82], or using charged coupled devices [83, 84]. Some implementations use digital memories for more permanent weight storage [85]. The works in [64–71, 73] are some other analog implementations previously discussed in Section 2.1.3 above. Although there are many advantages of implementing analog neural networks as discussed above, the disadvantage is that the analog chips are susceptible to noise and process parameter variations, and hence need a very careful design.

*2.3. Hybrid Neural Hardware Implementations.* Hybrid implementations combine analog, digital, and other strategies such as optical communication links with mixed mode designs in an attempt to get the best that each can offer. Typically hybrid implementations use analog neurons taking advantage of their smaller size and lower power consumption and use digital memories for permanent weight storage [85, 86]. The mixed-signal design of the analog neurons with the digital memories on the same die introduces a lot of noise problems and requires isolation of the sensitive analog parts from the noisy digital parts using techniques such as guard rings. Sackinger et al. demonstrate a high-speed character recognition application on the ANNA (Analog Neural Network Arithmetic and logic unit) chip [87]. The ANNA chip can be used for a wide variety of neural network architectures but is optimized for locally connected weight-sharing networks and time-delay neural networks (TDNNs). Zatorre-Navarro et al. demonstrate mixed mode neuron architecture for sensor conditioning [88]. It uses an adaptive processor that consists of a mixed four-quadrant multiplier and a current conveyor that performs the nonlinearity. Synaptic weights are stored using digital registers and the training is performed off-chip.

Due to the large number of interconnections, routing quickly becomes a bottleneck in digital ASIC implementations. Some researchers have proposed hybrid designs using optical communication channels. Maier et al. [89] show a hybrid digital-optical implementation that performs neural computations electronically, but the communication links between neural layers use an optical interconnect system. They demonstrate a magnitude of performance

improvement as compared to a purely digital approach. But on the flip side it increases hardware costs and complexity for converting signals between the electronic and the optical systems. Craven et al. [90] have proposed using frequency multiplexed communication channels to overcome the communication bottleneck in fully connected neural networks.

*2.4. Summary.* Custom neural network hardware implementations can best exploit the inherent parallelism in computations observed in artificial neural networks. Many implementations have relied on offline training of neural networks using software simulations. The trained neural network is then implemented in hardware. Although these implementations have good recall speedups, they are not directly comparable to the implementation reported here which supports on-chip training of neural networks. On-chip trainable neural hardware implementations have also been reported in literature. Most of the reported ones are custom ASIC implementations such as the GRD chip by Murakawa et al. [61], on-chip backpropagation implementation of Ayala et al. [15], CNAPS by Hammerstrom [62], MY-NEUPOWER by Sato et al. [63], and FPNA by Farquhar, et al. [66]. FPGA-based implementations of on-chip training algorithms have also been reported such as the backpropagation algorithm implementations in [48, 49, 57, 58]. An online trainable implementation of hyperbasis function networks has been reported in [60]. The implementation presented here differs from the reported ones in one or more of the following: (i) the artificial neural network supported (block-based neural networks in our case), (ii) the training algorithm, and (iii) the implementation platform. The design presented in this paper supports on-chip training without reliance on FPGA reconfigurations, unlike some of the approaches listed above. It uses genetic algorithms to train the BbNNs. The genetic operators such as selection, crossover, and mutation are implemented on the embedded processor PPC 405 on the FPGA die, similar to the approach of Schmitz et al. [73]. But unlike their approach the neural network designed is a digital implementation in the configurable logic portion of the same FPGA chip.

## 3. Block-Based Neural Networks

A block-based neural network (BbNN) is a network of neuron blocks interconnected in the form of a grid as shown in Figure 2 [91]. Neuron blocks are the information processing elements of the network and can have one of four possible internal configurations depending on the number of inputs and outputs: (i) 1-input, 3-output (1/3), (ii) 2-input, 2-output (2/2) (left side output), (iii) 2-input, 2-output (2/2) (right side output), and (iv) 3-input, 1-output (3/1). These are shown in Figure 3.

Outputs of the neuron block are a function of the summation of weighted inputs and a bias as shown in (1) below. The internal block configurations and the dataflow through the network are determined by the network structure. Figure 4 shows three unique $2 \times 2$ BbNN structures:

$$y_k = g\left(b_k + \sum_{j=1}^{J} w_{jk} x_j\right), \quad k = 1, 2, \ldots, K, \tag{1}$$

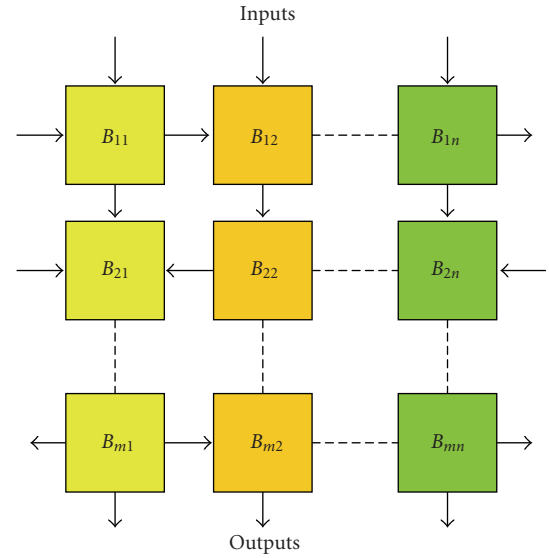



Figure 2: Block-based Neural Network topology.

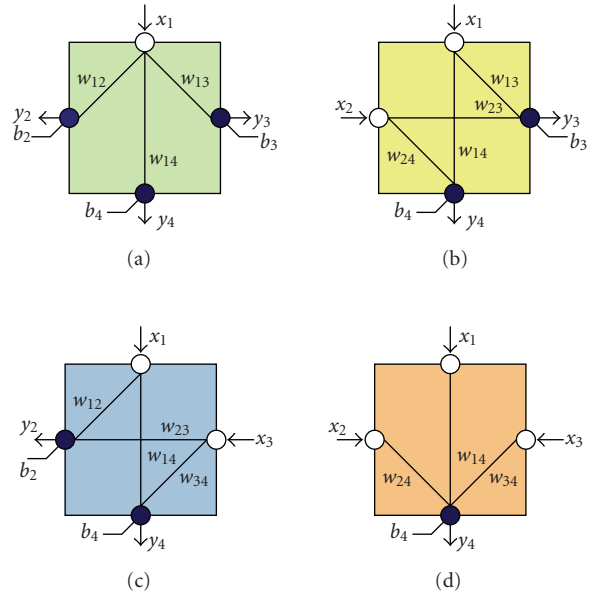



Figure 3: Four different internal configurations of a basic neuron block: (a) 1/3, (b) 2/2 (left), (c) 2/2 (right), and (d) 3/1 configurations.

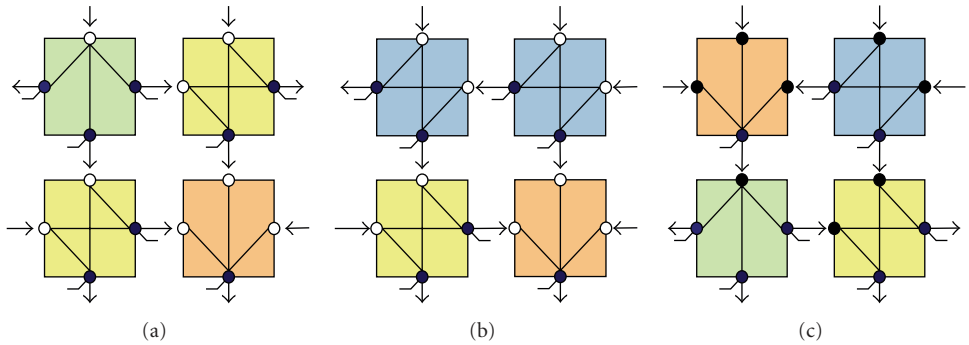



Figure 4: Three different $2 \times 2$ BbNN network structures.

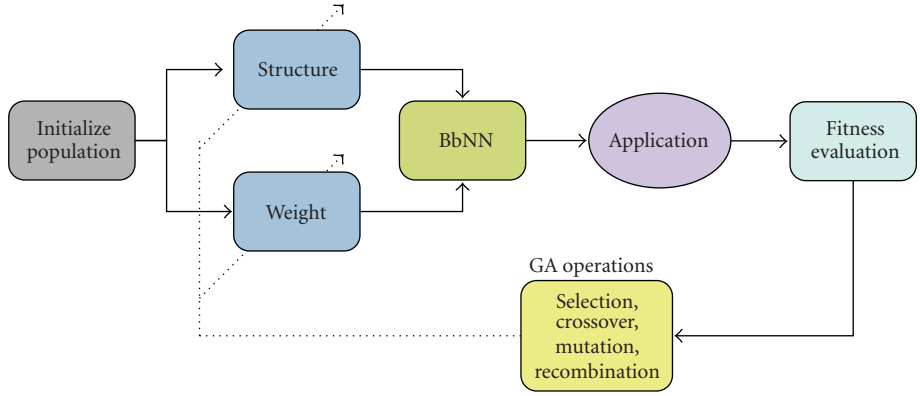



Figure 5: Flowchart of the genetic evolution process.

where,

$y_k$ : is the $k$th output signal of the neuron block,

$x_j$ : is the $j$th input signal of the neuron block,

$w_{jk}$ : is the synaptic weight connection between $j$th input node and $k$th output node,

$b_k$ : is the bias at $k$th output node,

$J, K$ : are the number of input and output nodes respectively of a neuron block,

$g(\bullet)$ : is the activation function.

Moon and Kong [91] have evaluated and compared BbNN and MLP characteristics. They state that any synaptic weight in an MLP network can be represented by a combination of more than one weight of the BbNN. Thus, a BbNN of $m \times n$ can represent the equivalent structure of the MLP network MLP$(n, m-1)$, where $n$ denotes the maximum number of neurons per layer and $m-1$ represents the total number of hidden layers in an MLP network.

BbNN training is a multiparametric optimization problem involving simultaneous structure and weight optimizations. Due to multimodal and nondifferentiable search space, global search techniques such as genetic algorithms are preferred over local search techniques to explore suitable solutions. Genetic algorithms (GAs) are evolutionary algorithms inspired from the Darwinian evolutionary model where in a population of candidate solutions (individuals or phenotypes) of a problem, encoded in abstract representations (called chromosomes or the genotypes), are evolved over multiple generations towards better solutions. The evolution process involves applying various genetic operators such as selection, crossover, mutation, and recombination to the chromosomes to generate successive populations with selection pressure against the least fit individuals. Figure 5 shows a flowchart of the genetic evolution process. The network structure and weights of the BbNN are encoded as chromosomes as shown in Figure 6. Detailed information on the BbNN GA evolution process can be found in [92–94]. BbNNs have been applied to navigation [91], classification [1, 92–94], and prediction [95] problems in the past.

## 4. BbNN FPGA Implementation

Many FPGA ANN implementations are static implementations, targeted and configured offline for individual applications. The main design objective of this implementation is enabling intrinsic adaptation of network structure and internal parameters such that the network can be trained online without relying on runtime FPGA reconfigurations. Previous versions of this implementation were reported in [1, 96]. The target system environment for the implementation is an embedded computing system. As a result various design choices were made to optimize area and power constraints. It was assumed that the hardware resources available for

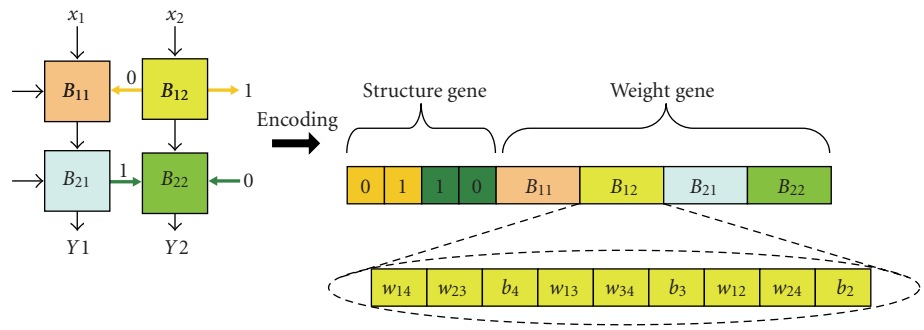



Figure 6: BbNN encoding.

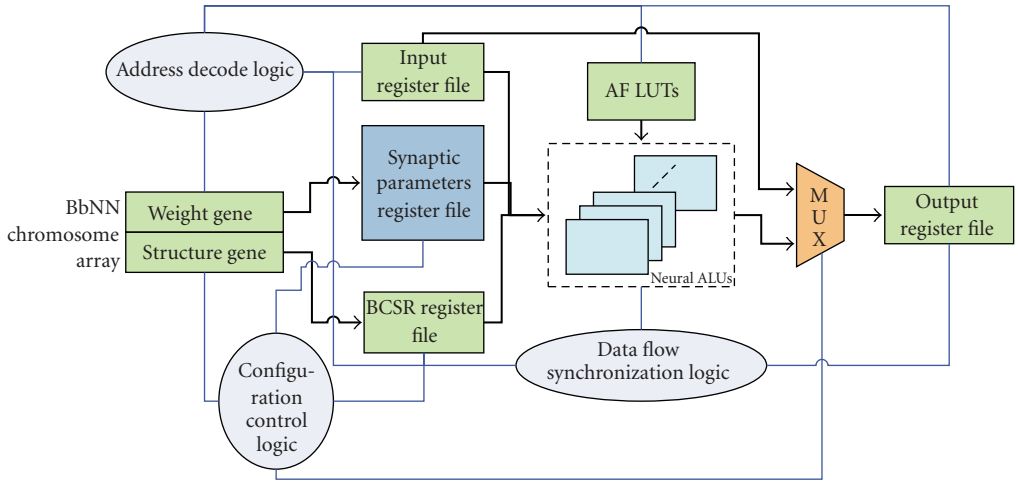



Figure 7: BbNN logic diagram.

this design in the target environment will be FPGA(s) and supporting memory and control circuits. Hence the design was prototyped on Xilinx Virtex-II Pro (XC2VP30) FPGA development boards [97]. The design was tested on two different prototyping boards, a stand-alone Digilent Inc. XUP development board [98] and an Amirix AP130 FPGA board [99]. The targeted FPGA has two on-chip PowerPC 405 embedded processor cores, 30,816 logic cells, 136 built-in $18 \times 18$ multipliers, and 2448 KBits (306 KBytes) of on-chip block RAM. The following sections describe the neuron block and network design in detail. Figure 7 shows the logic diagram of the design.

*4.1. Data Representation and Precision.* The ability to use variable bit-widths for arithmetic computations gives FPGAs significant performance and resource advantages over competing computational technologies such as microprocessors and GPUs. Numerical accuracy, performance, and resource utilization are inextricably linked, and the ability to exploit this relationship is a key advantage of FPGAs. Higher precision comes at the cost of lower performance, higher resource utilization, and increased power consumption. But at the same time, lower precision may increase the round-off errors adversely impacting circuit functionality.

The inputs, outputs, and internal parameters such as synaptic weights and biases in BbNN are all real valued. These can be represented either as floating point or fixed point numbers. Floating point representations often have a significantly wider dynamic range and higher precision as compared to fixed point representations. However, floating point arithmetic circuits are often more complicated, have a larger footprint in silicon, and are significantly slower compared to their fixed point counterparts. On the other hand fixed point representations have higher round-off errors when operating on data with large dynamic range. Although escalating FPGA device capacities have made floating point arithmetic circuits feasible for many applications, their use in our application will severely restrict the size of the network (i.e., the number of neuron blocks per network) that can be implemented on a single FPGA chip. Thus our implementation uses fixed point arithmetic for representing real-valued data. Also, [19–21] investigated the minimum fixed point precision needed for various benchmark classification problems on artificial neural networks and found 16 bits of precision as adequate for reliable operation. Hence all real valued data are represented as 16-bit fixed point numbers in our implementation.

*4.2. Activation Function Implementation.* Activation functions used in ANNs are typically nonlinear, monotonically increasing sigmoid functions. A custom circuit implementation of these functions may be area efficient as compared

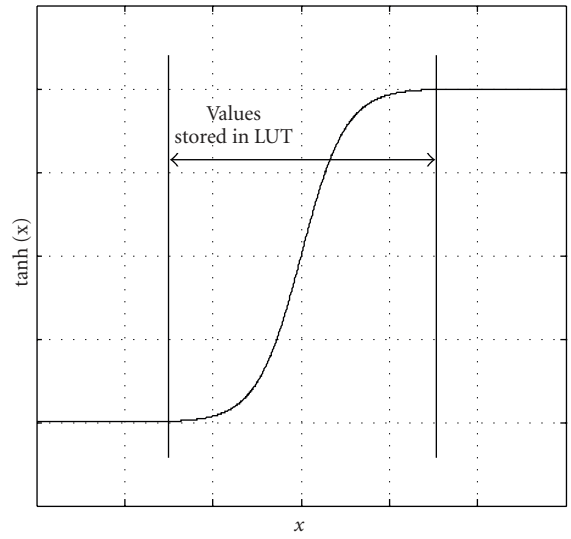



Figure 8: Illustrating activation function implementation in LUT.

to piecewise linearly approximated values stored in lookup tables (LUTs) but is inflexible and involves complicated circuitry. In our design we have chosen to use the internal FPGA BRAMs to implement LUTs which can be preloaded and reloaded as necessary with activation function values. The size of the lookup table required is directly associated with the data widths used. A 16-bit fixed point representation requires an LUT that is 16 bits wide and $2^{16}$ deep. This requires a total of 128 KBytes per LUT. Even though it may be simpler to use separate LUTs for each neuron block in the network such that each block can have parallel access to the LUTs, it is inefficient utilization of on-chip resources and the circuit will have longer setup times. For our design we have chosen to share an LUT across each column of neuron blocks in a network. This ensures that every neuron block that is actively computing at a given time will have independent access to the LUTs. This is guaranteed as no two blocks in a column can execute simultaneously for the case of feedforward networks implemented here. Sharing one LUT per column may also set an upper bound to the total number of columns implemented per FPGA due to limited BRAM resources per chip. But due to various optimizations described next the main bottleneck for our design is the number of logic resources on the FPGA used for neuron block designs and not the available internal memory. The 128 KB LUT is still large enough to restrict the number of columns that can be implemented per FPGA. For example, on the VirtexIIPro (V2P30) FPGA from Xilinx we would be restricted to only two columns. This severely restricts our ability to implement any interesting applications on the BbNN. Hence to further optimize the size of the LUT we restricted it to 16 bits $\times$ $2^{12}$ (i.e., 8 KB per LUT). Since the activation functions monotonically increase during only a small range of input data and saturate outside that window (e.g., hyperbolic tangent or logistic sigmoid functions) this optimization, in effect, stores only the transient portion of the activation function as illustrated in Figure 8.

*4.3. Neuron Block Design.* Kothandaraman designed a core library of neuron blocks with different internal block configurations for implementation on FPGAs [25]. A network can be stitched together with these cores using an HDL, but with fixed network structure and internal parameters. Thus a new network needs to be designed for each unique application. The objective to design an online, evolvable network necessitates dynamically adaptable network design that can change structure and internal parameters on-the-fly with little overhead.

A simplistic design can combine all cores in the library into a larger neuron block and use a multiplexor to select individual cores with the correct internal configuration. But the area and power overheads of such an approach render it impractical. Instead, a smarter block design is presented here that can emulate all internal block configurations dynamically as required and is less than a third the size of the simplistic larger block. For obvious reasons the block design is called the Smart Block-based Neuron (SBbN). An internal configuration register within each SBbN called the Block Control and Status Register (BCSR) regulates the configuration settings for the block. BCSR is a 16-bit register and is part of the configuration control logic section of the neuron block that defines the state and configuration mode of the block. All the bits of this register except 8 through 11 are read-only and the register can be memory or I/O mapped to the host systems address space for read and write operations. Figure 9 shows all the bit fields of the BCSR. States of BCSR bits 4 through 7 determine the current internal configuration of the neuron block. Setting bit 3 deactivates the block which then acts as a bypass from inputs to corresponding outputs. Figure 10 shows the BCSR settings and corresponding internal block configurations.

*4.4. Configuration Control Logic.* Configuration control logic uses the structure gene within the BbNN chromosome array as its input and sets the BCSR configuration bits of all the neuron blocks in the network. The translation process from the structure gene array into internal configuration bits of the neuron blocks is illustrated in Figure 11 for a $10 \times 4$ network. The translation logic extracts the structure gene from the chromosome array, divides it by the number of rows, and extracts the corresponding bits for each column from all the rows. The extracted column bits are divided across corresponding neuron blocks in the column and written to the internal BCSR of the corresponding neuron block. The neuron blocks not used are deactivated. Since the configuration control logic is combinational logic, the network is reconfigured to the correct network structure based on the structure gene loaded in the BbNN chromosome array immediately after a small delay.

*4.5. Address Decode Logic.* Address decoder provides a memory-mapped interface for the read/write registers and memories in the network design. It decodes address, data, and control signals for the input and output register files, the BbNN chromosome array, the BCSRs within each neuron block, and the activation function LUTs.

BCSR-block control and status register

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | AF1 | AF2 | AF3 | AF4 |

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| ND1 | ND2 | ND3 | ND4 | DACT | 0 | CB | FM |

(a)

| Bits | | Description |
|---|---|---|
| 15-12 | | *reserved* |
| 11 | AF1 | Node 1 Activation function enable ("0"-Purelin / "1"-LUT AF) |
| 10 | AF2 | Node 2 Activation function enable ("0"-Purelin /"1"-LUT AF) |
| 9 | AF3 | Node 3 Activation function enable ("0"-Purelin /"1"-LUT AF) |
| 8 | AF4 | Node 4 Activation function enable ("0"-Purelin /"1"-LUT AF) |
| 7 | ND1 | Node 1 direction (hardcoded as "0"-Input) |
| 6 | ND2 | Node 2 direction ("0"-Input /"1"-Output) |
| 5 | ND3 | Node 3 direction ("0"-Input /"1"-Output) |
| 4 | ND4 | Node 4 direction (hardcoded as "1"-Output) |
| 3 | DACT | Deactivate block |
| 2 | | *reserved* |
| 1 | CB | Configuration busy signal |
| 0 | FM | Neuron fire mode signal |

(b)

Figure 9: Block control and status register (BCSR).

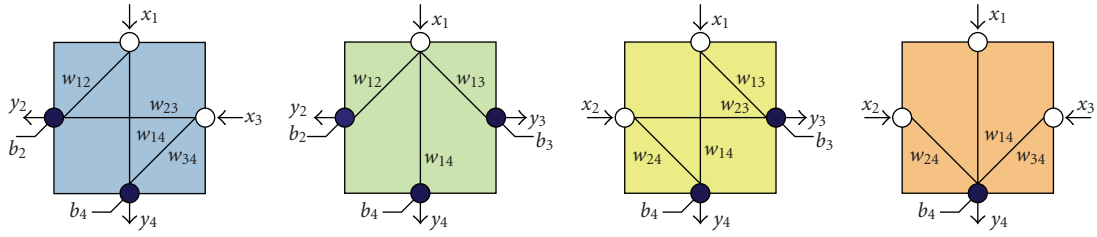



Figure 10: SBbN emulation of the internal block configurations based on BCSR settings.

*4.6. Dataflow Synchronization Logic.* To enable network scalability across multiple FPGAs the data synchronization between neuron blocks is asynchronous. Synchronization is achieved using generation and consumption of tokens as explained next. The logic associates a token with each input and output registers of every neuron block in the network. Each neuron block can only compute outputs (i.e., fire) when all of its input tokens are valid. On each firing the neuron block consumes all of its input tokens and generates output tokens. The generated output tokens in turn validate the corresponding input tokens of the neuron blocks next in the dataflow. This is illustrated in Figure 12. Black dots (•) in the figure represent valid tokens. Asynchronous communication mechanism between neuron blocks facilitates implementing larger network sizes either by scaling networks across multiple FPGAs or by folding the network in a time multiplexed fashion within a single FPGA.

*4.7. Intrinsic BbNN Evolution.* The design features explained above enable dynamic adaptations to network structure and internal synaptic parameters. Hence the network can be trained online, intrinsically under the control of the evolutionary training algorithm described in Section 3 above. There are several options for implementing the training algorithm based on the end system solution and other system design constraints. Examining the execution profiles of the serial genetic evolution code, the most computationally intensive section is found to be the population fitness evaluation function. Fitness evaluations involve applying

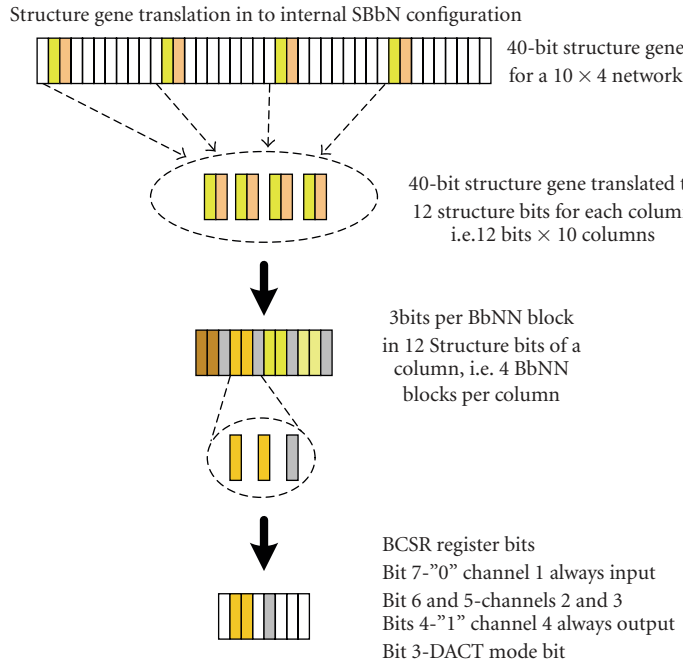



Figure 11: Gene translation process within the configuration control logic.

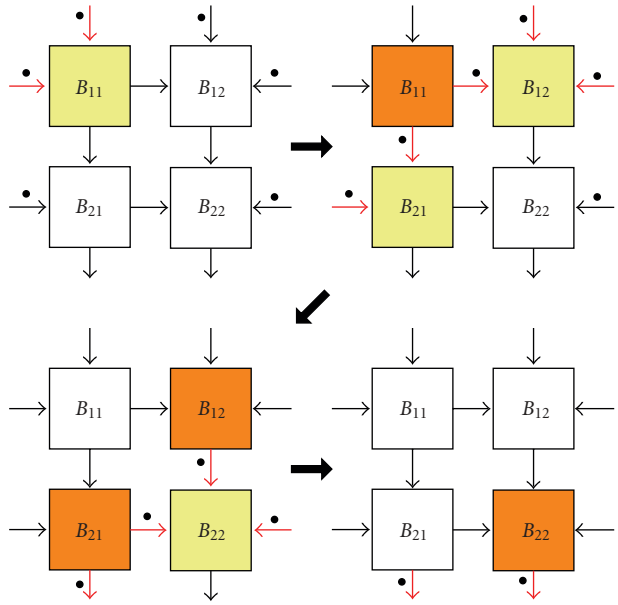



Figure 12: Dataflow synchronization logic.

the training patterns to the network and examining the measured outputs against expected values to determine the network fitness. This is performed for each network in the population to determine the average and maximum population fitness values. Genetic operations such as selection, crossover, and mutation are relatively less computationally intensive. Hence an obvious software-hardware partitioning is to perform genetic operations in software running on the host microprocessor and fitness evaluations in hardware executing in FPGAs. This has the benefit of dedicating all the available configurable logic space in the FPGAs to implement the neural network, facilitating implementation of larger network sizes.

Escalating FPGA logic densities has enabled building a programmable system-on-chip (PSoC) with soft or hard-core processors, memory, bus system, IO, and other custom cores on a single FPGA device. The Xilinx Virtex-II Pro FPGA used for prototyping our implementation has on-chip, hard-core PPC405 embedded processors which are used to design our PSoC prototyping platform. The platform is

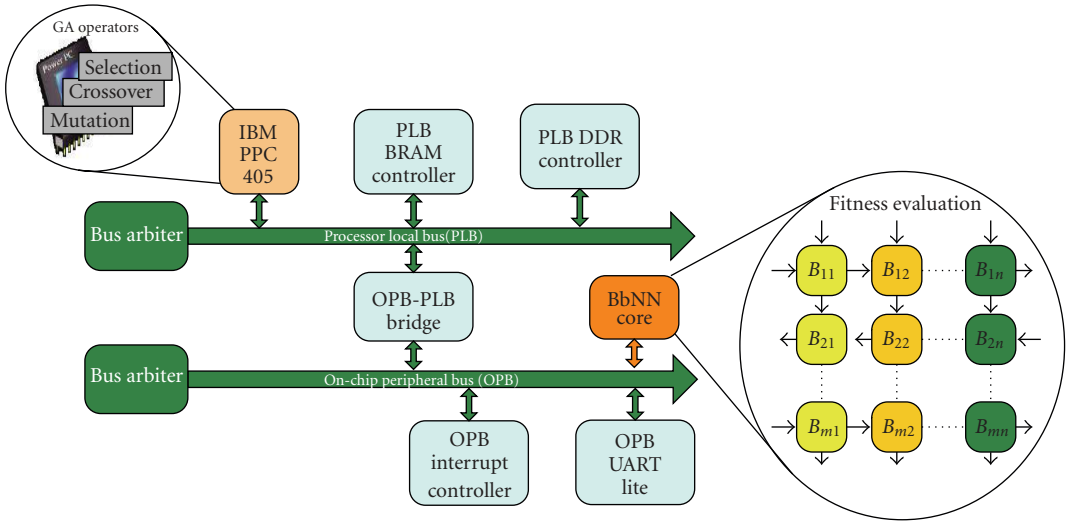



Figure 13: PSoC platform used for prototyping BbNN implementation.

Table 1: Peak and relative computational capacities and capacity per mW of commercial embedded processors. Relative values are normalized to PPC405 numbers.

| Processor | Organization | Cycle freq | Power | MOPS | Relative MOPS | MOPS/mW | Relative MOPS/mW |
|---|---|---|---|---|---|---|---|
| MIPS 24Kc | $1 \times 32$ | 261 MHz | 363 mW | 87 | 0.65 | 0.24 | 0.14 |
| MIPS 4KE | $1 \times 32$ | 233 MHz | 58 mW | 78 | 0.59 | 1.33 | 0.76 |
| ARM 1026EJ-S | $1 \times 32$ | 266 MHz | 279 mW | 89 | 0.67 | 0.32 | 0.18 |
| ARM 11MP | $1 \times 32$ | 320 MHz | 74 mW | 107 | 0.80 | 1.45 | 0.83 |
| ARM 720T | $1 \times 32$ | 100 MHz | 20 mW | 33 | 0.25 | 1.67 | 0.95 |
| PPC 405 | $1 \times 32$ | 400 MHz | 76 mW | 133 | 1.00 | 1.75 | 1.00 |
| PPC 440 | $1 \times 32$ | 533 MHz | 800 mW | 178 | 1.34 | 0.22 | 0.13 |
| PPC 750FX | $2 \times 32$ | 533 MHz | 6.75 W | 355 | 2.67 | 0.05 | 0.03 |
| PPC 970FX | $2 \times 64$ | 1 GHz | 11 W | 667 | 5.02 | 0.06 | 0.03 |

designed using Xilinx EDK and ISE tools. A block diagram for the designed platform is shown in Figure 13. The BbNN core is memory-mapped to the PPC405's address space via the on-chip peripheral bus (OPB). The genetic operations such as selection, crossover, mutation, and recombination are implemented in software executing on the PPC405 processor and the population fitness evaluation is implemented on the BbNN core as shown in Figure 13. On-board DDR is used as data memory to quickly access and store the chromosome populations and the training vectors for the evolution process. A fixed point version of the BbNN GA evolution algorithm is used due to the limited computational capacity of the on-chip PowerPC processor. The platform offers a very compact solution that is deployable and adaptable in field for embedded applications with area and power constraints.

In the case of target environments with less stringent area and power constraints, other higher-capacity embedded solutions such as single-board computers with FPGA accelerators can be used. The GA operators can thus be implemented on the on-board processor and the fitness evaluation can be performed in the FPGA using the BbNN core. Table 1 surveys computational capacities and capacity per mW of some commercial embedded processors. Also shown are relative values normalized to PPC405 numbers. The capacities are calculated to implement a single 2/2 BbNN block computation, that is 6 integer arithmetic operations. The calculations assume CPI of 1.0 and ignore instruction fetch and data transfer overheads. The Watt ratings are as published in the processor datasheets and do not include power consumption of other associated hardware resources. Based on these calculations, PPC405 offers the best computational capacity per mW out of the processors surveyed. The PPC970FX has 5 times the computational capacity of PPC405, but significantly higher power consumption. Although the numbers approximate capacities for neuron block computations, similar values for GA operators can be extrapolated from these numbers. For our assumed target

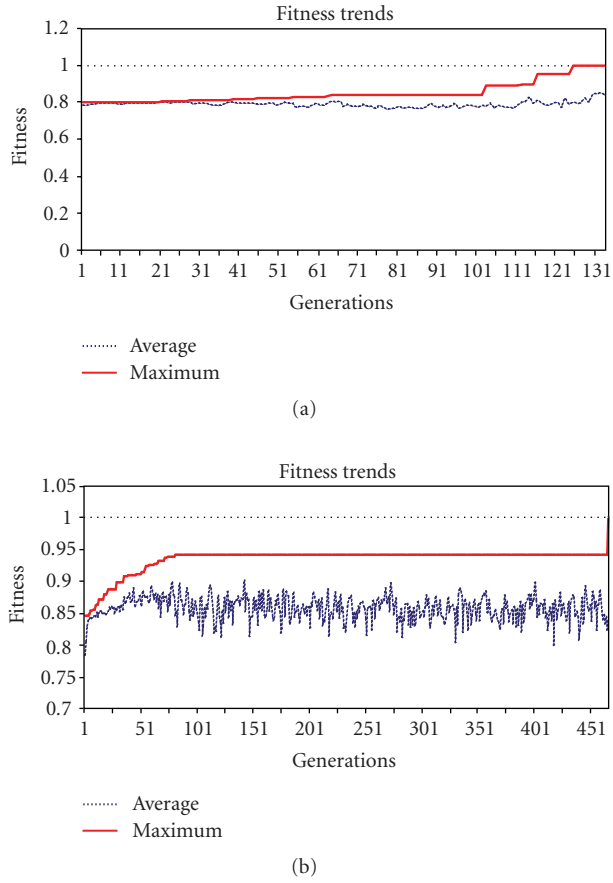



FIGURE 14: Fitness evolution trends for (a) 3-bit and (b) 4-bit parity examples.

system constraints, power efficiency is critical. Thus, the selection of the on-chip PPC405 to execute the GA operators in our prototype implementation is a sound design decision.

*4.8. Performance and Resource Utilization.* The postsynthesis timing analysis for the design reports a clock frequency of 245 MHz on the Xilinx Virtex-II Pro FPGA (XC2VP30). The designed block takes 10 clock cycles to compute outputs for six synaptic connections per block. Thus, each block has a computational capacity of 147 MCPS per block. Computational capacity is the measure of throughput defined as computational work per unit time. Hence for an artificial neural network it is determined by the number of synaptic connections processed per second (unit CPS). The computational capacity of the network is determined by the number of concurrent block executions, which in turn is dependent on the network structure. At peak computational capacity one block from each network column computes concurrently. Hence an $m \times n$ BbNN has a peak computational capacity of $147n$ MCPS. But the clock frequency on the actual implementation is limited by the on-chip peripheral bus (OPB) clock frequency which is set at 100 MHz for the prototyping platform. Thus, the peak computational capacity is limited to 60 MCPS per block or $60n$ MCPS for a $m \times n$ network size. The minimal platform (as shown in Figure 13) excluding the BbNN occupies about 13% of the Xilinx Virtex-II Pro FPGA (XC2VP30) resources.

Table 2 shows the postsynthesis device utilization summaries for various network sizes. According to the utilization summaries we can fit around 20 neuron blocks on a single FPGA chip along with the rest of the platform. Table 3 shows the postsynthesis device utilization results for a larger device (XC2VP70) from the same Xilinx Virtex-II Pro family of FPGA devices family. This device can hold around 48 neuron blocks.

## 5. Case Studies

Results from three case studies are presented. The case studies on $n$-bit parity classifier and Iris data classification demonstrate the functionality of the design. The case study on adaptive forward prediction demonstrates the benefit of online evolution capability.

*5.1. n-bit Parity Classification.* A parity classifier can be used to determine the value of the parity bit to get even or odd number of 1's in an "$n$" bit stream. The technique is widely used for error detection and correction with applications in communication and data storage. Results presented here show the outcome of BbNN training to determine the value of the parity bit for 3-bit and 4-bit data streams. A population size of 30 chromosomes is used for genetic evolution with crossover and mutation probabilities set at 0.7 and 0.1, respectively. The evolutionary algorithm uses tournament selection to choose parent chromosomes for crossover and elitism for recombination operations. A logistic sigmoid function was used as the activation function for the neuron block outputs. Figure 14 shows the average and maximum fitness curves for the 3-bit and 4-bit parity examples. The target fitness of 1.0 is reached after 132 generations in the case of the 3-bit parity problem and 465 generations for the 4-bit parity example. Figure 15 shows the evolution trends for the top five structures. Each color indicates a unique structure and the $y$-axis values determine the number of chromosomes per generation. Figure 16 shows the evolved networks for the 3-bit and the 4-bit parity examples. The average time per generation of the evolution with the PPC405 processor computing at 300 MHz was found to be 11 microseconds.

*5.2. Iris Data Classification.* This case study uses a well-known dataset in the machine learning community originally compiled by R. A Fisher [100]. The dataset has 150 samples of three classes of Iris plants, *Iris Setosa*, *Iris Versicolour,* and *Iris Virginica* with 50 samples per class. The dataset attributes are sepal length, sepal width, petal length, and petal width for the three classes of the Iris plants. The Iris Setosa class is linearly separable from the other two classes, Iris Versicolour and Iris Virginica. But the latter is not linearly separable from each other, which makes this an interesting problem to test neural classifiers. A BbNN was used to learn and correctly classify this dataset. The classification result is shown in Figure 17. The results show less than a 1.5% misclassification rate. A population size

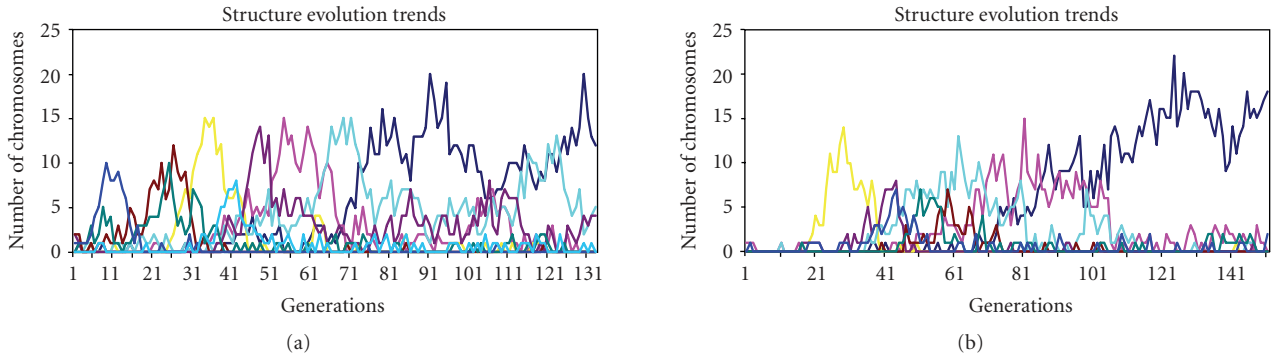



Figure 15: Structure evolution trends for (a) 3-bit and (b) 4-bit parity examples. Each color represents a unique BbNN structure.

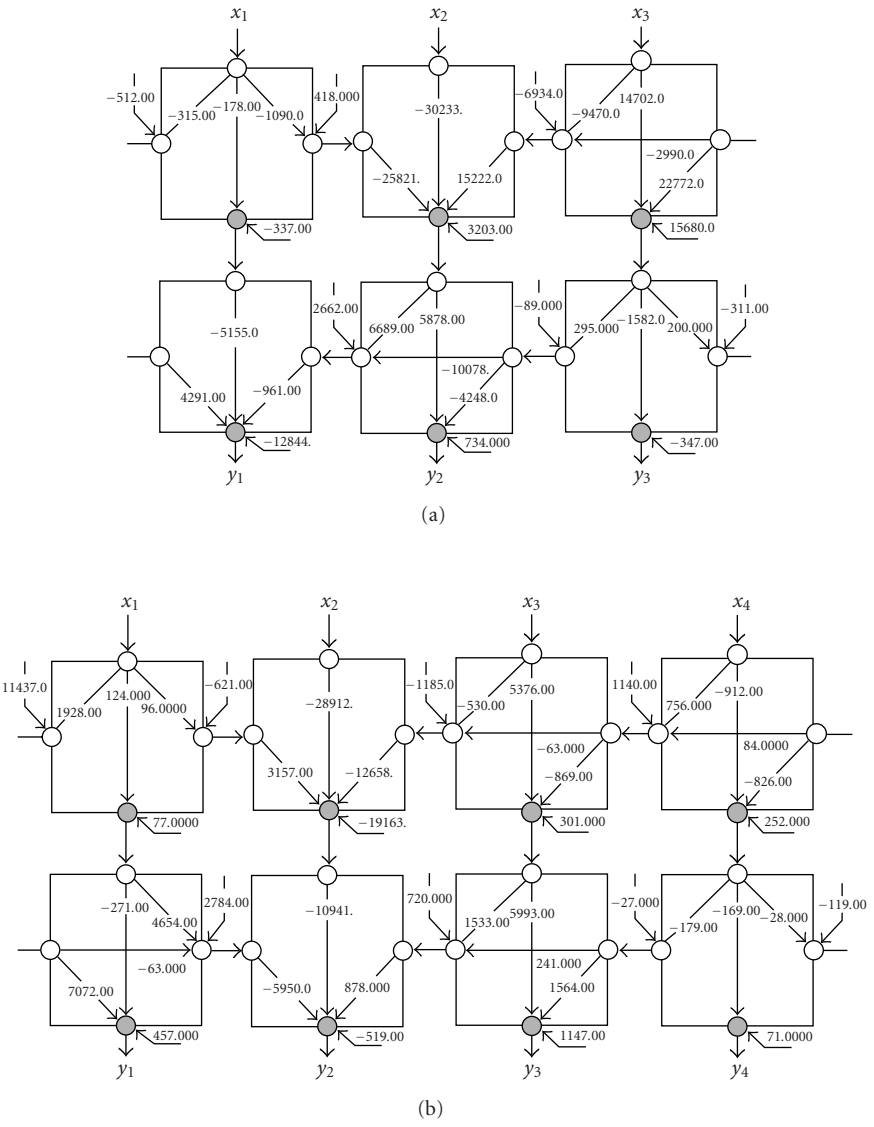



Figure 16: Evolved networks for (a) 3-bit and (b) 4-bit parity examples.

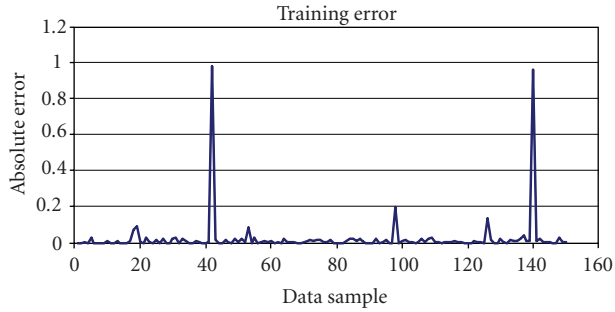



Figure 17: Training error for Iris data classification.

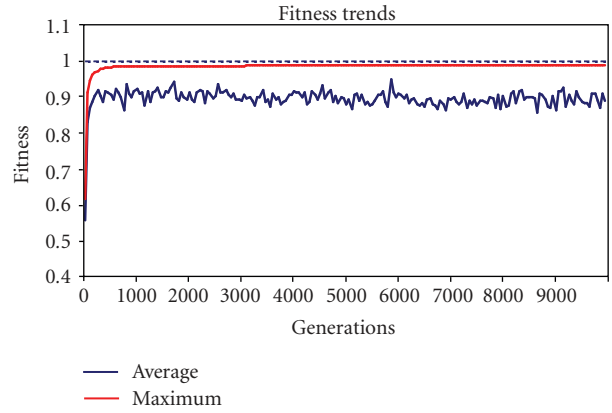



Figure 18: BbNN fitness evolution trends for Iris data classification.

of 80 chromosomes is used for evolution and crossover and mutation probabilities set at 0.7 and 0.2, respectively. Maximum fitness of 0.99 is achieved after 9403 generations. Figure 18 shows the average and maximum fitness trends. Figure 19 shows the structure evolution trends. Figure 20 shows the evolved network. The average evolution time to produce a new generation was found to be 23 microseconds.

*5.3. Adaptive Forward Prediction.* The objective of this case study is to demonstrate online training benefits of the BbNN platform. This feature is beneficial for applications in dynamic environments where changing conditions may otherwise require offline retraining and deployment to maintain system reliability. This simulation study will use a BbNN to predict future values of ambient luminosity levels in a room. The network will be pretrained offline to predict ambient luminosity levels in an ideal setup and then deployed in the test room. The actual ambient luminosity levels in the test room can be different from the training data due to various external factors such as a sunny or a cloudy day, number of open windows, and closed or open curtains. The actual levels can be recorded in real time using light sensors and used for online training of the BbNN predictor to improve its future predictions. This study could be applied to many applications sensitive to luminosity variations such as embryonic cell or plant tissue cultures.

*5.3.1. Offline Training Experimental Setup and Results.* The pretraining setup for our experiment is as follows. Figure 21(a) shows the normalized values of the ambient luminosity variations during the course of a day for the simulated training room. The plot also shows the training result for the BbNN. Figure 21(b) shows the training error. The training dataset for the network consists of past four luminosity observations as the inputs and the next luminosity level as the target output. Figure 22 shows the average and maximum fitness trends over the course of genetic evolution and the evolution parameters used for the training. Figure 23 shows the evolved network.

*5.3.2. Online Training Experimental Setup and Results.* The evolved network from the previous step is deployed in the simulated test room. Two case studies are considered that simulate the ambient luminosity variations in the test room. The first represents a cloudy day with lower ambient luminosity levels as compared to the ones considered in the offline training step and the second represents a sunny day with higher luminosity levels. These are shown in Figure 24.

The deployed BbNN platform is set up to trigger an online retraining cycle on observing greater than 5% prediction error. For the cloudy day test case the BbNN predicts the ambient luminosity reasonably well until 7:50 hours when the first retraining trigger is issued. The second retraining

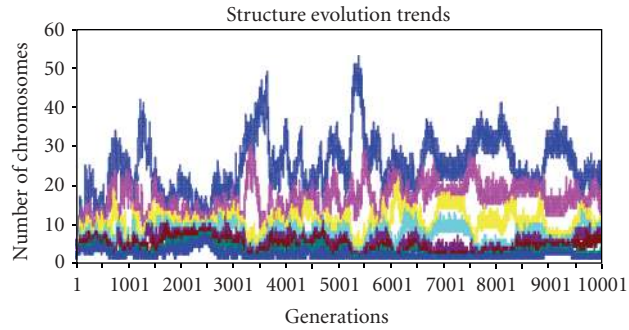



Figure 19: BbNN structure evolution trends for Iris data classification. Each color represents a unique BbNN structure.

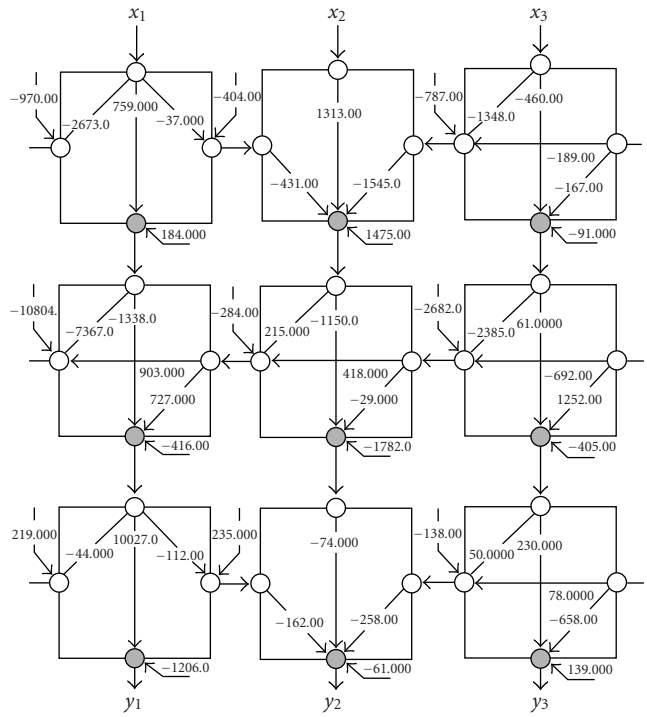



Figure 20: Evolved network for Iris data classification.

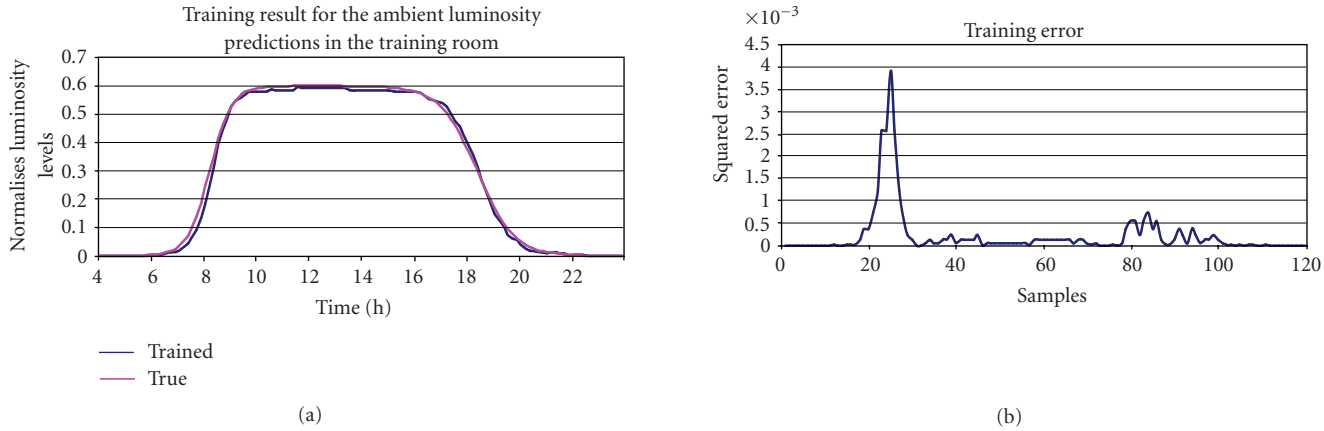
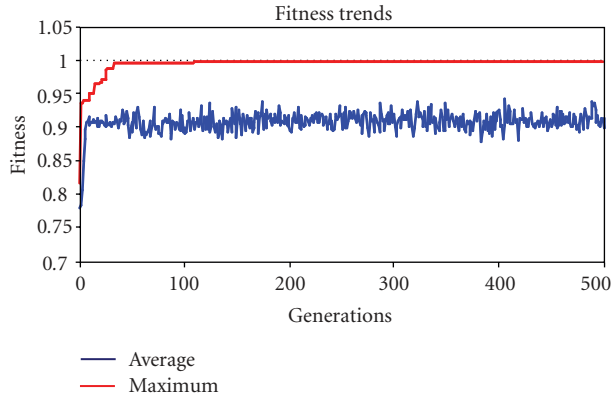



Figure 21: Pretraining result: (a) actual and predicted luminosity levels, and (b) training error.




(a)

| Parameter | Value |
|---|---|
| Activation function | Hypergolic tangent function |
| Selection strategy | Tournament selection |
| Population size | 80 |
| Maximum generations | 2000 |
| Structure crossover probability | 0.7 |
| Structure mutation probability | 0.3 |
| Weight mutation probability | 0.3 |
| Number of patterns | 120 |
| Inputs per pattern | 4 |
| Evolution strategy | Ellitist evolution |

(b)

Figure 22: (a) Average and maximum fitness values. (b) GA parameters used for training.

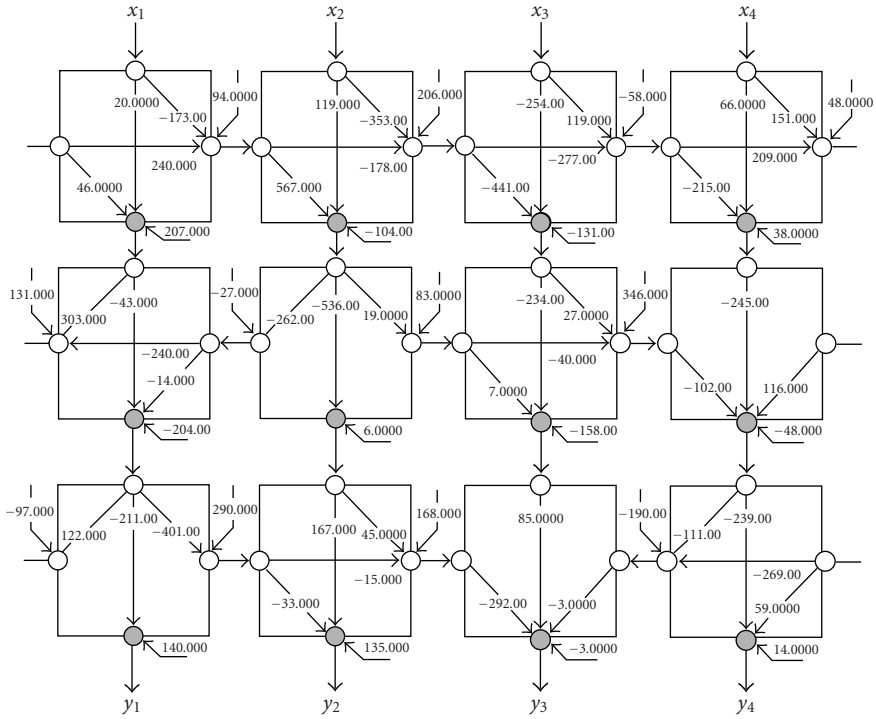



Figure 23: Evolved network after pre-training.

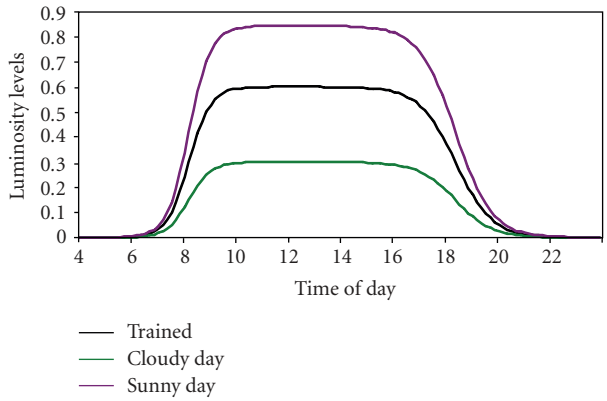



Figure 24: Luminosity levels used for offline training, cloudy day, and sunny day test cases.

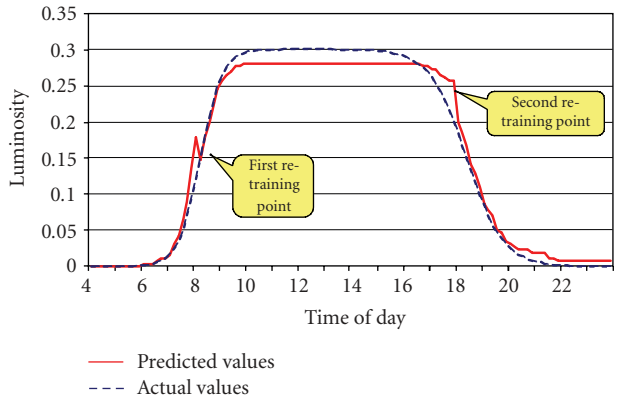



Figure 25: Online evolution operation for the cloudy day showing the two trigger points.

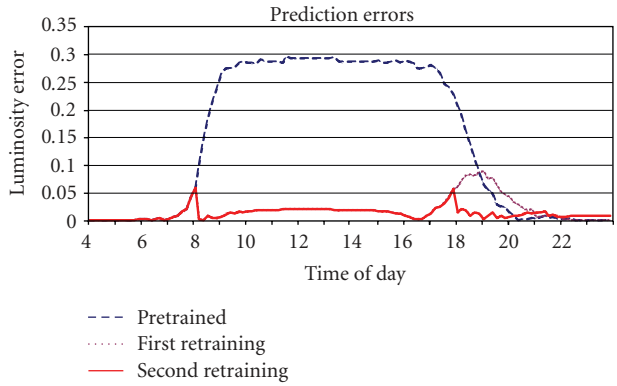



Figure 26: Prediction errors with and without online re-training.

Table 2: Device Utilization Summary on Xilinx Virtex-II Pro FPGA (XC2VP30).

| Network size | Number of slice registers | | Number of block RAMs | | Number of MULT18 × 18s | |
|---|---|---|---|---|---|---|
| | Used | Utilization | Used | Utilization | Used | Utilization |
| 2 × 2 | 2724 | 19% | 8 | 5% | 12 | 8% |
| 2 × 4 | 4929 | 35% | 16 | 11% | 24 | 17% |
| 2 × 6 | 7896 | 57% | 24 | 17% | 36 | 26% |
| 2 × 8 | 10589 | 77% | 32 | 23% | 48 | 35% |
| 2 × 10 | 12408 | 90% | 40 | 29% | 60 | 44% |
| 3 × 2 | 3661 | 26% | 8 | 5% | 18 | 13% |
| 3 × 4 | 7327 | 53% | 16 | 11% | 36 | 26% |
| 3 × 6 | 11025 | 80% | 24 | 17% | 54 | 39% |
| 3 × 8 | 14763 | 107% | 32 | 23% | 72 | 52% |
| 3 × 10 | 18456 | 134% | 40 | 29% | 90 | 66% |
| 4 × 2 | 4783 | 34% | 8 | 5% | 24 | 17% |
| 4 × 4 | 9646 | 70% | 16 | 11% | 48 | 35% |
| 4 × 6 | 14587 | 106% | 24 | 17% | 72 | 52% |
| 4 × 8 | 19508 | 142% | 32 | 23% | 96 | 70% |
| 4 × 10 | 24461 | 178% | 40 | 29% | 120 | 88% |

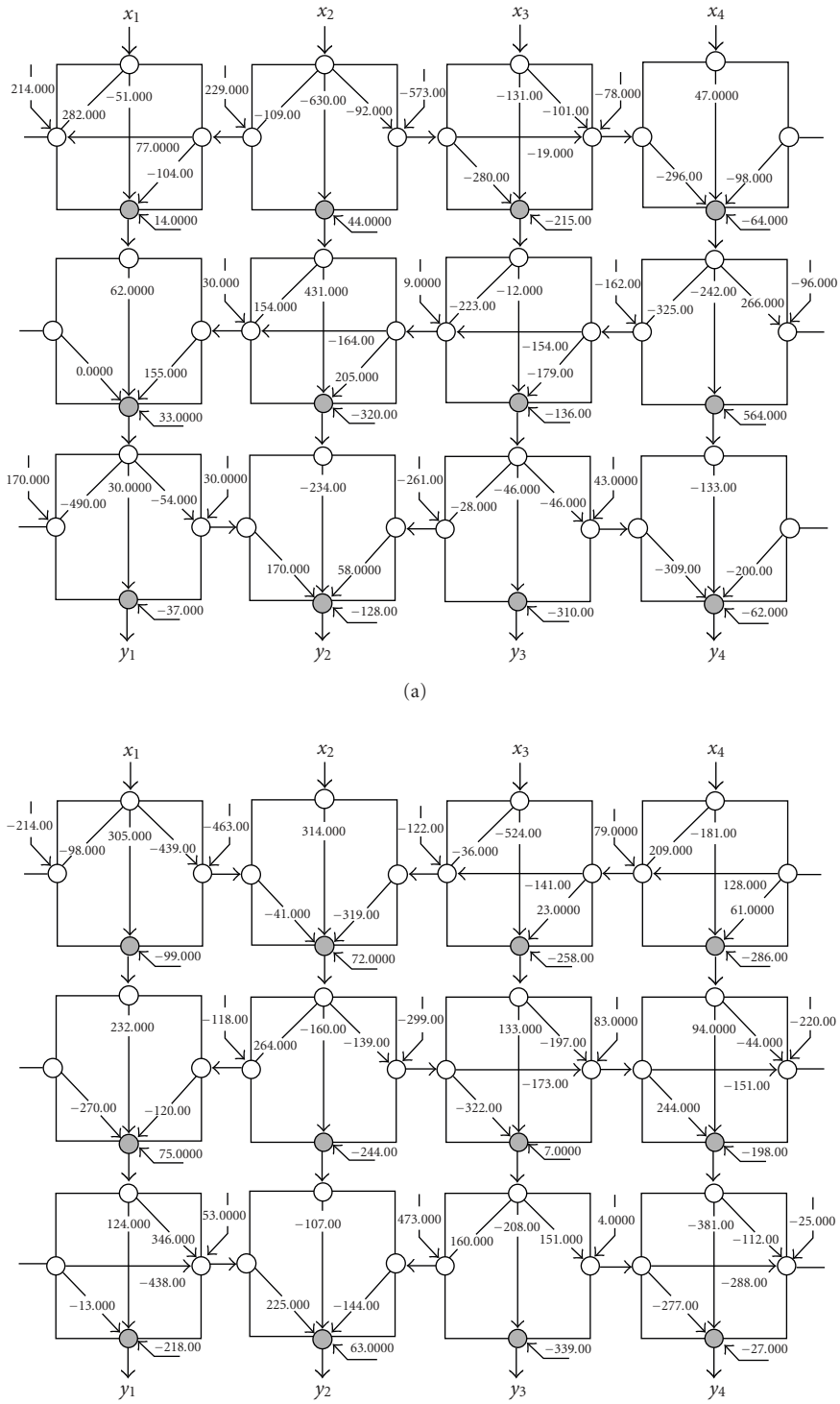



Figure 27: Evolved networks after retraining triggers: (a) first trigger point and (b) second trigger point.

trigger is issued at 17:50 hours. The actual and predicted luminosity values and the corresponding trigger points are shown in Figure 25. The prediction errors with and without online retraining are shown in Figure 26. Evolved networks after the first and second retraining cycles are shown in Figure 27.

In the sunny day test case the pretrained network performs poorly and requires eight retraining trigger points as shown in Figure 28. Figure 29 shows the prediction errors with and without retraining cycles. It can be seen that the network fails to correctly predict the transient rise in the luminosity level and over estimates after the first few

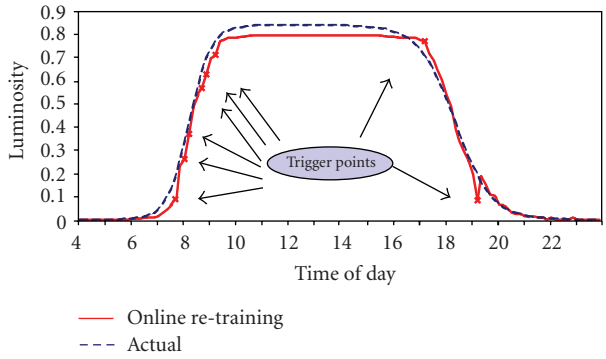



Figure 28: Online training result for the sunny day test case.

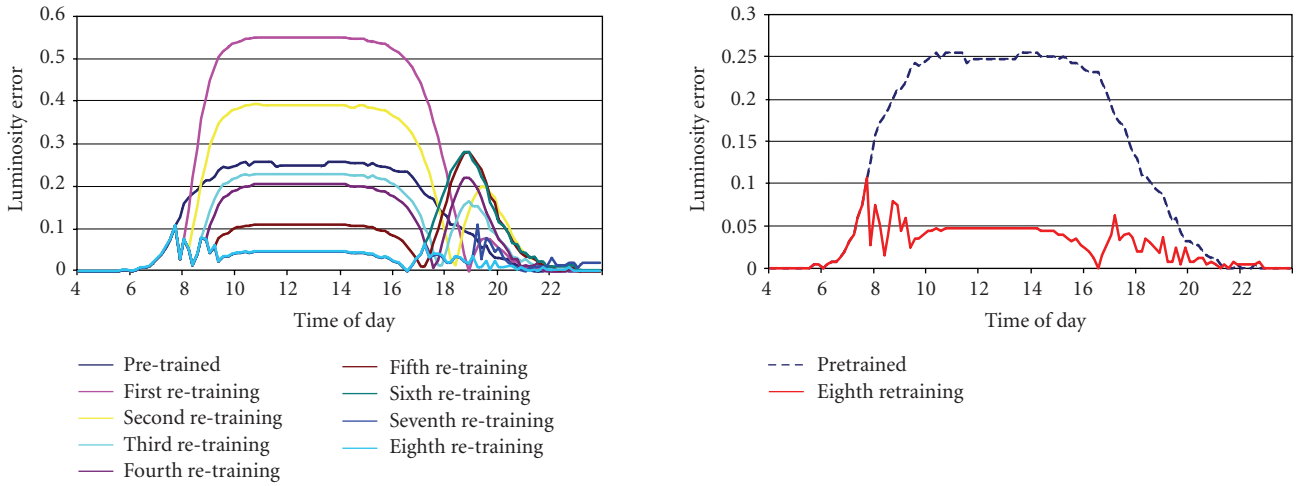



Figure 29: Prediction errors with and without retraining cycles for the sunny day test case.

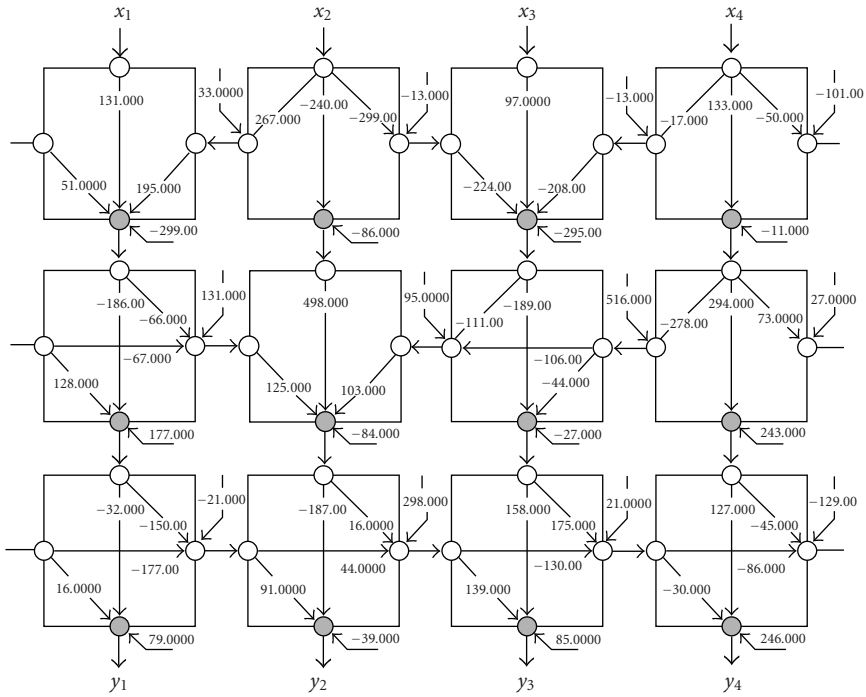



Figure 30: Evolved network after the eighth re-training cycle for the sunny day test case.

Table 3: Device utilization summary on Xilinx Virtex-II pro FPGA (XC2VP70).

| Network size | Number of slice registers | | Number of block RAMs | | Number of MULT18 × 18s | |
|---|---|---|---|---|---|---|
| | Used | Utilization | Used | Utilization | Used | Utilization |
| 2 × 2 | 2497 | 7% | 8 | 2% | 12 | 3% |
| 2 × 4 | 4929 | 14% | 16 | 4% | 24 | 7% |
| 2 × 6 | 7390 | 22% | 24 | 7% | 36 | 10% |
| 2 × 8 | 9915 | 29% | 32 | 9% | 48 | 14% |
| 2 × 10 | 12403 | 37% | 40 | 12% | 60 | 18% |
| 3 × 2 | 3661 | 11% | 8 | 2% | 18 | 5% |
| 3 × 4 | 7327 | 22% | 16 | 4% | 36 | 10% |
| 3 × 6 | 11025 | 33% | 24 | 7% | 54 | 16% |
| 3 × 8 | 14788 | 44% | 32 | 39% | 72 | 9% |
| 3 × 10 | 18461 | 55% | 40 | 12% | 90 | 27% |
| 3 × 12 | 22233 | 67% | 48 | 14% | 108 | 33% |
| 3 × 14 | 25652 | 77% | 56 | 17% | 126 | 38% |
| 3 × 16 | 29254 | 88% | 64 | 19% | 144 | 43% |
| 4 × 2 | 4783 | 14% | 8 | 2% | 24 | 7% |
| 4 × 4 | 9646 | 29% | 16 | 4% | 48 | 14% |
| 4 × 6 | 14561 | 44% | 24 | 7% | 72 | 21% |
| 4 × 8 | 19534 | 59% | 32 | 9% | 96 | 29% |
| 4 × 10 | 24470 | 73% | 40 | 12% | 120 | 36% |
| 4 × 12 | 29221 | 88% | 48 | 14% | 144 | 43% |
| 4 × 14 | 34389 | 103% | 56 | 17% | 168 | 51% |

retraining cycles necessitating multiple retrains. Figure 30 shows the evolved network after the eighth re-training cycle.

This case study demonstrates the online training benefits of the BbNN platform and its potential for applications in dynamic environments. Figures 26 and 29 demonstrate the performance of the network with and without re-training. Our study has not taken into consideration the time required to retrain the network. This time is dependent on the actual platform setup which can vary depending on the implementation constraints. Our design prototype currently uses the on-chip PPC 405 processor for executing the GA operators. For rapid convergence of the GA algorithm faster processors can be used with BbNN core executing on FPGAs for speeding up fitness evaluations.

## 6. Conclusions

In this paper we present an FPGA design for BbNNs that is capable of on-chip training under the control of genetic algorithms. Typical design process for ANNs involves a training phase performed using software simulations and the obtained network is designed and deployed offline. Hence, the training and design processes have to be repeated offline for every new application of ANNs. The novel online adaptability features of our design demonstrated using several case studies expand the potential applications for BbNNs to dynamic environments and provide increased deployment lifetimes and improved system reliability. The platform has been prototyped on two FPGA boards: a stand-alone Digilent Inc. XUP development board [98] and an Amirix AP130 development board [99], each with a Xilinx Virtex-II Pro FPGA. The ANN design can achieve peak computational capacity of 147 MCPS per neuron block on the Virtex-II Pro FPGAs. The paper presents three case studies. The first two, the n-bit parity classification and the Iris data classification, demonstrate the functionality of the designed platform. The case study on adaptive forward prediction demonstrates the benefits of online evolution under dynamically changing conditions. This work provides a platform for further research on design scalability, online unsupervised training algorithms, and applications of BbNNs in dynamic environments. To further speedup the evolution process, parallel GA algorithms can be used that can take advantage of multiple on-chip PowerPC processors per FPGA as well as scaling the design across multiple FPGAs. These are topics for further research.

## Acknowledgments

This work was supported in part by the National Science Foundation under Grant nos. ECS-0319002 and CCF-0311500. The authors also acknowledge the support of the UT Exhibit, Performance, and Publication Expense Fund

and thank the reviewers for their valuable comments which helped them in improving this manuscript.

## References


[1] S. G. Merchant and G. D. Peterson, "An evolvable artificial neural network platform for dynamic environments," in *Proceedings of the 51st Midwest Symposium on Circuits and Systems (MWSCAS '08)*, pp. 77–80, Knoxville, Tenn, USA, August 2008.

[2] H. de Garis, "Evolvable Hardware: Principles and Practice," *Communications of the Association for Computer Machinery (CACM Journal)*, August 1997.

[3] J. N. H. Heemskerk, "Overview of neural hardware," in *Neurocomputers for Brain-Style Processing: Design, Implementation and Application*, Unit of Experimental and Theoretical Psychology, Leiden University, Leiden, The Netherlands, 1995.

[4] H. P. Graf and L. D. Jackel, "Advances in neural network hardware," in *Proceedings of the International Electron Devices Meeting (IEDM '88)*, pp. 766–769, San Francisco, Calif, USA, December 1988.

[5] D. R. Collins and P. A. Penz, "Considerations for neural network hardware implementations," in *Proceedings of the 22nd IEEE International Symposium on Circuits and Systems (ISCAS '89)*, vol. 2, pp. 834–836, Portland, Ore, USA, May 1989.

[6] P. Ienne, "Architectures for neuro-computers: review and performance evaluation," Tech. Rep. 93/21, Swiss Federal Institute of Technology, Zurich, Switzerland, 1993.

[7] P. Ienne and G. Kuhn, "Digital systems for neural networks," in *Digital Signal Processing Technology*, P. Papamichalis and R. Kerwin, Eds., vol. 57, SPIE Optical Engineering, Orlando, Fla, USA, 1995.

[8] I. Aybay, S. Cetinkaya, and U. Halici, "Classification of neural network hardware," *Neural Network World*, vol. 6, no. 1, pp. 11–29, 1996.

[9] H. C. Card, G. K. Rosendahl, D. K. McNeill, and R. D. McLeod, "Competitive learning algorithms and neurocomputer architecture," *IEEE Transactions on Computers*, vol. 47, no. 8, pp. 847–858, 1998.

[10] T. Schoenauer, A. Jahnke, U. Roth, and H. Klar, "Digital neurohardware: principles and perspectives," in *Proceedings of the 3rd International Workshop on Neural Networks in Applications (NN '98)*, Magdeburg, Germany, February 1998.

[11] L. M. Reyneri, "Theoretical and implementation aspects of pulse streams: an overview," in *Proceedings of the 7th International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems*, pp. 78–89, Granada, Spain, April 1999.

[12] B. Linares-Barranco, A. G. Andreou, G. Indiveri, and T. Shibata, "Special issue on neural networks hardware implementations," *IEEE Transactions on Neural Networks*, vol. 14, no. 5, pp. 976–979, 2003.

[13] J. Zhu and P. Sutton, "FPGA implementations of neural networks—a survey of a decade of progress," in *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL '03)*, pp. 1062–1066, Lisbon, Portugal, September 2003.

[14] N. Aibe, M. Yasunaga, I. Yoshihara, and J. H. Kim, "A probabilistic neural network hardware system using a learning-parameter parallel architecture," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '02)*, vol. 3, pp. 2270–2275, Honolulu, Hawaii, USA, May 2002.

[15] J. L. Ayala, A. G. Lomeña, M. López-Vallejo, and A. Fernández, "Design of a pipelined hardware architecture for real-time neural network computations," in *Proceedings of the 45th Midwest Symposium on Circuits and Systems (MWSCAS '02)*, vol. 1, pp. 419–422, Tulsa, Okla, USA, August 2002.

[16] U. Ramacher, "Synapse-X: a general-purpose neurocomputer architecture," in *Proceedings of IEEE International Joint Conference on Neural Networks (IJCNN '91)*, vol. 3, pp. 2168–2176, Seattle, Wash, USA, July 1991.

[17] 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., pp. 37–61, Springer, Berlin, Germany, 2006.

[18] K. R. Nichols, M. A. Moussa, and S. M. Areibi, "Feasibility of floating-point arithmetic in FPGA based artificial neural networks," in *Proceedings of the 15th International Conference on Computer Applications in Industry and Engineering (CAINE '02)*, San Diego, Calif, USA, November 2002.

[19] J. L. Holt and T. E. Baker, "Back propagation simulations using limited precision calculations," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '91)*, pp. 121–126, Seattle, Wash, USA, July 1991.

[20] J. L. Holt and J.-N. Hwang, "Finite precision error analysis of neural network electronic hardware implementations," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN'91)*, pp. 519–525, Seattle, Washington, USA, July 1991.

[21] J. L. Holt and J.-N. Hwang, "Finite precision error analysis of neural network hardware implementations," *IEEE Transactions on Computers*, vol. 42, no. 3, pp. 281–290, 1993.

[22] E. Ros, E. M. Ortigosa, R. Agis, R. Carrillo, and M. Arnold, "Real-time computing platform for spiking neurons (RT-spike)," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 1050–1063, 2006.

[23] M. Porrmann, U. Witkowski, H. Kalte, and U. Ruckert, "Implementation of artificial neural networks on a reconfigurable hardware accelerator," in *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing*, pp. 243–250, Canary Islands, Spain, January 2002.

[24] C. Torres-Huitzil and B. Girau, "FPGA implementation of an excitatory and inhibitory connectionist model for motion perception," in *Proceedings of IEEE International Conference on Field Programmable Technology (FPT '05)*, pp. 259–266, Singagore, December 2005.

[25] S. Kothandaraman, "Implementation of block-based neural networks on reconfigurable computing platforms," MS Report, Electrical and Computer Engineering Department, University of Tennessee, Knoxville, Tenn, USA, 2004.

[26] D. Ferrer, R. González, R. Fleitas, J. P. Acle, and R. Canetti, "NeuroFPGA—implementing artificial neural networks on programmable logic devices," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 3, pp. 218–223, Paris, France, February 2004.

[27] C. T. Yen, W.-D. Weng, and Y. T. Lin, "FPGA realization of a neural-network-based nonlinear channel equalizer," *IEEE Transactions on Industrial Electronics*, vol. 51, no. 2, pp. 472–479, 2004.

[28] Q. Wang, B. Yi, Y. Xie, and B. Liu, "The hardware structure design of perceptron with FPGA implementation," in *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, vol. 1, pp. 762–767, Washington, DC, USA, October 2003.

[29] M. M. Syiam, H. M. Klash, I. I. Mahmoud, and S. S. Haggag, "Hardware implementation of neural network on FPGA for accidents diagnosis of the multi-purpose research reactor of Egypt," in *Proceedings of the 15th International Conference on Microelectronics (ICM '03)*, pp. 326–329, Cairo, Egypt, December 2003.

[30] M. Krips, T. Lammert, and A. Kummert, "FPGA implementation of a neural network for a real-time hand tracking system," in *Proceedings of the 1st IEEE International Workshop on Electronic Design, Test and Applications*, pp. 313–317, Christchurch, New Zealand, January 2002.

[31] J. Zhu, G. J. Milne, and B. K. Gunther, "Towards an FPGA based reconfigurable computing environment for neural network implementations," in *Proceedings of the 9th International Conference on Artificial Neural Networks (ICANN '99)*, vol. 2, pp. 661–666, Edinburgh, UK, September 1999.

[32] S. Happe and H.-G. Kranz, "Practical applications for the machine intelligent partial discharge disturbing pulse suppression system NeuroTEK II," in *Proceedings of the 11th International Symposium on High Voltage Engineering (ISH '99)*, vol. 5, pp. 37–40, London, UK, August 1999.

[33] M. Marchesi, G. Orlandi, F. Piazza, and A. Uncini, "Fast neural networks without multipliers," *IEEE Transactions on Neural Networks*, vol. 4, no. 1, pp. 53–62, 1993.

[34] M. van Daalen, T. Kosel, P. Jeavons, and J. Shawe-Taylor, "Emergent activation functions from a stochastic bit-stream neuron," *Electronics Letters*, vol. 30, no. 4, pp. 331–333, 1994.

[35] E. van Keulen, S. Colak, H. Withagen, and H. Hegt, "Neural network hardware performance criteria," in *Proceedings of IEEE International Conference on Neural Networks*, vol. 3, pp. 1955–1958, Orlando, Fla, USA, June 1994.

[36] H. O. Johansson, P. Larsson, P. Larsson-Edefors, and C. Svensson, "A 200-MHz CMOS bit-serial neural network," in *Proceedings of the 7th Annual IEEE International ASIC Conference and Exhibit*, pp. 312–315, Rochester, NY, USA, September 1994.

[37] M. Gschwind, V. Salapura, and O. Maischbergeres, "Space efficient neural net implementation," in *Proceedings of the 2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, Berkeley, Calif, USA, February 1994.

[38] A. F. Murray and A. V. W. Smith, "Asynchronous VLSI neural networks using pulse-stream arithmetic," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 3, pp. 688–697, 1988.

[39] P. Lysaght, J. Stockwood, J. Law, and D. Girma, "Artificial neural network implementation on a fine-grained FPGA," in *Proceedings of the 4th International Workshop on Field-Programmable Logic and Applications (FPL '94)*, pp. 421–431, Prague, Czech Republic, September 1994.

[40] V. Salapura, "Neural networks using bit stream arithmetic: a space efficient implementation," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '94)*, vol. 6, pp. 475–478, London, UK, May 1994.

[41] N. Chujo, S. Kuroyanagi, S. Doki, and S. Okuma, "An iterative calculation method of neuron model with sigmoid function," in *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, vol. 3, pp. 1532–1537, Tucson, Ariz, USA, October 2001.

[42] S. A. Guccione and M. J. Gonzalez, "Neural network implementation using reconfigurable architectures," in *Selected Papers from the Oxford 1993 International Workshop on Field Programmable Logic and Applications on More FPGAs*, pp. 443–451, Abingdon EE & CS Books, Oxford, UK, 1994.

[43] L. Mintzer, "Digital filtering in FPGAs," in *Proceedings of the 28th Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1373–1377, Pacific Grove, Calif, USA, November 1994.

[44] T. Szabo, L. Antoni, G. Horvath, and B. Feher, "A full-parallel digital implementation for pre-trained NNs," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '00)*, vol. 2, pp. 49–54, Como, Italy, July 2000.

[45] B. Noory and V. Groza, "A reconfigurable approach to hardware implementation of neural networks," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, vol. 3, pp. 1861–1864, Montreal, Canada, May 2003.

[46] E. Pasero and M. Perri, "Hw-Sw codesign of a flexible neural controller through a FPGA-based neural network programmed in VHDL," in *Proceedings of IEEE International Joint Conference on Neural Networks (IJCNN '04)*, vol. 4, pp. 3161–3165, Budapest, Hungary, July 2004.

[47] C.-H. Kung, M. J. Devaney, C.-M. Kung, C.-M. Huang, Y.-J. Wang, and C.-T. Kuo, "The VLSI implementation of an artificial neural network scheme embedded in an automated inspection quality management system," in *Proceedings of the 19th IEEE Instrumentation and Measurement Technology Conference (IMTC '02)*, vol. 1, pp. 239–244, Anchorage, Alaska, USA, May 2002.

[48] J. G. Eldredge and B. L. Hutchings, "Density enhancement of a neural network using FPGAs and run-time reconfiguration," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 180–188, Napa Valley, Calif, USA, April 1994.

[49] J. G. Eldredge and B. L. Hutchings, "RRANN: a hardware implementation of the backpropagation algorithm using reconfigurable FPGAs," in *Proceedings of IEEE International Conference on Neural Networks, IEEE World Congress on Computational Intelligence*, vol. 4, pp. 2097–2102, Orlando, Fla, USA, June 1994.

[50] C. E. Cox and W. E. Blanz, "GANGLION—a fast field-programmable gate array implementation of a connectionist classifier," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 3, pp. 288–299, 1992.

[51] A. Perez-Uribe and E. Sanchez, "FPGA implementation of an adaptable-size neural network," in *Proceedings of the 6th International Conference on Artificial Neural Networks (ICANN '96)*, pp. 383–388, Bochum, Germany, July 1996.

[52] H. F. Restrepo, R. Hoffmann, A. Perez-Uribe, C. Teuscher, and E. Sanchez, "A networked FPGA-based hardware implementation of a neural network application," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 337–338, Napa Valley, Calif, USA, April 2000.

[53] I. Kajitani, M. Murakawa, D. Nishikawa, et al., "An evolvable hardware chip for prosthetic hand controller," in *Proceedings of the 7th International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems (MicroNeuro '99)*, pp. 179–186, Granada, Spain, April, 1999.

[54] K. Mathia and J. Clark, "On neural network hardware and programming paradigms," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '02)*, vol. 3, pp. 2692–2697, Honolulu, Hawaii, USA, June 2002.

[55] D. Hajtas and D. Durackova, "The library of building blocks for an "integrate & fire" neural network on a chip," in *Proceedings of IEEE International Joint Conference on Neural Networks (IJCNN '04)*, vol. 4, pp. 2631–2636, Budapest, Hungary, July 2004.

[56] Jayadeva and S. A. Rahman, "A neural network with O(N) neurons for ranking N numbers in O(1/N) time," *IEEE Transactions on Circuits and Systems I*, vol. 51, no. 10, pp. 2044–2051, 2004.

[57] J. D. Hadley and B. L. Hutchings, "Design methodologies for partially reconfigured systems," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 78–84, Napa Valley, Calif, USA, April 1995.

[58] R. Gadea, J. Cerda, F. Ballester, and A. Macholi, "Artificial neural network implementation on a single FPGA of a pipelined on-line backpropagation," in *Proceedings of the 13th International Symposium on System Synthesis*, pp. 225–230, Madrid, Spain, September 2000.

[59] K. Paul and S. Rajopadhye, "Back-propagation algorithm achieving 5 Gops on the Virtex-E," in *FPGA Implementations of Neural Networks*, pp. 137–165, Springer, Berlin, Germany, 2006.

[60] U. Witkowski, T. Neumann, and U. Ruckert, "Digital hardware realization of a hyper basis function network for on-line learning," in *Proceedings of the 7th International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems (MicroNeuro '99)*, pp. 205–211, Granada, Spain, April 1999.

[61] M. Murakawa, S. Yoshizawa, I. Kajitani, et al., "The GRD chip: genetic reconfiguration of DSPs for neural network processing," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 628–639, 1999.

[62] D. Hammerstrom, "A VLSI architecture for high-performance, low-cost, on-chip learning," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '90)*, vol. 2, pp. 537–544, San Diego, Calif, USA, June 1990.

[63] Y. Sato, K. Shibata, M. Asai, et al., "Development of a high-performance, general purpose neuro-computer composed of 512 digital neurons," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '93)*, vol. 2, pp. 1967–1970, Nagoya, Japan, October 1993.

[64] T. Tang, O. Ishizuka, and H. Matsumoto, "Backpropagation learning in analog T-model neural network hardware," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '93)*, vol. 1, pp. 899–902, Nagoya, Japan, October 1993.

[65] B. Linares-Barranco, E. Sanchez-Sinencio, A. Rodriguez-Vazquez, and J. L. Huertas, "A CMOS analog adaptive BAM with on-chip learning and weight refreshing," *IEEE Transactions on Neural Networks*, vol. 4, no. 3, pp. 445–455, 1993.

[66] E. Farquhar, C. Gordon, and P. Hasler, "A field programmable neural array," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '06)*, pp. 4114–4117, Kos, Greece, May 2006.

[67] F. Tenore, R. J. Vogelstein, R. Etienne-Cummings, G. Cauwenberghs, M. A. Lewis, and P. Hasler, "A spiking silicon central pattern generator with floating gate synapses [robot control applications]," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '05)*, vol. 4, pp. 4106–4109, Kobe, Japan, May 2005.

[68] G. Indiveri, E. Chicca, and R. J. Douglas, "A VLSI reconfigurable network of integrate-and-fire neurons with spike-based learning synapses," in *Proceedings of the 12th European Symposium on Artificial Neural Networks (ESANN '04)*, Bruges, Belgium, April 2004.

[69] B. Girau, "FPNA: applications and implementations," in *FPGA Implementations of Neural Networks*, pp. 103–136, Springer, Berlin, Germany, 2006.

[70] B. Girau, "FPNA: concepts and properties," in *FPGA Implementations of Neural Networks*, pp. 63–101, Springer, Berlin, Germany, 2006.

[71] M. Holler, S. Tam, H. Castro, and R. Benson, "An electrically trainable artificial neural network (ETANN) with 10240 'floating gate' synapses," in *Artificial Neural Networks: Electronic Implementations*, pp. 50–55, IEEE, New York, NY, USA, 1990.

[72] M. L. Mumford, D. K. Andes, and L. L. Kern, "The Mod 2 Neurocomputer system design," *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 423–433, 1992.

[73] T. Schmitz, S. Hohmann, K. Meier, J. Schemmel, and F. Schurmann, "Speeding up hardware evolution: a coprocessor for evolutionary algorithms," in *Evolvable Systems: From Biology to Hardware*, vol. 2606 of *Lecture Notes in Computer Science*, pp. 274–285, Springer, Berlin, Germany, 2003.

[74] A. Omondi, J. Rajapakse, and M. Bajger, "FPGA neurocomputers," in *FPGA Implementations of Neural Networks*, pp. 1–36, Springer, Berlin, Germany, 2006.

[75] L. D. Jackel, H. P. Graf, and R. E. Howard, "Electronic neural network chips," *Applied Optics*, vol. 26, no. 23, pp. 5077–5080, 1987.

[76] E. Farquhar and P. Hasler, "A bio-physically inspired silicon neuron," *IEEE Transactions on Circuits and Systems I*, vol. 52, no. 3, pp. 477–488, 2005.

[77] B. Linares-Barranco, E. Sanchez-Sinencio, A. Rodriguez-Vazquez, and J. L. Huertas, "A modular T-mode design approach for analog neural network hardware implementations," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 5, pp. 701–713, 1992.

[78] C. Gordon, E. Farquhar, and P. Hasler, "A family of floating-gate adapting synapses based upon transistor channel models," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '04)*, vol. 1, pp. 317–320, Vancouver, Canada, May 2004.

[79] E. Farquhar, D. Abramson, and P. Hasler, "A reconfigurable bidirectional active 2 dimensional dendrite model," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '04)*, vol. 1, pp. 313–316, Vancouver, Canada, May 2004.

[80] G. Cauwenberghs, C. F. Neugebauer, and A. Yariv, "An adaptive CMOS matrix-vector multiplier for large scale analog hardware neural network applications," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '91)*, vol. 1, pp. 507–511, Seattle, Wash, USA, July 1991.

[81] O. Barkan, W. R. Smith, and G. Persky, "Design of coupling resistor networks for neural network hardware," *IEEE Transactions on Circuits and Systems*, vol. 37, no. 6, pp. 756–765, 1990.

[82] T. J. Schwartz, "A neural chips survey," *AI Expert*, vol. 5, no. 12, pp. 34–38, 1990.

[83] A. J. Agranat, C. F. Neugebauer, and A. Yariv, "A CCD based neural network integrated circuit with 64 K analog programmable synapses," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '90)*, vol. 2, pp. 551–555, San Diego, Calif, USA, June 1990.

[84] L. W. Massengill and D. B. Mundie, "An analog neural hardware implementation using charge-injection multipliers and neuron-specific gain control," *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 354–362, 1992.

[85] A. Passos Almeida and J. E. Franca, "A mixed-mode architecture for implementation of analog neural networks with digital programmability," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '93)*, vol. 1, pp. 887–890, Nagoya, Japan, October 1993.

[86] M. R. DeYong, R. L. Findley, and C. Fields, "The design, fabrication, and test of a new VLSI hybrid analog-digital neural processing element," *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 363–374, 1992.

[87] E. Sackinger, B. E. Boser, J. Bromley, Y. LeCun, and L. D. Jackel, "Application of the ANNA neural network chip to high-speed character recognition," *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 498–505, 1992.

[88] G. Zatorre-Navarro, N. Medrano-Marqués, and S. Celma-Pueyo, "Analysis and simulation of a mixed-mode neuron architecture for sensor conditioning," *IEEE Transactions on Neural Networks*, vol. 17, no. 5, pp. 1332–1335, 2006.

[89] K. D. Maier, C. Beckstein, R. Blickhan, W. Erhard, and D. Fey, "A multi-layer-perceptron neural network hardware based on 3D massively parallel optoelectronic circuits," in *Proceedings of the 6th International Conference on Parallel Interconnects*, pp. 73–80, Anchorage, Alaska, USA, October 1999.

[90] M. P. Craven, K. M. Curtis, and B. R. Hayes-Gill, "Consideration of multiplexing in neural network hardware," *IEE Proceedings: Circuits, Devices and Systems*, vol. 141, no. 3, pp. 237–240, 1994.

[91] S.-W. Moon and S.-G. Kong, "Block-based neural networks," *IEEE Transactions on Neural Networks*, vol. 12, no. 2, pp. 307–317, 2001.

[92] W. Jiang, S. G. Kong, and G. D. Peterson, "ECG signal classification using block-based neural networks," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '05)*, vol. 1, pp. 326–331, Montreal, Canada, July 2005.

[93] W. Jiang, S. G. Kong, and G. D. Peterson, "Continuous heartbeat monitoring using evolvable block-based neural networks," in *Proceedings of IEEE International Joint Conference on Neural Networks (IJCNN '06)*, pp. 1950–1957, Vancouver, Canada, July 2006.

[94] W. Jiang and S. G. Kong, "Block-based neural networks for personalized ECG signal classification," *IEEE Transactions on Neural Networks*, vol. 18, no. 6, pp. 1750–1761, 2007.

[95] S. G. Kong, "Time series prediction with evolvable block-based neural networks," in *Proceedings of IEEE International Joint Conference on Neural Networks (IJCNN '04)*, vol. 2, pp. 1579–1583, Budapest, Hungary, July 2004.

[96] S. Merchant, G. D. Peterson, S. K. Park, and S. G. Kong, "FPGA implementation of evolvable block-based neural networks," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC '06)*, pp. 3129–3136, Vancouver, Canada, July 2006.

[97] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet," Product Specification, DS083 (v4.6), March 2007.

[98] Xilinx, "Xilinx University Program Virtex-II Pro Development System Hardware Reference Manual," Hardware Reference Manual, UG069 (v1.0), March 2005.

[99] Amirix, "AMIRIX Systems Inc. PCI Platform FPGA Development Board Users Guide," User Guide, DOC-003266 Version 06, June 2004.

[100] R. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, pp. 179–188, 1936.