Neurocomputing 96 (2012) 57–65

Contents lists available at SciVerse ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

Applicability of approximate multipliers in hardware neural networks

Uroš Lotrič, Patricio Bulić *

University of Ljubljana, Faculty of Computer and Information Science, Ljubljana, Slovenia

ARTICLE INFO

Available online 11 May 2012

Keywords:
Hardware neural network
Iterative logarithmic multiplier
FPGA
Digital design
Computer arithmetic

ABSTRACT

In recent years there has been a growing interest in hardware neural networks, which express many benefits over conventional software models, mainly in applications where speed, cost, reliability, or energy efficiency are of great importance. These hardware neural networks require many resource-, power- and time-consuming multiplication operations, thus special care must be taken during their design. Since the neural network processing can be performed in parallel, there is usually a requirement for designs with as many concurrent multiplication circuits as possible.

One option to achieve this goal is to replace the complex exact multiplying circuits with simpler, approximate ones. The present work demonstrates the application of approximate multiplying circuits in the design of a feed-forward neural network model with on-chip learning ability. The experiments performed on a heterogeneous PROBEN1 benchmark dataset show that the adaptive nature of the neural network model successfully compensates for the calculation errors of the approximate multiplying circuits. At the same time, the proposed designs also profit from more computing power and increased energy efficiency.

1. Introduction

Artificial neural networks can be implemented in many different ways. For the majority of research and commercial needs it is now common to implement them as software for general-purpose processors. However, there are many niche applications, for example real-time systems requiring very large computational power, fault-tolerant systems for the processing of safety-critical tasks, energy-efficient solutions for mobile devices, and massively produced price-sensitive consumer electronic products, for which the software solutions are not satisfactory.

In contrast to the mainly sequential software neural network models, the hardware implementations or hardware neural networks can take advantage of the neural network model architecture. In comparison to the ordinary approaches, the hardware designs can be better tailored to the processing needs, which can result in much higher performance and/or smaller power consumption.

Hardware neural networks are built using many different technologies ranging from digital, analogue, and hybrid microchips to even optical computing [1–3]. Focusing on microchips, the design of the application-specific integrated circuits (ASICs) is time consuming and requires a lot of resources. An interesting low-budget alternative has been found in reconfigurable devices, among which the field programmable gate array (FPGA) technology is the most widely known [4,3]. It consists of pre-built logic blocks and programmable routing resources that can be arbitrarily configured to implement custom hardware functionality. Although smaller and slower than ASIC solutions, the FPGA chips provide hardware-timed speed and reliability and rapid-prototyping capabilities. Moreover, their concept of independent building blocks allows us to build autonomous circuits leading to massively parallel designs.

Although the digital designs of hardware neural networks usually result in larger circuit sizes compared to the analogue ones, they have many advantages over them. They are less susceptible to noise and temperature variations and so the computation is repeatable and exact to the required precision. It is also possible to perform on-chip learning and store the obtained weights in the chip memory. Moreover, the digital design enables a straightforward integration of the hardware neural network module into the more complex digital designs [2,5,6].

The choice of technology dictates the usage of a neural network model and vice versa. Besides the implementations of models like feed-forward neural networks, radial bases function models, and Kohonen maps, new models are emerging, for example spiking, cellular and neuromorphic neural networks to mention just a few [2]. To clearly demonstrate our ideas, we have confined our work to the well-known feed-forward neural network with a sigmoid activation function and the integrated on-chip back-propagation learning ability.

The basic neural network processing unit is an artificial neuron. In the most commonly used McCulloch and Pitts model, the activation potential, calculated as the weighted sum of the

* Corresponding author. Tel.: +386 1 4768361; fax: +386 1 4264647.
E-mail address: patricio.bulic@fri.uni-lj.si (P. Bulić).

0925-2312/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.neucom.2011.09.039

neuron inputs, is passed to the non-linear activation function to get the neuron output. This processing involves a lot of multiplications and additions as well as a computation of non-linear functions. Many attempts were made to simplify and speed-up the above operations. Skrbek [7] presented an optimized implementation of multiplication, square root, logarithm, exponent and non-linear activation functions by using only linear approximations, which in hardware design simplify to shift registers and adders only. A lot of work has also been done on the calculation and mapping of the activation functions [5–8].

To make efficient use of the parallelism that is inherently present in neural network models, the designs which enable concurrent use of a large number of multiplication circuits are desired. When resources are limited, the number of multiplication circuits can be increased only if the circuits are implemented by using fewer resources. Due to the complexity of the circuits needed for the floating-point operations, the first idea is to constrain the designs to the fixed-point implementations, which can make use of simpler integer adders and multipliers [9,10]. For an exact fixed-point multiplication a matrix multiplier is usually used. Unfortunately, it typically requires a lot of space on the chip, i.e., the m-bit matrix multiplier is composed of $m-2$ m-bit carry-save adders and one m-bit carry-propagate adder. Therefore, special care must be taken to minimize the bit precision of the inputs and weights in order to reduce its size [11].

For further optimization, ideas similar to the work of Skrbek [7] can be applied directly to the multipliers. Many approximate multiplication circuits exist, for example truncated and logarithmic multipliers [12–15], which consume fewer resources and less power and are faster than the exact multipliers. When using them, calculation errors might cause a serious degradation of the neural network's performance, if the teaching is performed off-chip. However, if the neural network learning is performed on-chip, the models should compensate for erroneous calculations during the learning phase, leading to simpler designs without considerably affecting the learning capability.

Truncated multipliers are extensively used in digital signal processing, where the importance of the multiplication speed as well as the resource and power consumption prevail over a high computation accuracy. The basic idea of these techniques is to discard some of the less significant partial products and to introduce a compensation circuit to reduce the approximation error [14].

Yet another approximate way to perform multiplication is to use a logarithmic approximation [12]. Logarithmic multiplication introduces an operand conversion from the integer number system into the logarithm number system. The multiplication of two operands is performed in three phases: calculating the operand logarithms, the addition of the operand logarithms and the calculation of the antilogarithm. The main advantage of this method is the substitution of the multiplication with addition. However, this simple idea has a substantial weakness—the logarithm and anti-logarithm cannot be calculated exactly. In the well-known Mitchell algorithm [12] for logarithmic multiplication, a significant error is caused by the first-order Taylor series expansion of the logarithm and the antilogarithm functions; therefore, an error-correction circuit is preferred.

The one stage iterative logarithmic multiplier [15] follows the ideas of Mitchell, but uses different error-correction circuits. The final hardware implementation involves only one adder and a few shifters, resulting in the reduced usage of logic resources and power consumption. As the 16-bit multiplier with one error correction circuit proposed by Babic et al. [15] showed substantial resource and power savings, while keeping the average relative error under 1%, we decided to investigate the applicability of such a multiplier in a hardware realization of neural networks.

The remainder of the paper is organized as follows. In the next section an approximate iterative logarithmic multiplier is presented in detail. In the third section the highly parallel neural processing unit used in our experiments is briefly described. Its design, specially suited for feed-forward neural networks, allows it to be used in the forward pass as well as the backward pass. In Section 4 the performance of the proposed solution is tested on many classification and regression benchmark problems. The performance figures are given in comparison to the hardware implementation using exact matrix multipliers as well as a floating-point implementation. The main findings on the applicability of approximate multipliers in hardware network designs are summarized at the end.

2. Iterative logarithmic multiplier

The iterative logarithmic multiplier (ILM) was proposed by Babic et al. in [15]. It simplifies the logarithm approximation introduced in [12] and introduces an iterative algorithm with various possibilities for achieving an error as small as required and the possibility of achieving an exact result.

2.1. Mathematical formulation

The logarithm of the product of two non-negative integer numbers, N_1 and N_2 can be written as the sum of the logarithms

$$\log_2(N_1 \cdot N_2) = \log_2 N_1 + \log_2 N_2. \tag{1}$$

By denoting $k_1 = \lfloor \log_2 N_1 \rfloor$ and $k_2 = \lfloor \log_2 N_2 \rfloor$, the logarithm of the product can be approximated as $k_1 + k_2$. In this case the calculation of the approximate product,

$$N_1 \cdot N_2 \approx 2^{k_1 + k_2} \tag{2}$$

requires only one add and one shift operation, but it has a large error.

To decrease this error, the following procedure is proposed in [15]. A non-negative integer number N can be written as

$$N = 2^k + N^{(1)}, \tag{3}$$

where k is a characteristic number, indicating the place of the leftmost 1 or the leading 1 bit in its binary representation, and the number $N^{(1)} = N - 2^k$ is the remainder of the number N after the removal of the leading 1.

Following the notation in Eq. (3), the product of two numbers can be written as

$$\begin{aligned} P_{\text{true}} = N_1 \cdot N_2 &= (2^{k_1} + N_1^{(1)}) \cdot (2^{k_2} + N_2^{(1)}) \\ &= 2^{k_1 + k_2} + N_1^{(1)} \cdot 2^{k_2} + N_2^{(1)} \cdot 2^{k_1} + N_1^{(1)} \cdot N_2^{(1)} \\ &= P_{\text{approx}}^{(0)} + E^{(0)}. \end{aligned} \tag{4}$$

While the first approximation of the product

$$P_{\text{approx}}^{(0)} = 2^{k_1 + k_2} + N_1^{(1)} \cdot 2^{k_2} + N_2^{(1)} \cdot 2^{k_1} \tag{5}$$

can be calculated by applying only a few shift and add operations, the term

$$E^{(0)} = N_1^{(1)} \cdot N_2^{(1)}, \quad E^{(0)} > 0, \tag{6}$$

representing the absolute error of the first approximation, requires multiplication.

Similarly, the proposed multiplication procedure can be performed on multiplicands from Eq. (6) such that

$$E^{(0)} = C^{(1)} + E^{(1)}, \tag{7}$$

where $C^{(1)}$ is the approximate value of $E^{(0)}$, and $E^{(1)}$ is the corresponding absolute error. The combination of Eqs. (4) and

(7) gives

$$P_{\text{true}} = P^{(0)}_{\text{approx}} + C^{(1)} + E^{(1)} = P^{(1)}_{\text{approx}} + E^{(1)}. \quad (8)$$

By repeating the described procedure we can obtain an arbitrarily precise approximation of the product by summing up iteratively the obtained correction terms $C^{(j)}$

$$P^{(i)}_{\text{approx}} = P^{(0)}_{\text{approx}} + \sum_{j=1}^{i} C^{(j)}. \quad (9)$$

The number of iterations required for an exact result is equal to the number of bits with the value of 1 in the operand with the smaller number of bits with the value of 1. Babic et al. [15] showed that in the worst-case scenario the relative error introduced by the proposed multiplier $E^{(i)}_r = E^{(i)}/N_1N_2$ decays exponentially with the rate $2^{-2(i+1)}$. Table 1 presents the average and maximal relative errors with respect to the number of considered iterations.

The proposed method assumes non-negative numbers. To apply the method on signed numbers, it is most appropriate to specify them in a sign and magnitude representation. In that case, the sign of the product is calculated as the EXOR (exclusive or) operation between the sign bits of both multiplicands.

2.2. Hardware implementation

The implementation of the proposed multiplier is described in [15]. The multiplier with one error correction circuit, which is used in the rest of the paper and shown in Fig. 1, is composed of two pipelined basic blocks, of which the first one calculates an approximate product $P^{(0)}_{\text{approx}}$, while the second one calculates the error-correction term $C^{(1)}$. The task of the basic block is to calculate one approximate product according to Eq. (5). To decrease the maximum combinational delay in the basic block, pipelining is used to implement the basic block. The pipelined implementation of the basic block is shown in Fig. 2 and has four stages. Stage 1 calculates the two characteristic numbers, k_1, k_2, and the two residues, $N_1^{(1)}$, $N_2^{(1)}$. The residues are outputted in stage 2, which also calculates k_1+k_2, $N_1^{(1)} \cdot 2^{k_2}$ and $N_2^{(1)} \cdot 2^{k_1}$. Stage 3 calculates $2^{k_1+k_2}$ and $N_1^{(1)} \cdot 2^{k_2} + N_2^{(1)} \cdot 2^{k_1}$, which are summed up to the approximation of the product $P^{(0)}_{\text{approx}}$ in stage 4. After the initial latency of five clock periods the proposed iterative logarithmic multiplier enables the products to be calculated in each clock period. The estimated device utilization in terms of programmable hardware components, i.e., slices and lookup tables, and dynamic power consumption at a frequency of 40 MHz for the 16-bit pipelined implementations of the proposed multiplier with one error-correction circuit and the classical matrix multiplier, are compared in Table 2. For the design entry, we used the Xilinx ISE 12.3-WebPACK and designed with VHDL. The design was synthesized with the Xilinx Xst Release 12.3. To analyze the power consumptions in the multipliers we used the Xilinx XPower Analyzer 12.3. The power consumption is estimated at a clock frequency of 40 MHz with a signal (toggle) rate of 12.5% and an output load of 5 pF. We have estimated only the dynamic (logic and signals) power, as the quiescent (leakage) power and the IOBs power are practically equal for both multipliers.

Table 1
Average and maximal relative errors for 16-bit iterative multiplier [15].

Number of iterations i	0	1	2	3
Average $E^{(i)}_r$ (%)	9.4	0.98	0.11	0.01
Max $E^{(i)}_r$ (%)	25.0	6.25	1.56	0.39

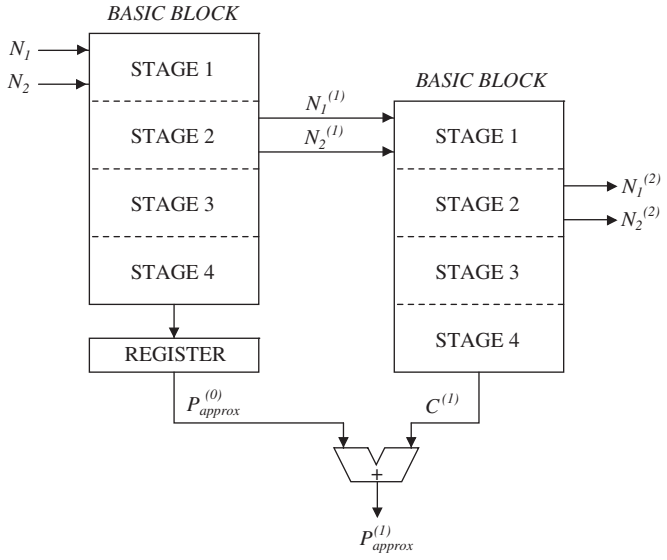

Fig. 1. Block diagram of a pipelined iterative logarithmic multiplier with one error-correction circuit.

3. Multilayer perceptron with a highly parallel neural unit

One of the most widely used neural networks is the multilayer perceptron, which gained its popularity with the development of the back-propagation learning algorithm [16]. Despite its simple idea the learning phase still presents a hard nut to crack when hardware implementations of the model are in question.

3.1. Multilayer perceptron

A multilayer perceptron is a feed-forward neural network consisting of a set of source nodes forming the input layer, one or more hidden layers of computation nodes, and an output layer of computation nodes. A computation node or a neuron n in a layer l computes its output as

$$x^l_n = \varphi(v^l_n) \quad \text{with } v^l_n = \sum_i \omega^l_{ni} x^{l-1}_i, \quad (10)$$

where $\varphi(v^l_n)$ is usually some non-linear activation function, and v^l_n is an activation potential given as a scalar product of neuron weights ω^l_{ni} and outputs from the previous layer x^{l-1}_i. According to Eq. (10), the inputs to the model are denoted as x^0_i.

The objective of a learning algorithm is to find such a set of weights that minimizes the performance function, usually defined as a squared error between the calculated outputs and the target values. For the back-propagation learning rule, the weight update equation in its simplest form becomes

$$\omega^l_{ni} = \eta \delta^l_n x^l_i, \quad (11)$$

where η is a learning parameter whilst δ^l_n for the output layer and the hidden layers are given as

$$\delta^l_n = \varphi'(v^l_n)(t_n - x^l_n) \quad \text{and} \quad \delta^l_n = \varphi'(v^l_n) \sum_o \delta^{l+1}_o w^{l+1}_{no}, \quad (12)$$

respectively. In the above equation t_n denotes the n-th element of a target output. For the efficiency of the hardware implementation, we decided to update the weights after presenting each input sample to the model and not to use more advanced update rules.

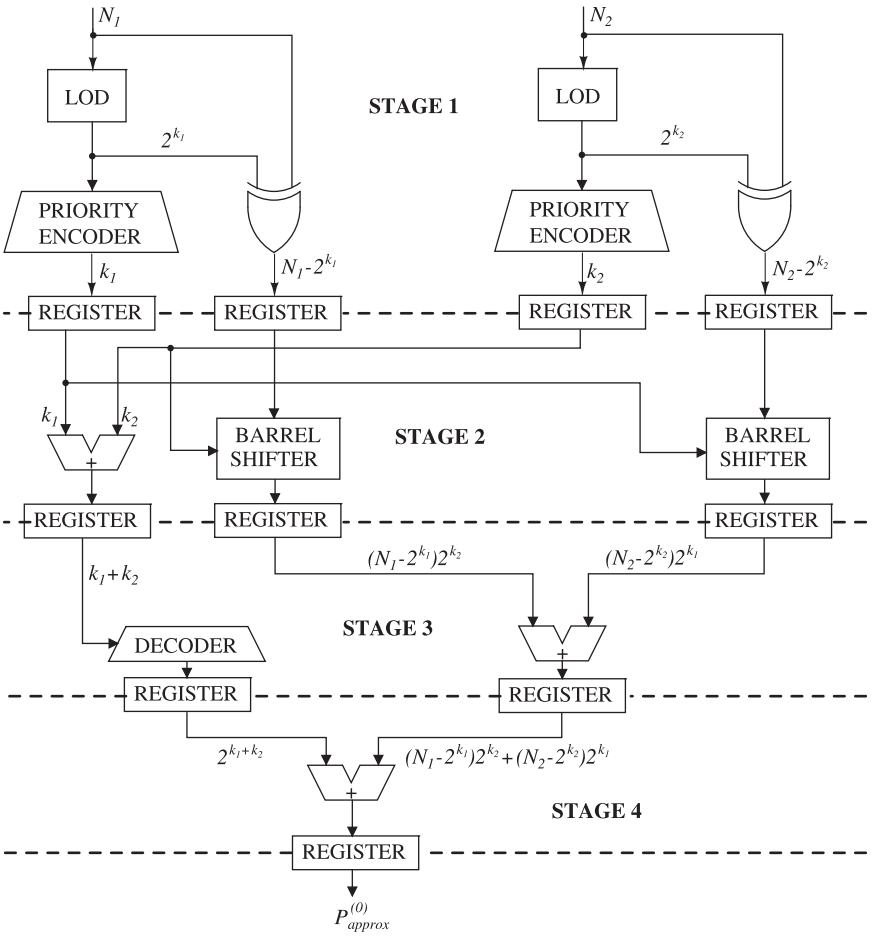

Fig. 2. Block diagram of a pipelined basic block.

Table 2
Device utilization and power consumption of multipliers obtained on the Xilinx Spartan 3 XC3S1500-5FG676 FPGA circuit.

Multiplier	Slices	Lookup	Power (mW)
Iterative logarithmic with one error correction circuit	427	803	10.6
Matrix	477	1137	13.53

3.2. Parallel implementation

A multilayered perceptron exhibits two levels of concurrency: a coarse-grained computation of the outputs from all the neurons in a layer and a fine-grained computation of each neuron's activation potential. However, due to the limited resources, usually only one of the approaches is used in the implementation.

A lot of existing solutions rely on the first concept [2,17,3], where each neuron is treated as a building block composed of a multiplier, an adder, and other accompanying circuits. In this case, the computation of Eqs. (10) and (12) is performed concurrently on all the neurons in a layer, but sequentially inside each neuron. This concept is perfectly suited for processing, while in the learning phase a lot of resources cannot be used simultaneously, for example, the new data should not be fed to the neurons in the first layer until all the weights are updated.

The second concept exploits the similarity between the calculation of the activation potential and the delta of neurons in hidden layers. In both cases, the most complex is the calculation of the scalar product, denoted with the summation in Eqs. (10) and (12). Here, the calculation of the scalar product is done in parallel, while the output of neurons in a layer is obtained in a sequential manner. In contrast to the first concept, here the parallel computation of the scalar product can be used in all the above equations, making this concept especially suitable for the implementation of large hardware neural networks on small FPGA circuits.

We have developed a highly parallel neural unit that calculates the scalar product of two vectors in only one clock cycle [18]. The inputs to the neural units are first passed to the multipliers from which the products are then fed to the adders, organized in a tree-like structure, as shown in Fig. 3. To support the efficient computation of the above equations, the unit has many output ports. Besides the scalar product (port SP), it is designed to calculate the element-wise products (ports EWP) needed for the efficient parallel multiplication in Eqs. (11) and (12), as well as the first level sum (ports FLS) for the parallel calculation of the differences $(t_n - x_n^l)$ in Eq. (12).

If hardware neural network learning is performed off-chip, it is important to calculate the products as well as the activation function very precisely. A lot of solutions for the calculation of the latter can be found in the literature, ranging from a piecewise linear approximation [6], a least-square approximation [8] to an approximate calculation of the exponents [7]. When learning is performed on-chip, larger errors can be tolerated. Moreover, with only one highly parallel neural unit we can afford to hard code the activation function and its derivative in look-up tables (LUTs) with the required precision. The values of the activation function

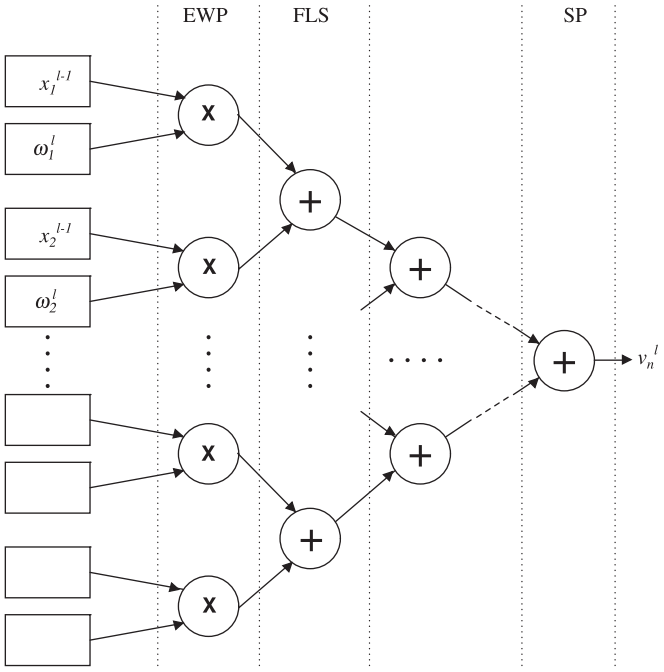

Fig. 3. A highly parallel neural unit [18] (only the computation of Eq. (10) is illustrated).

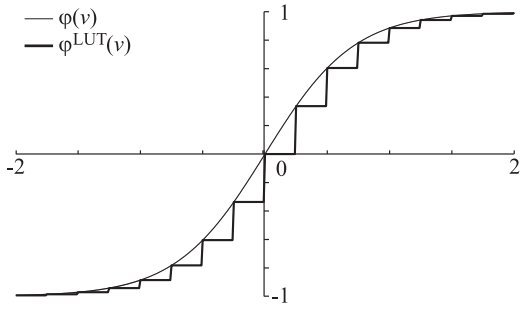

Fig. 4. Activation function $\varphi(v)=\tanh(1.4v)$ and its LUT approximation with $b=2^4$ elements in interval $[-2, +2]$.

defined by a LUT with b elements in an interval $[-r,+r]$, taking into account only the quantization of the activation potential, are obtained from the equation

$$\varphi^{\mathrm{LUT}}(v)=\varphi(\lfloor v(2r/b)^{-1}\rfloor(2r/b)). \tag{13}$$

The effect of the proposed quantization is presented in Fig. 4.

To use the neural unit (NU), a set of subsidiary units is needed: a RAM memory for storing weights, registers for keeping the inputs, outputs, and partial results, multiplexers (MUX) for loading the proper data to the neural unit, lookup tables with stored values of the activation function (LUT) and its derivative (LUTd) and three state machines. The forward pass and the backward pass are controlled by the Learn and Execute state machines, respectively, which are supervised by the Main state machine. A simplified scheme of the implementation is shown in Fig. 5, where data-paths used for processing are denoted with the thick black lines, additional data-paths needed during learning with the thick gray lines, and control signals with the thin black lines.

In order to gain as much as possible from the neural unit, it should be capable of calculating a scalar product of the largest vectors that appear in the computation. The hardware circuit thus becomes very complex and can only be operated at lowered frequencies. For example, a unit with 32 18-bit multipliers and consequently 31 18-bit adders in a tree-like structure was implemented in the Spartan 3 XC3S1500-5FG676 FPGA chip. While separate multiplications can run at a maximum frequency of 50 MHz, the proposed unit managed to run at a still acceptable 30 MHz [18].

4. Experimental work

To assess the performance of the iterative logarithmic multiplier, a set of experiments was performed on multilayer perceptron neural networks with one hidden layer. The models were compared in terms of the classification or approximation accuracy, the speed of convergence, and the power consumption. Three types of models were evaluated: (a) an ordinary software model (SM) using floating-point arithmetic, (b) a hardware model with exact matrix multipliers ($\mathrm{HM_M}$), and (c) the proposed hardware model using the iterative logarithmic multipliers with one error-correction circuit ($\mathrm{HM_L}$).

The models were evaluated on the PROBEN1 collection of freely available benchmarking problems for the neural network learning [19]. A rather heterogeneous collection contains 15 datasets from 12 different domains, and all but one consist of real-world data. Among them 11 datasets are from the area of pattern classification and the remaining four are from the area of function approximation. The datasets, containing from a few hundred to a few thousand input–output samples, have been already divided into training, validation and test sets, generally in the proportion 50:25:25. The number of attributes in the input samples ranges from 9 to 125 and in output samples from 1 to 19. Before modelling, all the input and output samples were rescaled to the interval $[-0.8, +0.8]$.

4.1. Setup

The testing of the models on each of the datasets mentioned above was performed in two steps. After finding the best software models, the modelling of the hardware models started, keeping the same number of neurons in the hidden layer. During the optimization of the software models, the number of neurons in the hidden layer was varied in such a way that the number of model weights did not exceed the number of training samples, where the number of inputs and outputs is determined by a dataset. The model topology with respect to the dataset is given in Table 3. Due to the heterogeneity of the datasets, values of the learning parameter η, ranging from 2^{-2} to 2^{-12}, were used. They are expressed in powers of two in order to replace the first multiplication in Eq. (11) with a simple shift circuit. For both hardware models the weights were limited to the interval $[-4, +4]$. The processing values, including the inputs and outputs, were represented with 16 bits in the interval $[-1, +1]$. The values of the activation function $\varphi(v)=\tanh(1.4\,v)$ and its derivatives for $b=2^8$ equidistant values of v from the interval $[-2, 2]$ were stored in two separate lookup tables.

By applying the early-stopping criterion, the learning was stopped as soon as the classification or approximation error on the validation set started to grow. The model parameters that gave the best performance on the validation set were further used to assess the performance of the models on the test set, consisting only of the samples that were not used during the learning phase.

4.2. Weight precision

The impact of weight precision on the model performance was studied in terms of the normalized squared error, defined as

$$E=\left\langle \frac{\langle (t_n-x_n^2)^2\rangle_s}{(\max_s t_n-\min_s t_n)^2}\right\rangle_n, \tag{14}$$

where t_n denotes the n-th element of the target, x_n^2 denotes the n-th output from the second (output) layer. In the above equation

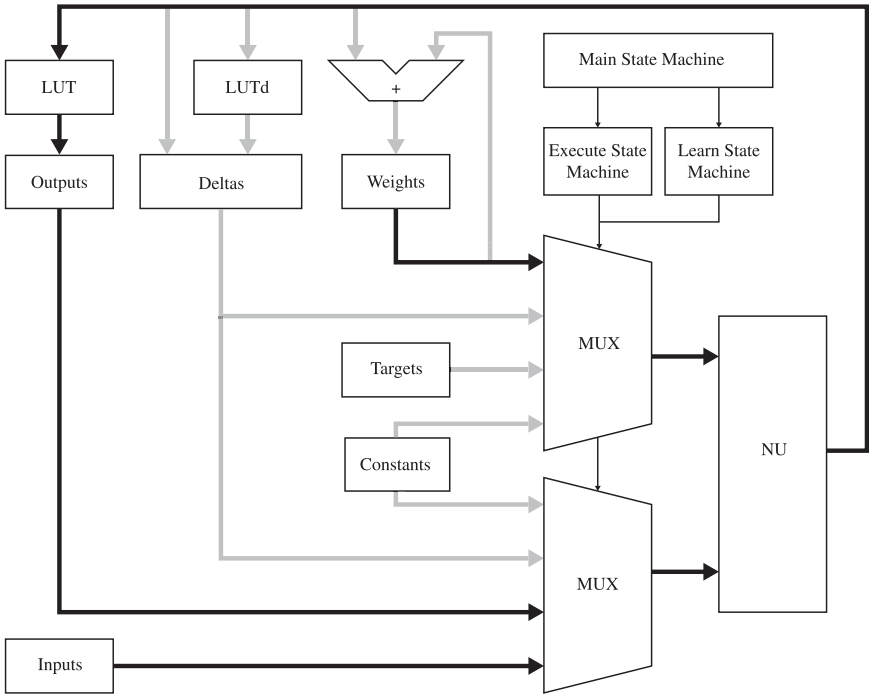

Fig. 5. Neural network implementation scheme [18].

Table 3
Properties of datasets and corresponding model topology given in terms of inputs–hidden neurons–outputs.

Dataset	Samples	Topology	Free params.
cancer1	699	9–6–2	74
card1	690	51–6–2	326
diabetes1	768	8–7–2	79
gene1	3175	120–8–3	995
glass1	214	9–12–6	198
heart1	920	35–8–2	306
heartc1	303	35–8–2	306
horse1	364	58–12–3	747
mushroom1	8124	125–39–2	4994
soybean1	683	35–22–19	1229
thyroid1	7200	21–48–3	1203
building1	4208	14–56–3	1011
flare1	1066	24–4–3	115
hearta1	920	35–3–1	112
heartac1	303	35–4–1	149

$\langle\cdot\rangle_s$ and $\langle\cdot\rangle_n$ denote averaging over all the samples and output attributes, respectively, whilst $\min_s$ and $\max_s$ denote the minimal and maximal values among all the samples.

As presented in Fig. 6 for Hearta1 dataset, the normalized squared error exhibits a typical exponential decrease for an increasing precision of the weights. However, the increasing precision of the weights also requires more and more hardware resources. Since there is a big drop in the normalized squared error when the precision is increased from 16 to 18 bits, and since we can make use of numerous prefabricated 18×18-bit matrix multipliers in the new Xilinx FPGA programmable circuits, our further analysis is confined to an 18-bit weight precision.

4.3. Classification problems

The model performance for the first 11 datasets in Table 3 is given in Fig. 7. The average values and standard deviations for all types of models over 10 runs are given in terms of three

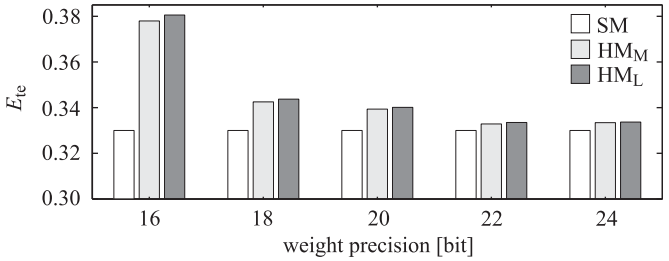

Fig. 6. Performance of the models with respect to the weight precision on Hearta1 dataset.

measures: the number of epochs, the normalized squared error E_{te} and the percentage of misclassified samples p_{te}^{miss}. For each dataset the results obtained with the models SM, HM_M, and HM_L are presented with white, gray and black bars, respectively.

The results obtained for the software models using the back-propagation algorithm are similar to those reported in [19], where more advanced learning techniques were applied. The most noticeable difference between the software and hardware models is in the number of epochs needed to train a model. The number of epochs in the case of the hardware models is for many datasets an order of magnitude smaller than in the case of the software models. The reason probably lies in the inability of the hardware models to further optimize the weights due to their representation in limited precision.

As a rule, the hardware models exhibit slightly poorer performance in the case of the normalized squared error and the percentage of misclassified samples. The discrepancy is very large for the gene1 and thyroid1 datasets, where, apparently, more than 18 bits representation of the weights is needed to close the gap.

4.4. Approximation problems

The last four datasets in Table 3 are from the approximation domain, so their performance was assessed only in terms of the

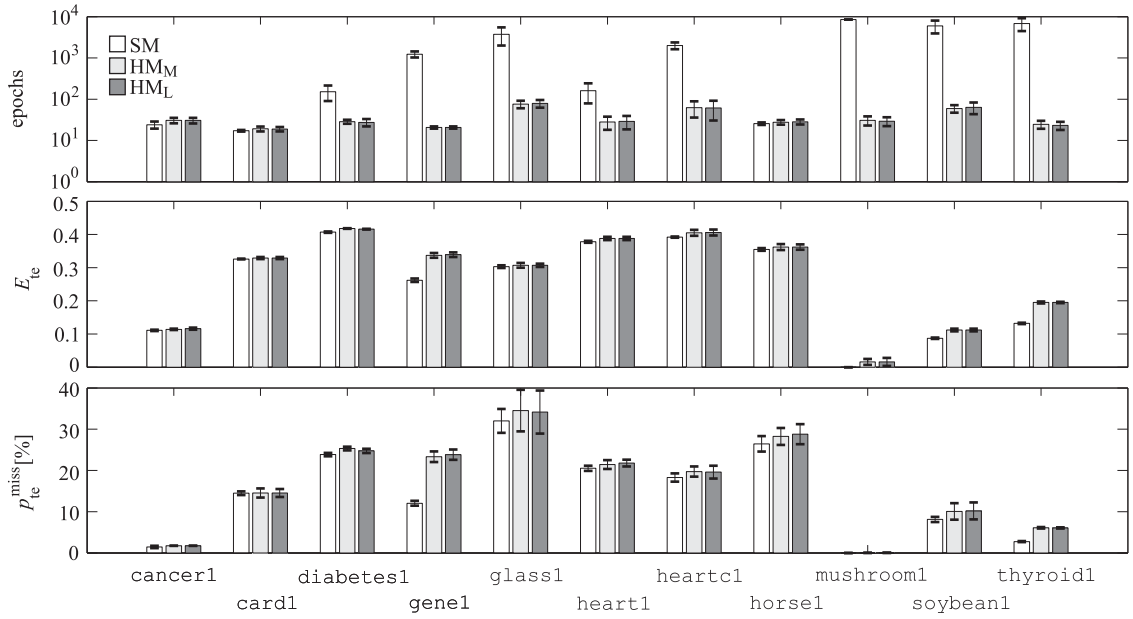

Fig. 7. Performance of the models in terms of the number of epochs (top), the normalized squared error (middle) and percentage of misclassified samples (bottom). Average values over 10 runs as well as standard deviations are given.

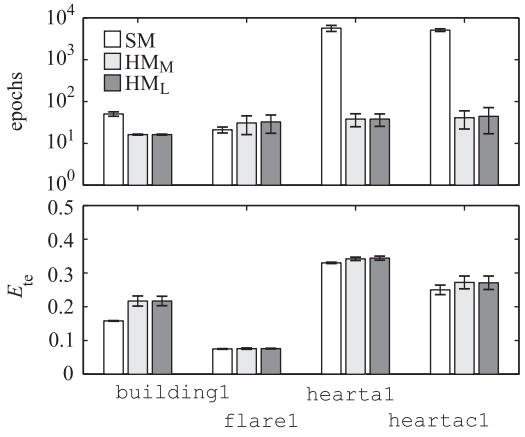

Fig. 8. Performance of the models in terms of the number of epochs (top) and the normalized squared error (bottom). Average values over 10 runs and standard deviations are given.

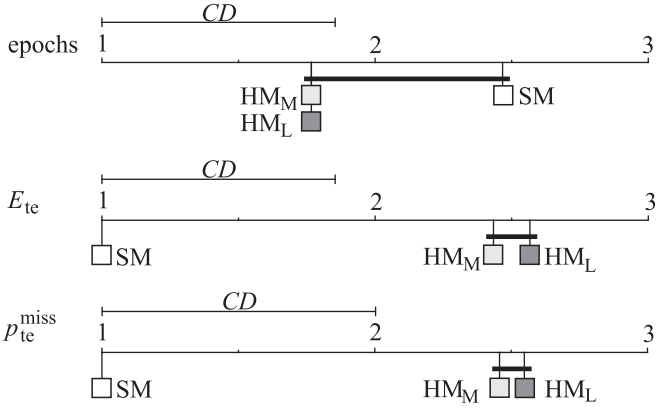

Fig. 9. Comparison of models in terms of the number of epochs (top), normalized squared error (middle) and percentage of misclassified samples (bottom). Models that are not significantly different are connected with a thick line.

number of epochs and the normalized squared error E_{te}. In Fig. 8 the same color coding is used as in the classification problems.

Similar conclusions as in the case of the classification problems can be drawn, as follows. Due to the limited precision of the weights, the learning process in the hardware models stops earlier, and the hardware models exhibit slightly poorer performance in terms of the normalized squared error.

4.5. Statistical evaluation

Using the statistical tests recommended by [20], we determined the statistical significance of the results. For the comparison of the three models, the Friedman non-parametric test was applied. The analysis is based on model ranks, which are separately determined for each dataset. Rank 1 is assigned to the best model, rank 2 to the second best and so on. In the case of ties, the average ranks are calculated in order to keep the sum of the ranks constant. According to the Nemenyi test, the performance of the two models is significantly different if the corresponding average ranks differ by a critical difference. The critical distance for three models and the confidence level 0.05 is given as $CD = 3.314S^{-1/2}$, with S being the number of datasets used in the analysis.

The analysis is visually presented in Fig. 9, where the average ranks of the three models are given in terms of the number of epochs, E_{te} and p_{te}^{miss}. While the analysis of the first two measures is made on 15 datasets, the analysis of p_{te}^{miss} is performed only on the 11 datasets for the classification problems. The models, for which average ranks differ by less than the critical difference CD, are connected with a thick line to stress that their differences are not statistically relevant. As we already observed, the limited bit precision of the weights means that the hardware models take fewer epochs to train, but the difference is not significant. As expected, owing to the same speculation the software models significantly outperform the hardware models in terms of the measures E_{te} and p_{te}^{miss}. Most importantly, the comparison of the hardware models HM_M and HM_L reveals that the replacement of the exact matrix multipliers with the proposed approximate iterative logarithmic multipliers does not have any significant effect on the performance of the models. The reason for the very good compensation of the errors caused by an inexact multiplication can be found

Table 4
Estimation of FPGA device utilization for a neural network model with 32 inputs, 8 hidden neurons and 10 outputs using 16×18-bit matrix multipliers.

Part of the circuit	Slices ($\times 1000$)	Lookup tables ($\times 1000$)
Whole model	27	38
Neural unit	25 (92%)	34 (89%)
Multipliers	24 (89%)	33 (87%)

in the excellent ability to adapt, common to all neural network models.

4.6. Device utilization

The proposed neural unit needs to be applied many times to calculate the model output; therefore, it is important to be as small and as efficient as possible. The estimation of the device utilization in terms of the Xilinx Spartan 3 FPGA programmable circuit building blocks for a model with 32 exact 16×18 matrix multipliers is shown in Table 4. According to the analysis of the multipliers in Table 2, the replacement of the matrix multipliers with the iterative logarithmic multipliers can lead to more than 10% smaller device utilization and more than 20% smaller power consumption.

5. Conclusion

Neural networks offer a high degree of internal parallelism, which can be efficiently used in custom chip designs. Our work has been focused on the efficient digital design of a hardware neural network using field-programmable gate-array technology. The work was aimed at the design of a resource-, speed- and power-consumption efficient, feed-forward neural network with on-chip learning ability.

Neural network processing comprises a huge number of multiplications. To gain as much as possible from the custom design, multiplications must be performed in parallel. However, multiplication circuits consume a lot of resources, time and power. Since the resources on a chip are limited, different strategies are applied to overcome the limitations. The first idea is to replace the floating-point arithmetic with fixed-point arithmetic. However, to further increase the performance the exact fixed-point matrix multipliers must be replaced with some approximate solutions.

The hardware neural network presented in this paper is built around an iterative logarithmic multiplier, which can use many levels of correction circuits to iteratively approximate a product to the arbitrary precision. It also enables the pipelined design of correction circuits, which significantly reduce the propagation time of a signal through a circuit. The iterative logarithmic multiplier with only one correction circuit is enough to reduce the multiplication error, on average, to less than 1%.

The proposed logarithmic multiplier needs fewer resources and consequently leads to designs with more concurrent units on the same chip. In contrast to the majority of the proposed designs, where a special hardware unit is used for each neuron, our design contains only one highly parallel neural unit, which is capable of the fast parallel calculation of a neuron output. Since the same circuit can be used in forward and backward passes, it is more suitable for hardware neural network designs targeting small FPGA chips.

The performance of the proposed hardware neural network with iterative logarithmic multipliers was compared to the usual software models and hardware neural network with exact matrix multipliers. The models were tested on the PROBEN1 benchmark dataset, consisting of classification and approximation problems. Although the training of the software models, on average, takes longer, the difference is not statistically significant. More encouraging is the fact that in terms of the observed measures, i.e., the number of training epochs, the normalized squared error, and the percentage of misclassified samples, there is no statistically significant difference in the performance of both hardware models.

Due to the highly adaptive nature of neural network models, which compensated the erroneous calculation, the replacement of the multipliers did not have any notable impact on the models' processing and learning accuracy. Furthermore, the consumption of fewer resources per multiplier also results in more power-efficient circuits. The power consumption, which was reduced by roughly 20%, makes the hardware neural network models with iterative logarithmic multipliers favorable candidates for battery-powered applications.

Acknowledgments

This research was supported by the Slovenian Research Agency under Grants P2-0241 and P2-0359, and by the Slovenian Research Agency and the Ministry of Civil Affairs, Bosnia and Herzegovina, under Grant BI-BA/10-11-026.

References

[1] F.M. Dias, A. Antunes, A.M. Mota, Artificial neural networks: a review of commercial hardware, Eng. Appl. Artif. Intell. 17 (2004) 945–952.
[2] J. Misra, I. Saha, Artificial neural networks in hardware: a survey of two decades of progress, Neurocomputing 74 (2010) 239–255. Artificial brains.
[3] J. Bailey, R. Wilcock, P. Wilson, J. Chad, Behavioral simulation and synthesis of biological neuron systems using synthesizable VHDL, Neurocomputing 74 (2011) 2392–2406.
[4] J. Zhu, P. Sutton, FPGA implementations of neural networks—a survey of a decade of progress, in: P.Y.K. Cheung, G.A. Constantinides, J.T. de Sousa (Eds.), FPL, Lecture Notes in Computer Science, vol. 2778, Springer, 2003, pp. 1062–1066.
[5] A. Armato, L. Fanucci, E. Scilingo, D.D. Rossi, Low-error digital hardware implementation of artificial neuron activation functions and their derivative, Microprocessors Microsystems Embedded Hardware Design 35 (2011) 557–567.
[6] P. Ferreira, P. Ribeiro, A. Antunes, F.M. Dias, A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function, Neurocomputing 71 (2007) 71–77. (Dedicated Hardware Architectures for Intelligent Systems; Advances on Neural Networks for Speech and Audio Processing).
[7] M. Skrbek, Fast neural network implementation, Neural Network World 5 (1999) 375–391.
[8] N. Nedjah, R. da Silva, L. Mourelle, M. da Silva, Dynamic MAC-based architecture of artificial neural networks suitable for hardware implementation on FPGAs, Neurocomputing 72 (2009) 2171–2179. (Lattice Computing and Natural Computing (JCIS 2007)/Neural Networks in Intelligent Systems Design (ISDA 2007)).
[9] U. Lotrič, P. Bulić, Logarithmic multiplier in hardware implementation of neural networks, in: A. Dobnikar, U. Lotric, B. Ster (Eds.), ICANNGA (1), Lecture Notes in Computer Science, vol. 6593, Springer, 2011, pp. 158–168.
[10] D. Anguita, A. Ghio, S. Pischiutta, S. Ridella, A support vector machine with integer parameters, Neurocomputing 72 (2008) 480–489. (Machine Learning for Signal Processing (MLSP 2006)/Life System Modelling, Simulation, and Bio-inspired Computing (LSMS 2007)).
[11] J. Holt, J.-N. Hwang, Finite precision error analysis of neural network hardware implementations, IEEE Trans. Comput. 42 (1993) 281–290.
[12] J.N. Mitchell, Computer multiplication and division using binary logarithms, IRE Trans. Electron. Comput. EC-11 (1962) 512–517.
[13] V. Mahalingam, N. Ranganathan, Improving accuracy in Mitchell's logarithmic multiplication using operand decomposition, IEEE Trans. Comput. 55 (2006) 1523–1535.
[14] N. Petra, D. De Caro, V. Garofalo, E. Napoli, A. Strollo, Truncated binary multipliers with variable correction and minimum mean square error, IEEE Trans. Circuits Syst. I Regular Pap. 57 (2010) 1312–1325.
[15] Z. Babić, A. Avramović, P. Bulić, An iterative logarithmic multiplier, Microprocessors Microsystems Embedded Hardware Design 35 (2011) 23–33.
[16] S. Haykin, Neural Networks: A Comprehensive Foundation, second ed., Prentice Hall, 1998.

[17] V. Pedroni, Circuit Design with VHDL, The MIT Press, 2004.
[18] M. Gutman, U. Lotrič, Implementation of neural network with learning ability using FPGA programmable circuit, in: B. Zajc, A. Trost (Eds.), Nineteenth International Electrotechnical and Computer Science Conference, ERK 2010, vol. B, IEEE Slovenian Section, pp. 173–176.
[19] L. Prechelt, Proben1—a set of neural network benchmark problems and benchmarking rules, 1994.
[20] J. Demšar, Statistical comparisons of classifiers over multiple data sets, J. Mach. Learn. Res. 7 (2006) 1–30.

Uroš Lotrič received the B.Sc. degree in physics, and the M.Sc. and Ph.D. degrees in computer science from the University of Ljubljana, Ljubljana, Slovenia, in 1994, 1997, and 2000, respectively. He is currently an Associate Professor with the Faculty of Computer and Information Science, University of Ljubljana. His research interests include soft-computing methods, distributed processing, and their applications.

Patricio Bulić received his B.Sc. degree in electrical engineering, and M.Sc. and Ph.D. degrees in computer science from the University of Ljubljana, Slovenia, in 1998, 2001 and 2004, respectively. He is an Associate Professor at the Faculty of Computer and Information Science, University of Ljubljana. His main research interests include computer architecture, digital design, parallel processing and vectorization techniques. He is a member of the IEEE Computer Society and ACM.