An Embedded System for Handwritten Digit Recognition

Luca B. Saldanha^{a,*}, Christophe Bobda^a

^a*CSCE Department, University of Arkansas, Fayetteville, AR*

Abstract

The goal of this work is the design and implementation of a low-cost system-on-FPGA for handwritten digit recognition, based on a relatively deep and wide network of perceptrons. In order to increase the performance of the application on embedded processors whose performances are way below standard general purpose CPUs, a regularization method was used during the training phase of the neural network that allows for the drastic reduction of floating point operations. Our implementation can achieve a $3\times$ speed-up toward a raw implementation without optimization, while keeping the accuracy in acceptable ranges. Our efforts reinforce the fact that FPGAs are suited for deploying complex artificial intelligence modules.

Keywords: neural network, soft processor, regularization, image processing;

1. Introduction

Artificial intelligence is in the verge of becoming ubiquitous in our society. A considerable amount of money made by Internet companies comes from systems based on machine learning and user data mining for recommending movies, personalized advertisement and even match making. Soon autonomous intelligent systems will share our world. Some already do, especially to substitute humans on high risk activities. But these are very sophisticated and expensive systems.

*Corresponding author

Email address: lbochisa@uark.edu (Luca B. Saldanha)

URL: <http://www.csce.uark.edu/~lbochisa/> (Luca B. Saldanha)

The barriers imposed by several constraints are still to be broken: size, weight, power consumption (SWAP) and cost, for example.

10 Embedded systems using reconfigurable devices appear as a possible solution. Even though the most complex problems in the artificial intelligence field still require supercomputer-like performance, FPGAs are already an alternative for less complicated tasks. In this work we demonstrate that by applying field specific knowledge and by exploiting the full range of capabilities that an
15 FPGA has to offer, it is possible to use them to implement practical systems that rely on relatively deep and wide neural networks, such as handwritten digit recognition.

In 2011, Gomperts and Ukil, [1], already devised a VLSI implementation of an artificial neural network. They stressed that this unconventional computational model presents complications such as a high hardware cost and a large
20 number of arithmetic operations (especially floating or fixed point). Dinu et. al. ([2]) presented an algorithm for hardware implementation of neural networks, but with a simplified activation function that allows for the simplification of the calculations due to its boolean properties. Another successful and low cost
25 implementation of a learning structure (support vector machines) in FPGA can be found at [3].

All these hardware models still require a great engineering effort, in the sense that for every specific problem, the designer must be aware of a great number of details that are not always clear at a high level abstraction. We advocate
30 a simple approach. Since FPGA logic is inadequate for complex floating point operations, we are targeting embedded soft processors, such as the MicroBlaze ([4]), or, more recently introduced hard coded processing cores embedded in the FPGA fabric ([5] for example). We propose to circumvent the use of floating point operations through a combination of code re-organization and dedicated
35 hardware accelerators. The hardware blocks are used to treat the input data, making useful transformations to conveniently feed the neural network. Further explanation on the system architecture will be provided in sections 2 and 5.

Crunching the input data has a clear positive effect when it comes to improv-

ing the accuracy of the system, but it does not directly augment its performance.

40 The effect is rather incidental. As we will demonstrate in subsequent sections, excluding non-relevant sections of the image in the case of digit recognition (e. g. the background), will significantly reduce the number of arithmetic operations in the neural network. This fact alone, however, is not enough. Further improvements on the training phase of the neural network shall be performed.

45 An established concept in computational models dictates that, with very high probability, the best model is not more complex than the data suggests. This line of thought is referred to as the Occam’s razor ([6]). Several methods are applied to the training process of neural networks to take advantage of this property ([7]). L1 and L2 regularization are an example, as demonstrated in
50 [8], [9]. They exploit the hypothesis that some neural network connections can be discarded and promote simplicity in the structure (the case of L1 regularization) or the hypothesis that weaker connections (small-scaled weights) have a similar simplifying effect (L2 regularization). In the remaining of the paper we demonstrate that L1 regularization has an interesting side effect that can be
55 applied to our problem. By discarding the majority of the neuron’s connections, the neural network becomes feasible in very simple embedded processors, with low clock frequency and extremely low cost.

By exploiting this side effect of L1 regularization, we were able to design and implement a handwritten digit recognition system relying on a neural network
60 predictor trained using this method. It can achieve a 3x speed-up compared to the same system trained without any optimization, making it possible to be executed in a simple soft processor.

The rest of the paper is organized as follows. Section 2 will give the reader an overview of the proposed architecture. Then, section 3 will lay the mathematical
65 ground that is fundamental for the understanding of our approach. Next, the training process will be explained in section 4 and every hardware block will be briefly described (as well as the software design approach) in section 5. Finally the results will follow (section 6) as long as a brief conclusion (section 7).

Figure 1: System overview.

2. System Overview

70 The system can be described overall as a hardware/software solution for the specific application of handwritten digit recognition. Hardware accelerators are used for the image processing pipeline, liberating the software of the heavy burden of sequentially dealing with image pixels. Another advantage is to significantly reduce the exchange of data with external memories, a frequent
75 bottleneck for SoCs performance. Figure 1 depicts a high level schematic of the whole system. Every block was written in VHDL, except for the MicroBlaze processor and DDR3 controller, which were generated using tools from the FPGA manufacturer.

The input data must be treated properly in order to comply with existing
80 datasets used for off-line training (see section 4). There is nothing special in our implementation for the image filters, except, perhaps, for the proper choice of operations and their order. The image is captured with a regular digital camera and the blocks *Camera Interface* and *Bayer to Grayscale* treat the input pixels to the desired grayscale format. Then, a thresholding operation takes place to
85 discard the irrelevant background and random noise that might have occurred. Finally, we apply a series of Gaussian filters followed by subsampling, in order to reduce the size of the image to the desired amount of pixels. All relevant

details of these hardware blocks can be checked at subsection 5.1.

Hardware blocks for convolution and thresholding operations usually exhibit
90 real time behaviour while keeping the resource usage to a minimum. On the
other hand, floating point operations using dedicated DSP blocks (e. g. Xilinx’s
DSP48A1) are commonly not directly supported. Even custom made blocks tai-
lored specifically for a given task require a significant portion of FPGA logic.
Therefore, implementations of neural network based predictors, with a high
95 number of neurons, are not suited for parallel processing, making a sequential
processor with floating point operations capabilities fundamental. The *MicroB-
laze* processor is used to deploy the neural network based predictor, its output
is used by the *Drawing Module* to render the predicted digit onto the image (for
debugging purposes) and finally, after passing through a DDR3 based frame
100 buffer, the result is shown in a TFT display.

This is a typical example of the importance of hardware and software mixed
solutions. The central task for the application can not be implemented in hard-
ware blocks (it would not fit in the device) and it is too slow to execute in
low-performance embedded processors. The former obstacle can not be over-
105 come, at least without a great engineering effort, but the second can be tackled
using software optimization methods and by understanding the elemental prop-
erties of the problem at hand. The next section will go deep into the subject
and a possible solution will be demonstrated.

3. Neural Networks

110 The implementation of a relatively complex neural network on a cheap, small
and low-power FPGA using a standard soft-processor as the main processing
unit is the fundamental contribution of this work. Thus, it is mandatory to
deepen our knowledge in this matter. This section will briefly guide the reader
through the basics of neural networks based on standard, non-linear, continu-
115 ous activation functions. We will then go over the training process based on
gradient descent and backpropagation. Finally, we will show that by using an

Figure 2: The perceptron model.

L1 regularization method we can drastically reduce the number of weights of the neural network, making it possible to be implemented in a low-frequency embedded processor in real time.

120 3.1. Basics

Neural networks have been studied for a long time. They date back to at least 1943 [10] but the principles involved can be traced to early works on linear regression methods in the 1800s as noted by [11]. A great number of neural network models was developed ever since, therefore it is important to emphasize that in this work we are particularly interested in multi layer perceptrons (MLP) [12]. The term *neural network* will refer solely to *multi layer perceptron* throughout the remainder of the text.

A perceptron is basically a biologically inspired model of a neuron. It is excited by other neurons (or by external signals) and triggers an output based on the sum of its inputs after being scaled by weights that model how strong two perceptrons are connected. This transformation is called the activation function. Figure 2 represents a perceptron and its functionality is described by equation 1. $\mathbf{W}$ denotes the weight vector (W_1, W_2) , $\mathbf{x}$ the input vector $(x_1, x_2)^T$ and b is the neuron's bias.

$$\hat{y} = a(\mathbf{W}\mathbf{x} + b) \quad (1)$$

When organized in layers, one layer feeding into the next, multiple perceptrons form a neural network. By appropriate choosing the activation function, this

130 structure becomes a high dimensional function approximator. By adjusting the
various weights of the network, a multi-dimensional manifold can be twisted
through space to closely fit the function that the data represents. If it is deep and
large enough, an MLP becomes a powerful tool, capable of representing complex
relationships between input data and their meaning, finding correlations that
135 even experienced humans cannot perceive.

3.2. Gradient Descent

As mentioned before, if we intelligently tweak the weight and bias values
we can represent any function in any dimensional space, given that our neural
network has enough layers and each layer has enough neurons. The problem with
140 this incredibly powerful model is that the function we are trying to delineate is
obviously unknown, otherwise we could simply apply it through regular math.
Therefore, a method to learn the function must be devised. In this work, we
are interested in supervised learning, i.e., a finite set of input data (features) and
their respective expected output (labels) is available and we could use it to infer
145 the parameters of the neural network performing some optimization method.
This process is called training.

Although the function we want to learn is unknown, it is possible to evaluate
how close the approximation is using the known data. Equation 2 measures the
fitness of the neural network at the current moment. It expresses the sum of
the squared error over the entire training set (size N) considering the M output
dimensions. y_{ij} is the true output value and $\hat{y}_{ij}$ is the output of the MLP (i. e.
the result of equation 1 applied by the output neurons).

$$SSE = \sum_{i=1}^N \sum_{j=1}^M (y_{ij} - \hat{y}_{ij})^2 \quad (2)$$

Using fitness estimations such as the SSE, it is possible to modify the weights
and bias and evaluate the change. There is a great variety of optimization
methods available to perform this task: genetic algorithms, simulated annealing,
150 hill climbing, among others. Although every one has some advantages, the

research field of machine learning steers towards one single method and its variations, for its effectiveness and scalability - gradient descent.

Gradient descent is based on analytically calculating the direction to where the error function decreases the most, i. e. the negative of the derivative of the error function in relation to the weights. The derivation of the gradient is easier if we consider, at first, a neural network with only one layer. Equation 3 expresses this derivative at the j -th output dimension in relation with the weight connecting the k -th input to the neuron, whereas equation 4 is its expansion when $\hat{y}_{ij}$ is expanded and the chain rule is applied (for a thorough explanation on the chain rule, please see [13]).

$$\frac{\partial SSE}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \sum_{i=1}^N (y_{ij} - \hat{y}_{ij})^2 \quad (3)$$

$$\frac{\partial SSE}{\partial w_{jk}} = -2 \sum_{i=1}^N (y_{ij} - a(\mathbf{W}_j \mathbf{x}_i + b)) a'(\mathbf{W}_j \mathbf{x}_i + b) x_i \quad (4)$$

Every term of equation 4 can be evaluated as long as the activation function is differentiable. Then, after calculating the gradient, every weight can be updated (and the bias, although the rule for bias update was omitted, since it can easily be found following the same procedure) by taking a step in the opposite direction (scaled by a relatively small learning rate η , since the aforementioned calculation is true only locally).

$$w_{jk} \leftarrow w_{jk} - \eta \frac{\partial SSE}{\partial w_{jk}} \quad (5)$$

There is an ongoing discussion regarding the update rule using gradient descent. Some researchers advocate that we must evaluate the *true* gradient before applying equation 5 ([14], [15]), as equation 4 depicts. Others ([16] for example) claim that we can apply what is called *stochastic gradient descent*. This means that there is no need to sum the derivative over the entire dataset, the update can be applied at every instance presented to the network. Therefore, equations 4 and 5 become:

$$\frac{\partial SE}{\partial w_{jk}} = -2(y_j - a(\mathbf{W}_j \mathbf{x} + b)) a'(\mathbf{W}_j \mathbf{x} + b) x \quad (6)$$

$$w_{jk} \leftarrow w_{jk} - \eta \frac{\partial SE}{\partial w_{jk}} \quad (7)$$

The rest of the paper will refer solely to *stochastic gradient descent* as the optimization method.

155 Since this analysis is true only for a neural network with a single layer, the gradient related to any weight when other layers are added can be easily calculated using a method called *backpropagation*. It refers to the possibility of iteratively discover how much of the current squared error is due to each neuron. It is not in the scope of this work to derive backpropagation rules, we must only
160 emphasize that it follows the strict calculus analysis of the gradient and it is an extremely cheap method to implement.

For the sake of completeness, we must mention that the activation function used in this work was the *tanh* (hyperbolic tangent). This function has several favorable properties: it is monotonic, evaluates to 0 at the origin, approximates
165 the identity function when close to the origin and it is differentiable ($\tanh'(x) = 1 - \tanh^2(x)$).

3.3. L1 Regularization

Perhaps the most crucial problem faced by neural networks training procedure is *overfitting*. It refers to approximating the training data too closely, failing to generalize to unseen data points. One way of avoiding overfitting is weight regularization. This basically means not allowing the weights to become too large, favoring less complex solutions. The regularization method of choice for this work was L1 regularization. It is derived by adding a penalty term to equation 2 equal to the sum of the absolute values of all the weights in the network (scaled by some factor). Thus, larger weights will produce, artificially, a larger squared error and the update rule will automatically correct the weight to become smaller (unless the true squared error points too strongly in the other

Figure 3: Random sample of the MNIST dataset.

direction). Equation 8 shows the new update rule:

$$w = \begin{cases} w' - \delta & \text{if } w'' \geq 0 \\ w' + \delta & \text{if } w'' < 0 \end{cases} \quad (8)$$

w' refers to the weight after being normally updated using backpropagation, w'' refers to the original weight before the update and δ is the empirically chosen scaling factor.

L1 regularization has the extra effect of reducing to near zero all the weights that do not contribute significantly to the neural network. So, by trimming the weights whose absolute value is below some threshold, we can drastically reduce the complexity of the MLP by avoiding the majority of the floating point operations that are the greatest burden for embedded processors. The next section will show preliminary results regarding this aspect of L1 regularization.

4. Training on the MNIST Dataset

The MNIST dataset ([17]) is composed by a training set of 60000 and a test set of 10000 handwritten digits (each has 28×28 grayscale pixels, each being represented by 8 bits). Figure 3 shows a random sample of this dataset. The neural network used in this work is composed of 3 layers. The input layer has 300 neurons, each neuron receives as input the 784 pixels of the handwritten digit. The subsequent layers have 100 and 10 neurons, respectively. The 10 output neurons will be activated proportionally to the network prediction, i. e., if the

Figure 4: Training evolution.

185 network predicts a 0, the first output perceptron will evaluate to a number near 1, depending on its confidence, and the other neurons will evaluate to positive values near 0 or to negative values. The same is true for the prediction of a 1 (the second neuron will output a high value and the others low values), and so on.

190 This MLP was trained with stochastic gradient descent and backpropagation on the 60000 instances of the training set presented, obviously, one at a time in a random order. The period in which all training instances are presented to the network exactly one time will be referred as an epoch (the training consists of several epochs). All input values were normalized by a division by 256. It is arguable whether the input normalization has any benefit, but we must emphasize that the activation function is capable of representing numbers between -1.0 and 1.0 , thus making all involved values fall inside this interval is intuitively a good idea. It is important to remember that the final deployment of this system will be in a simple embedded processor that may have limitations in its floating point representations. When $\tanh(x)$ is applied to relatively high numbers, a small representation error can significantly change the output, since this function saturates very easily.

205 The learning rate was set to 0.01 and it decreases by a factor of 0.997 after every epoch of training. The weights and bias were initialized to small random values picked from a normal distribution with mean equal to 0 and deviation equal to $\max(0.3, 1.0/N)$, where N denotes the number of inputs of the given

Figure 5: Accuracy reduction related to the number of weights.

perceptron. The L1 regularization decay was set to 0.0001. Figure 4 shows the number of misses over the 10000 test instances after each epoch.

The best result was obtained after 79 epochs (335/10000 misses, which took
 210 approximately 3.5 hours on a standard workstation (Intel Core 2 Duo CPU @
 3 GHz). Of course, a missing rate of 3.35% is far from the state-of-the art,
 but for the purposes of this work it is perfectly acceptable. Considering the
 best results achieved for digit recognition using this database, all of them make
 use of powerful workstations and many hours of training, therefore, a trade-off
 215 between quality of results and required computational power must exist.

As depicted in figure 5, we can take advantage of L1 regularization to reduce
 the number of weights. The elimination of weights (the number of weights is
 shown in the right vertical axis) whose absolute value is above some threshold
 (horizontal axis) is drastic while the accuracy (shown in the left vertical axis) is
 220 kept, to a certain point, at an acceptable range. The original number of weights
 is 266200 and was left out of the plot for better visualization, since a threshold
 of only 0.001 reduced the number of weights to only 11565. Incredibly, by
 discarding all weights out of the -0.005 to 0.005 range we notice an improvement
 of accuracy (3.29% against the original 3.35%) while reducing the number of
 225 weights to merely 5836, a 97.81% reduction. If we can afford a 10% worsening
 in the number of misses, we can set the threshold to 0.021 and the number of
 weights would be 2102.

A more profound comment on why the accuracy of the network increases when more than 90% of the weights are discarded is necessary. It is not clear why this happens, but at least three considerations may help us understand this phenomenon. First, the Occam's razor principle can be at play. Perhaps 266200 weights introduce far more complexity than the problem demands in reality - whenever the complexity of the model approaches the complexity of the problem, the better the model performs. So why maintaining such a huge structure if it is sub-utilized, one might ask. The removal of connections and neurons would lead to an irregular network, making it difficult to program and analyze. This is another advantage of a software implementation, the MLP is merely a logical structure, unused neurons and connections are simply bytes in a memory. Second, as the test data was not used at all during the training phase, some overfitting of the MLP to the training data is expected, even when a regularization method is used. This means that to some extent, the neural network will fail to generalize to data it has never seen before. Finally, a less relevant issue arises when it comes to the floating point representation. Maybe representation errors in such a small scale (the discarded weights were those whose absolute value was the smallest among all) can lead to non-negligible noise being propagated, and possibly amplified, by the network.

5. System Architecture

This section will go through the details of the implementation. Every hardware block will be described as well as the software design. Details on the physical deployment of the system will be given further on section 6.

The development of the image pipeline makes use, partially, of a framework for rapid prototyping of embedded vision applications - described in [18]. First the complete system was implemented in software, using a regular PC and the OpenCV library ([19]) for image processing functions. At this point, the functional behavior could be verified and changes in the structure could be easily made and evaluated. Then, the hardware/software partitioning took place and

the hardware blocks were implemented in SystemC ([20]). Every OpenCV function was then substituted by the SystemC counterpart and the whole system could be simulated and verified again. For descending to the final VHDL implementation, each block could be implemented separately and their correctness could be easily established by comparing their outputs with the already known to be precise SystemC version.

5.1. Hardware Blocks

5.1.1. Camera Interface

The camera used in the system is based on an 1/3-Inch Wide-VGA CMOS digital image sensor, [21]. The pixel readout is somewhat standard considering other sensors in the market. The image is captured in a Bayer pattern, even rows contain blue and green pixels whilst odd rows consist of red and green. The camera is capable of capturing images up to 752×480 pixels at a maximum frame rate of 60 fps, more than enough for our application. After the Bayer pattern is transformed in RGB triplets using a 3x3 window for interpolation, its intensity is calculated following the formula $Y = 0.2126R + 0.7152G + 0.0722B$, using fixed-point operations (24 bits for the decimal part and 8 bits for the integer part). This leaves us with pixel values ranging from 0 to 255.

5.1.2. Threshold

In order to attenuate illumination variances and differences in the background, a threshold must be applied. The MNIST dataset is composed of bright digits written over a dark background. In other words, the pen strokes have positive values and the background is always 0. To better approximate this scenario, the usually white (but light gray in practice) background must be nullified and the digit per se needs to be inverted as the intensity representation is the opposite. Equation 9 shows the precise operation (a value of 100 for T was empirically found to be adequate).

$$Y' = \begin{cases} 0 & \text{if } Y > T \\ 255 - Y & \text{if } Y \leq T \end{cases} \quad (9)$$

$$\frac{1}{16} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Figure 6: 5x5 Gaussian filter.

5.1.3. Gaussian Pyramid

A 224×224 virtual rectangle in the image is used as the region-of-interest (ROI). This ROI needs to be reduced to 28×28 pixels to serve as input to the neural network. Complex methods for image scaling are difficult to implement directly in hardware, specially when only a limited number of DSP blocks is available. Thus, a slightly less complicated method was used, but was proven to be just as effective. The image is filtered through a Gaussian convolutional kernel ([22]) as shown in figure 6, an then subsampled by a factor of 2. The blurring effect conserves some of the information that would be otherwise lost. This process is repeated 3 times, thus reducing the whole image by a factor of 8.

5.1.4. Display Interface

Although seeing the image captured by the camera is not of strict importance (the prediction of the neural network could be communicated through other communication medium, such as serial or JTAG), the current algorithm is still sensitive to scaling and rotation, meaning that the digit must be centered in the region-of-interest in order to be correctly processed. Therefore, a display module was connected to the system, allowing the user to see in real time the delimitation of the ROI as well as the neural network prediction. The developer can also benefit from such component for it grants the possibility of analyzing the performance of the whole image pipeline.

After the drawing of the bounding box around the ROI and the drawing of the predicted digit, the image goes through an external DDR3 memory that

acts as a frame buffer. The purpose of this step is twofold. First, the camera
300 and the display act in different clock domains. They also operate in different
frame rates, so some form of synchronization is mandatory. The display chosen
for this application is a TFT-PROTO, produced by MikroElektronika, [23]. Its
resolution is 320×240 and is connected to the FPGA by a serial SPI interface.

5.1.5. *MicroBlaze*

305 Finally and perhaps the most important block of all, the MicroBlaze proces-
sor ([4]). The MicroBlaze is a well known soft processor, i. e. it is implemented
using the FPGA standard logic blocks and, therefore, is highly customizable,
developed and distributed by Xilinx. The IP creation is a smooth process, where
the designer defines the connectivity of the MicroBlaze with a number of avail-
310 able peripherals (as well as custom peripherals, if desired), the characteristics of
the processor (performance, area, internal architecture to some extent, and so
on) and the external interface (to be connected to the FPGA logic or external
pins).

The features of the particular instance of the MicroBlaze used in this work
315 are as follows. A DDR3 controller is connected to the main AXI bus ([24]). The
DDR3 memory will hold the executable, so the MicroBlaze was equipped with
both instruction and data caches (64 kBytes of size, 8 words per line). Other
peripherals were connected to an AXI-Lite bus (a debug module to allow JTAG
communication with the processor, both to download the executable code and to
320 act as a standard input/output connection to the software; a custom interface
to receive the images from the FPGA logic through a dedicated block RAM
based FIFO buffer; a standard timer). Also, an extended floating point unit
was enabled, allowing for 64 bit operations.

5.2. *Software Design*

325 The software deployed in the embedded processor is rather simple. It is
merely a port of the previously implemented PC version with some simple
changes. As the training procedure was performed off-line on a standard work-

Figure 7: Hardware setup.

station, only the neural network structure for feeding the image through the neurons was maintained. The trained weights were hard coded into C++ header files to avoid the need for a file system. The whole program was written in C++,
 330 cross-compiled with GCC (O3 optimization enabled), leaving a footprint of 4.5 MBytes. Results regarding its performance are given in the next section.

6. Results

The system was synthesized using Xilinx tools for FPGA design: ISE, XPS
 335 and EDK (version 14.5). The FPGA of choice was a Spartan 6 (part number XC6SLX150-FGG484-2). The whole system consists of a GigaBee XC6SLX Series board, manufactured by Trenz Electronic (the datasheet can be found at [25]) as well as a custom made auxiliary board, with connections to the camera, the display and other peripherals for communication (more information can be
 340 found at our group's webpage: [26]). Figure 7 is a picture of the whole setup. The image pipeline was clocked at a frequency of 27 MHz, achieving a frame rate of approximately 60 frames per second. The MicroBlaze was clocked at 100 MHz. Table 1 refers to the resource utilization in the FPGA.

Table 2 shows a comparison of the average execution time for a different
 345 number of weights. The original neural network, with all the 266200 weights, takes 2.938 seconds for every prediction. Considering the processor limitations

Table 1: Resource utilization.

Resource	Usage	Description
Registers	6771 out of 184304.	1-bit flip-flops.
LUTs	7743 out of 92152.	Lookup tables.
Block RAM (18 Kb)	81 out of 268.	Total of 1458 Kbits.
Block RAM (9 Kb)	2 out of 536.	Total of 18 Kbits.
I/O Pins	125 out of 338.	-

in terms of resources and clock frequency, this result alone is not poor at all. New families of FPGA featuring hard processors embedded in the fabric, such as the Zynq, offer much faster clock frequencies and optimized architectures. The solution presented in this work could be easily ported, even to products manufactured by other companies.

From the second row of table 2 we notice a steep reduction of the execution time due to the exclusion of weights whose magnitude is lower than some threshold. By discarding all weights greater than -0.001 and smaller than 0.001, only 4.34% of them are left, leading to a reduction of 59.9% of the execution time while keeping the same accuracy. If the user can afford a missing rate of only 5%, the execution time can be further reduced to approximately 1 second per prediction.

7. Conclusion

In this work we have shown that simple and cheap embedded systems are already able of achieving acceptable performance when executing complex tasks. A smart software/hardware partitioning still needs to be executed by the designer, but current research is advancing towards automatic design flows. Difficult problems can be solved by deeply studying the target application field, in this case the field of machine learning. To the best of our knowledge, very

Table 2: Software performance.

Weight Threshold	Number of Weights	Misses (out of 10000)	Prediction time (s)
0.000	266200	335	2.938
0.001	11565	335	1.179
0.021	2102	365	1.118
0.067	1218	497	1.099

few research institutes are gravitating towards development and deployment of artificial intelligence in embedded processors using hardware accelerators, what makes this topic extremely relevant.

Immediate future work is needed to improve this solution. Currently only
 370 brighter background in relation to the handwritten digit can be handled, ex-
 tensions to deal with different contrasts could greatly enhance the robustness
 of the system. Also, automatic detection of relative rotation angles and scale
 should be devised. Considering technical implementation issues, we could cite
 the considerable size of the executable, 4 MBytes. This is due to the hard cod-
 375 ing of the weights in C++ header files. A better sparse matrix representation
 could reduce this size to a point where even the external memory could be ne-
 glected, leaving the whole code in internal FPGA memory, further improving
 the performance.

Acknowledgment

380 This work was supported in part by the National Science Foundation (NSF)
 under Grant 1302596.

Authors

Mr. Saldanha is currently a Ph. D. student in Computer Engineering at the University of Arkansas, United States (since 2014). He received his bachelor degree in Computer Engineering in 2011 from Universidade Federal do Rio Grande do Sul, Brazil, having spent the year of 2009 as a research assistant at the University of Potsdam, Germany. From 2010 to 2014, Mr. Saldanha worked in the industry of telecommu-

Dr. Bobda received the Licence in Mathematics from the University of Yaounde, Cameroon, in 1992, the diploma of Computer Science and the Ph.D. degree (with honors) from the University of Paderborn in Germany in 1999 and 2003, respectively. In June, 2003, he joined the Department of Computer Science at the University of Erlangen-Nuremberg in Germany as Post-Doc. In 2005, Dr. Bobda was appointed assistant professor at the University of Kaiserslautern. From 2007 to 2010, Dr. Bobda was Professor at the University of Potsdam. Since then, Dr. Bobda is an assistant professor at the University of Arkansas, United States.

References

- [1] A. Gomperts, A. Ukil, F. Zurfluh, Development and implementation of parameterized fpga-based general purpose neural networks for online applications., IEEE Trans. Industrial Informatics 7 (1) (2011) 78–89.
- [2] A. Dinu, M. Cirstea, S. Cirstea, Direct neural-network hardware-implementation algorithm, Industrial Electronics, IEEE Transactions on 57 (5) (2010) 1845–1848. doi:10.1109/TIE.2009.2033097.

- [3] M. Ruiz-Llata, G. Guarnizo, M. Yebenes-Calvino, Fpga implementation of a support vector machine for classification and regression, in: Neural Networks (IJCNN), The 2010 International Joint Conference on, 2010, pp. 1–5. doi:10.1109/IJCNN.2010.5596820.
- [4] Xilinx, MicroBlaze Processor Reference Guide, v9.0 (01 2008).
- [5] Xilinx, Zynq-7000 All Programmable SoC Overview, v1.7 (10 2014).
- [6] A. Baker, Simplicity, in: E. N. Zalta (Ed.), The Stanford Encyclopedia of Philosophy, fall 2013 Edition, 2013.
- [7] F. Girosi, M. Jones, T. Poggio, Regularization theory and neural networks architectures, Neural Computation 7 (2) (1995) 219–269. doi:10.1162/neco.1995.7.2.219.
- [8] Z. Xu, H. Zhang, Y. Wang, X. Chang, Y. Liang, L1/2 regularization, SCIENCE CHINA Information Sciences 53 (6) (2010) 1159–1169.
- [9] A. Y. Ng, Feature selection, l1 vs. l2 regularization, and rotational invariance, in: Proceedings of the Twenty-first International Conference on Machine Learning, ICML '04, ACM, New York, NY, USA, 2004, pp. 78–. doi:10.1145/1015330.1015435.
URL <http://doi.acm.org/10.1145/1015330.1015435>
- [10] W. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, Bulletin of Mathematical Biophysics 7 (1943) 115–133.
- [11] J. Schmidhuber, Deep learning in neural networks: An overview, CoRR abs/1404.7828.
URL <http://arxiv.org/abs/1404.7828>
- [12] F. Rosenblatt, The perceptron – a perceiving and recognizing automaton, Tech. Rep. 85-460-1, Cornell Aeronautical Laboratory, Buffalo, NY, 00344 (1957).

- [13] P. Dawkins, Paul's online math notes, <http://tutorial.math.lamar.edu/Classes/CalcI/ChainRule.aspx> (March 2015).
- 440 [14] J. C. Principe, N. R. Euliano, W. C. Lefebvre, Neural and Adaptive Systems: Fundamentals Through Simulations, John Wiley and Sons, 2000.
- [15] L. V. Fausett, Fundamentals of neural networks : architectures, algorithms, and applications, Prentice-Hall International, Englewood Cliffs, NJ, 1994.
URL <http://opac.inria.fr/record=b1084188>
- 445 [16] D. R. Wilson, T. R. Martinez, The general inefficiency of batch training for gradient descent learning, Neural Networks 16 (10) (2003) 1429–1451.
- [17] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (11) (1998) 2278–2324.
- 450 [18] M. Mefenza, F. Yonga, L. B. Saldanha, C. Bobda, Velipassalar, A framework for rapid prototyping of embedded vision applications, to appear (2014).
- [19] G. Bradski, The opencv library, Dr. Dobb's Journal of Software Tools.
- [20] P. R. Panda, Systemc: A modeling platform supporting multiple design
455 abstractions, in: Proceedings of the 14th International Symposium on Systems Synthesis, ISSS '01, ACM, New York, NY, USA, 2001, pp. 75–80.
doi:10.1145/500001.500018.
URL <http://doi.acm.org/10.1145/500001.500018>
- [21] Aptina Imaging, 1/3-Inch Wide-VGA CMOS Digital Image Sensor, rev. A
460 10/08 EN (10 2008).
- [22] R. Szeliski, Computer Vision: Algorithms and Applications, 1st Edition, Springer-Verlag New York, Inc., New York, NY, USA, 2010.
- [23] MikroElektronika, TFT PROTO, ver. 2.00.

- [24] Xilinx, AXI Bus Functional Models, v1.9 (06 2011).
- 465 [25] Trenz Electronic, GigaBee XC6SLX Series User Manual, v 1.01 (05 2012).
- [26] S. R. Group, Smartes, <http://smartes.uark.edu/> (December 2014).