NeNEB - An Application Adjustable Single Chip Neural Network Processor for Mobile Real Time Image Processing

L. Larsson, S. Krol, K. Lagemann
University of Hamburg, Dept. of Computer Science / Group TECH
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany
e-mail: larsson@informatik.uni-hamburg.de

Abstract

A digital implementation of the recall phase of a backpropagation neural network for real time image classification is presented. The motivation has been, that parallelism of a neural network has less advantage, if the input data stream is sequential as the pixel stream of an usual CCD camera. Additional, classifying a stream of pixels with a single chip neural processor implanted into a camera avoids the bottleneck caused by image data transfer and storage, which is neglected often. But the chip will not be restricted to real time image processing applications. The chip realizes a network with 32 output neurons, 8 hidden neurons and up to 64K inputs ($\approx$ 512K synapses). It is estimated to classify 50 gray scale images per second of size up to 256×256 *pixels, which complies to 26M cps. The weights of hidden neurons are stored in external memory, whereas weights of output neurons are stored on-chip.*

1 Introduction

The backpropagation (*BPG*) algorithm [1, 2, 3], introduced by D. E. Rumelhart, J. L. McCelland, G. E. Hinton and R. J. Williams, is used to train multi layer feed forward neural networks by supervised learning. Our hardware implementation called *NeNEB* (*German for **Ne**uronales **N**etzwerk zur **E**chtzeitklassifizierung von **B**ildern, neural network for real time classification of images*) has been inspired by *ALVINN* (*Autonomous Land Vehicle In a Neural Network*) presented in [4, 5], where an 30×32 image (reduced resolution) is used as the input vector of a *BPG* network with 4 hidden neurons and 30 output neurons. The output vector is used for steering the vehicle in real time. Our implementation is intended to small autonomous vehicles or robots.

We choose an external weight storage scheme for the synaptic weights of the hidden neurons combined with bit serial multipliers to reduce communication with the external weight memory and to reduce the chip area for the multipliers. This allows a low cost single chip implementation which provides the processing speed necessary for real time image processing tasks. Additionally, we choose a piecewise linear approximation of sigmoid function as presented in [6].

Published by IEEE in Proceedings of the International Workshop on Neural Networks for Identification, Control, Robotics, and Signal/Image Processing, NICROSP'96, pp. 154-162, August 21-23, 1996, Venice, Italy, ISBN 0-8186-7456-3

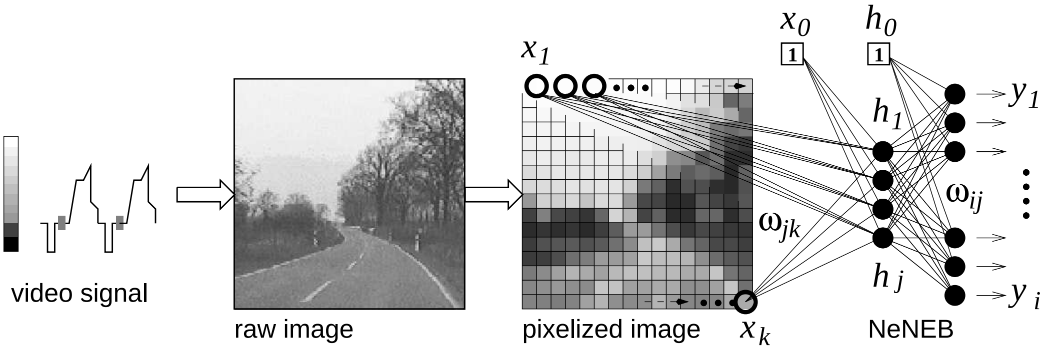

Figure 1. Video Signal Processing and Calculating Scheme *Each of the (eight) hidden neurons h_j calculates (serial for pixel x_1 to pixel x_k) its output activity. Similar, each output neuron y_i calculates (serial for hidden neuron h_1 to hidden neuron h_j) its output activity. All neurons will be initialized with their individual thresholds.*

1.1 Image Preprocessing

Image preprocessing differs from [4, 5], but we also reduce the resolution of the primary image. Reduction of the resolution is necessary to get a rate of pixels compatible with our neural network implementation. It strongly reduces the hardware effort and holds the freedom to adjust the system in a wide range of system frequencies.

To reduce the resolution of the raw input images, we *pixelize* them (fig. 1). *Pixelizing* here means applying a 2×2 mask to the source image to calculate inter lattice pixels similar to a Laplace filter, but with a rectangular filter characteristic in the frequency domain of the image [7, 8]. Both the number of columns c and the number of rows r are reduced by a factor of 2^n, so that the pixel rate is reduced by a factor of $(2^n)^2$. This allows to adapt any input pixel rate to the performance of the network. We choose *pixelizing* as image preprocessing, because humans easily recognize images, even if the number of pixels is reduced from say 512^2 to $256^2, 128^2, 64^2$. Each pixelized frame is an input vector to the neural network (NeNEB) and is processed pixel by pixel to be classified (fig. 1).

In the proposed system, the video signal of a low cost gray scale CCD camera is digitized by a standard 8 bit Flash-ADC (MC10319) [9]. This signal is fed into a purely digital VHDL model of a frame grabber, which has been developed and was prototyped by mapping to two *Altera 7128 CPLDs* [10]. The grabber continuously extracts raw images frame by frame from the sampled composite video signal.

1.2 BPG Network

This section summarizes the backpropagation network model used for NeNEB. Each neuron i of a net computes

$$y_i = g(\theta_i + \sum_{j=1}^{n} \omega_{ij} \cdot x_j). \tag{1}$$

In case of *bias* equation (1) is modified by $\omega_{i0} := \theta_i$, $x_0 := 1$ and $j = 0$. A net with k inputs, j hidden and i output neurons maps an image $\vec{x}$ of k pixels to a classification

$$\vec{y} = F(\Omega_{ij}, \Omega_{jk}, \vec{x}), \tag{2}$$

where Ω_{ij} and Ω_{jk} are weight matrices. With these, the feed forward mapping (2) is given by

$$\begin{aligned} \vec{h} = \Omega_{jk} \cdot \vec{x}, \quad \vec{\gamma} = g(\vec{h}), \\ \vec{o} = \Omega_{ij} \cdot \vec{\gamma}, \quad \vec{y} = g(\vec{o}), \end{aligned} \tag{3}$$

where $\vec{x}$, $\vec{h}$, $\vec{\gamma}$, $\vec{o}$ and $\vec{y}$ are column vectors and

$$\begin{aligned} g(a_i) &= 1/(1 + e^{-a_i/\tau}) \\ g'(a_i) &= g(a_i) \cdot g(-a_i)/\tau, \end{aligned} \tag{4}$$

with a temperature parameter τ (*typical* $0.1 < \tau < 10$). Training is done with patterns $(\vec{x}; \vec{t})_M$ ($\vec{x} \approx$ *input image*; $\vec{t} \approx$ *target classification*), until

$$E = \frac{1}{2} \sum_M (\vec{y} - \vec{t})^2_M \tag{5}$$

becomes minimal (or less a limit), by computing

$$\begin{aligned} \vec{d} = \vec{y} - \vec{t}, \qquad & \vec{\delta} = g'(\vec{o}), \\ \vec{e} =^t \Omega_{ij} \cdot (\vec{d} \otimes \vec{\delta}), \quad & \vec{\epsilon} = g'(\vec{e}), \end{aligned} \tag{6}$$

for each pattern, where ${}^t\Omega_{ij}$ is the transposed matrix Ω_{ij} and $\otimes$ a component like vector-vector product

$$\vec{c} = \vec{a} \otimes \vec{b} \Leftrightarrow c_i = a_i \cdot b_i. \tag{7}$$

With this, the update of the matrices

$$\begin{aligned} \Omega_{ij} &:= \Omega_{ij} - \eta \cdot \vec{\delta} \cdot^t \vec{\gamma} \quad + \quad \alpha \cdot \Delta\Omega_{ij} \\ \Omega_{jk} &:= \Omega_{jk} - \eta \cdot \vec{\epsilon} \cdot^t \vec{x} \quad + \quad \alpha \cdot \Delta\Omega_{jk} \end{aligned} \tag{8}$$

appears in direction of the negative gradient of the error function E, with a learning rate η (*typical* $0.001 < \eta < 0.5$) and momentum parameter α (*typical* $0.1 < \alpha < 0.9$).

2 Implementation

To implement the BPG algorithm, one has to build the basic operations of this algorithm. These are matrix multiplication (3), the sigmoid function $g(a_i)$ and its derivation $g'(a_i)$ (4), the vector difference, vector-vector product (7), multiplication with transposed matrix (6) and updating of the matrices (8). We implement the feed forward path of the BPG-Algorithm. To implement the feed forward path only, one just have to realize the matrix multiplication (3) and the sigmoid function $g(a_i)$

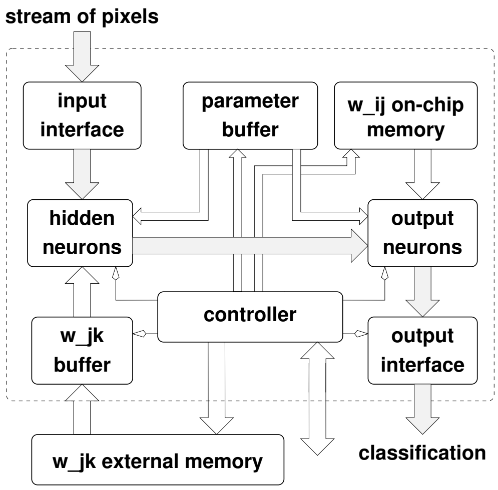

Figure 2. **NeNEB-Chip** *block diagram.*

(4). Our hardware implementation enfolds the realization of VHDL modules being functional equivalent to equation (1) for the neurons, performing the classification of equation (2). The sigmoid function (4) is approximated by piecewise linear straight lines. The gradients of these straight lines are of powers of two. Backpropagation training accordingly to equations (6), (7) and (8) was done by a program written in C. As the result of training the program generates a net file containing the synaptic weight matrices. This net file is then converted into the data for memory storage in the scheme of the NeNEB chip. BPG neural networks have a high inherent parallelism, which is useful for implementation on massive parallel architectures [11]. But if the input data stream is serial, this inherent parallelism gives less advantage. So, we realized eight parallel hidden neurons, where each neuron computes its weighted sum over its inputs sequentially – pixel by pixel (fig. 5).

3 Architecture

We choose an external weight storage scheme for the synaptic weights of the hidden neurons combined with bit serial multipliers to reduce the number of pads for access communication with the external weight memory and to reduce the chip area for the multipliers. The NeNEB chip contains address counters allowing stand alone processing together with an ROM chip for the weight memory storage. The usage of the NeNEB chip as a co-processor of a host computer is also possible. Additionally, we choose a piecewise linear approximation of sigmoid function as presented in [6]. A network with up to 65536 inputs x_k, 8 hidden neurons h_j and 32 output neurons y_i (a [65536-8-32] net) will be possible. The input data range is $\approx 0.0, \ldots, 1.0$ with 8 bit resolution. To realize the large number ($\approx 512K$) of synaptic weights between inputs and hidden neurons, these weights will be stored in an external memory. The storage scheme chosen to reduce the necessary pads for communication between NeNEB chip

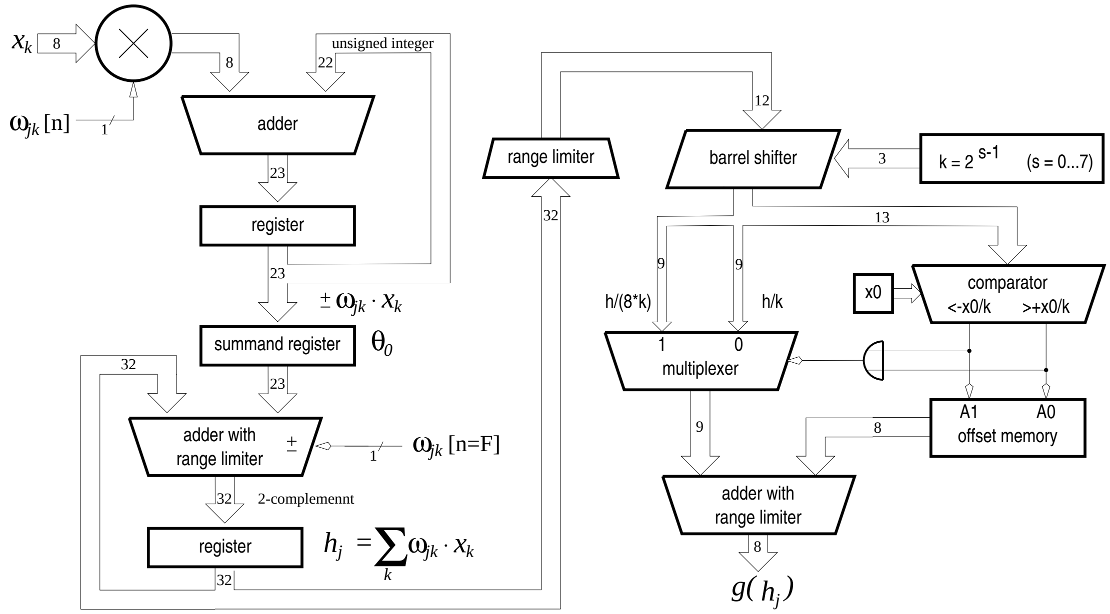

Figure 3. **Neuron Implementation Scheme** *using multiplier performing weighting of the neuron's inputs by multiplying an input activity* x_k *by synaptic weight* ω_{jk} *bit serially and summing these products. The sigmoid function* $g(x)$ *is approximated by a piecewise linear function like [6].*

and memory will be described.

3.1 Neuron

The hidden neurons and output neurons are implemented according to figure (3), except that output neurons weight memory is on-chip (fig. 2). The multiplication $\omega_{jk} \cdot x_k$ is done by a bit serial multiplier. There are 17 clock periods necessary to complete a bit serial multiplication (*17 = 1+15+1 clock periods are necessary per pixel, 1 to initialize, 15 for bit sequential multiplication* $x \cdot \omega$*, 1 to handle sign*). So, the pixel clock frequency has to be a factor of 17 lower than the chip clock.

3.2 Sigmoid Function

A piecewise linear approximation $y(x)$ of the sigmoid function $g(x)$ (4) similar to [6] has been chosen. An inverse temperature parameter $c = 1/\tau$ with $c \in \{0.0625, 0.125, 0.25, 0.5, 1, 2, 4, 8\}$ can be chosen. The slope of the sigmoid function approximation is determined by the slope parameter $\mathsf{s} \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ and the offsets x0 stored in the offset memory (fig. 3). With this, the sigmoid function can be interpolated in a wide range (fig. 4). The approximation of the sigmoid function is defined as

$$g(x) \approx y(x) = \begin{cases} 0 & \text{if} \quad \mathsf{x}/(8*\mathsf{k})+o_- \quad < \quad 0 \\ \mathsf{x}/(8*\mathsf{k})+o_- & \text{if} \quad \mathsf{x} \quad \leq \quad -\mathsf{x0} \\ \mathsf{x}/\mathsf{k} & \text{if} \quad -\mathsf{x0} < \mathsf{x} < +\mathsf{x0} \\ \mathsf{x}/(8*\mathsf{k})+o_+ & \text{if} \quad \mathsf{x} \quad \geq \quad +\mathsf{x0} \\ 1 & \text{if} \quad \mathsf{x}/(8*\mathsf{k})+o_+ \quad > \quad 1 \end{cases} \quad \text{with} \quad \mathsf{k} = 2^{\mathsf{S}-1}. \tag{9}$$

The reciprocal gradient parameter k will be chosen as closest to $1/g'(x)$. To complete the chosen approximation (9) for the sigmoid function, the threshold(s) $\mathsf{x0}$ (and $-\mathsf{x0}$) have to be determined. The $\mathsf{x0}$ (and $-\mathsf{x0}$) are determined, so that the area difference

$$\begin{aligned} F(x_0) &= |G(x_m) - Y(x_0, x_m)| = |\int_0^{x_m} g(x)dx - \int_0^{x_m} y(x_0, x)dx| \\ &\approx |(x_m + \tau \cdot \log_2(\tfrac{1}{2})) - (\int_0^{x_0} y_1(x)dx + \int_{x_0}^{x_1} y_2(x)dx + \int_{x_1}^{x_m} y_3(x)dx)| \end{aligned} \tag{10}$$

between the sigmoid function $g(x)$ and the piecewise linear approximated sigmoid function $y(x)$ becomes zero for the interval $[0, \ldots, x_{max}]$. It's sufficient to regard the interval $[0, \ldots, x_{max}]$ instead of $[-x_{max}0, \ldots, x_{max}]$, because the sigmoid function $g(x)$ and it's approximation $y(x)$ behave symmetric with respect to $x = 0$. The integration limit $x_m < \infty$ is for integral convergence. The final results are $\mathsf{k} = 1/g'(0)$ with $\mathsf{k} \in \{\frac{1}{2}, 1, 2, 4, 8, 16, 32, 64\}$ and $\mathsf{x0} \approx 0.3825 \cdot \mathsf{k}$. With $a = \frac{7}{8 \cdot \mathsf{k}} \cdot \mathsf{x0}$ and $b = g(0)$, the offsets are $o_- = b - a$ and $o_+ = b + a$.

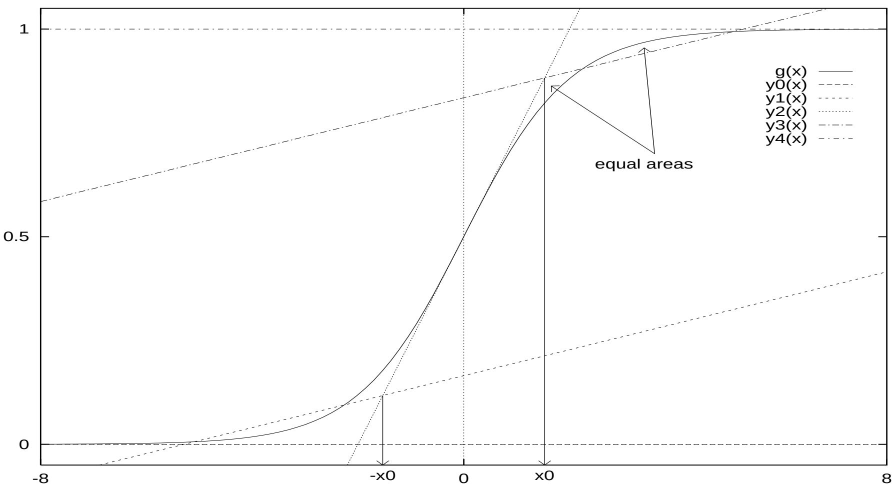

Figure 4. Piecewise Linear Approximation of Sigmoid Function *The sigmoid function $g(x)$ is approximated by five straight lines named $y_0(x)$, $y_1(x)$, $y_2(x)$, $y_3(x)$, $y_4(x)$, whereat $y_0(x)$ and $y_4(x)$ are results of range limitations. So, three lines $y_1(x), y_2(x), y_3(x)$ with two different gradients $1/k$ and $1/(8 \cdot k)$ approximate $g(x)$.*

3.3 Hidden Neurons Weight Storage Scheme

The scheme of data organization (fig. 5) for storage of the weights of the hidden neurons has been designed, to allow many weights with a small number of pads for

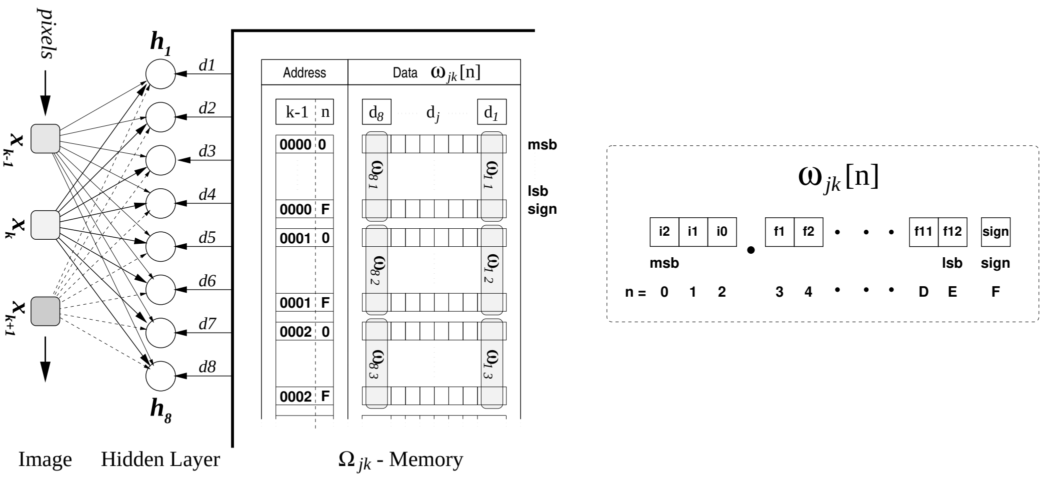

Figure 5. **Synaptic weight storage** *in a standard byte wide organized memory. Each data line ($d_8 \ldots d_1$) is connected to one hidden neurons bit serial input $\omega_{jk}[n]$. Successive addresses hold the successive bits of the weights. This scheme allows a large number of synapses for the hidden layer neurons.*

communication with the external weight memory. The eight hidden neurons compute in parallel but bit serial. So, eight data lines (pads) only are necessary for the eight hidden neurons. These are connected to the hidden neurons bit serial multipliers. The bit serial multiplication saves chip area. The data bits of each weight are stored on sequent addresses. Additionally, 20 address lines (two bytes per ω_{jk}) to address the weight memory are provided which allows up to 64K synapses per hidden neuron.

3.4 Numerical Representations

Different types of numerical representations take different ranges of interim results into account. The number of bits necessary has been taken into account with respect to literature [12, 13] but is also based on our own simulations. Also, the range and possible magnitude of input data have been taken into account. This is important while summing up products $x \cdot \omega$ to avoid overflow. Choices are shown in table (1).

component	$\approx$ range	$[\pm][bit].[bit]$	representation
$x_k,\ y_i$	$0.0 \ \ldots \ 1.0$	$[0].[8]$	unsigned
ω_{ij}, ω_{jk}	$-8.0 \ \ldots \ 8.0$	$[\pm][3].[12]$	signed
$\lvert\omega\rvert \cdot x$	$0.0 \ \ldots \ 8.0$	$[3].[20]$	unsigned
$\sum \omega \cdot x$	$-2048.0 \ldots 2048.0$	$[\pm][11].[20]$	2's complement
h_j, o_i	$-8.0 \ \ldots \ 8.0$	$[\pm][3].[8]$	2's complement
$g(h_j), g(o_j)$	$0.0 \ \ldots \ 1.0$	$[0].[8]$	unsigned
x0/k	$-1.0 \ \ldots \ 1.0$	$[\pm][0].[8]$	2's complement
offset(x0)	$0.0 \ \ldots \ 1.0$	$[0].[8]$	unsigned

Table 1. Fixed Point Data Representations

3.5 Neural Network Training (C)

The training of a net is done with `float` (*single precision*) on a workstation. While training, a continuous sigmoid function has been used. But the piecewise linear approximated sigmoid function could be used for training as well. After successful completion of the training, the synaptic weights are converted to the discrete values needed by the NeNEB chip. Classification results of the software simulation (*single precision*) have been used for comparison with classification results of the hardware simulation (*integer arithmetics*).

3.6 Hardware Simulation (VHDL)

A VHDL model for NeNEB has been developed. Several example problems were used with the functional VHDL model to verify the NeNEB chip: The simple *xor* (2-2-1 net), a *decoder* (3-3-8 net) and *character recognition* of of 5×7 pixels (35-8-26 net) have been successfully fed through the functional VHDL model of the NeNEB chip. The recall behavior of the NeNEB chip VHDL model – with it's integer arithmetics and the piecewise linear approximated sigmoid function – was the same as the behavior of the C program.

4 Results

The real time frame grabber VHDL model and the neural network processor NeNEB are still under development. All components of the NeNEB-Chip have been simulated (*VHDL System Simulator (VSS)*) and synthesized (*Design Compiler (DC)*) with *SYNOPSYS* [14] using the *ES2 ecpd07* CMOS library as the target process. Functional validation of the synthesized logic is done successfully at 60 MHz clock frequency. With this, the estimated performance is $\approx$ 25 times higher than the performance of a workstation, which is in the range of $\approx 1Mcps$.

device	$[mm^2]$	comment
Hidden Neuron	0.63	external weight memory, up to 512k ω_{jk}
Output Neuron	1.1	on-chip weight memory, up to 32×8 ω_{ij}
Interface	1.0	host, memory, etc.
FSMs	0.26	sequencers, etc.
Counters	0.2	address counters, etc.
Total Chip Area	42.0	active area, unrouted, without pads

Table 2. **Estimated Chip Areas** *as synthesis results done with SYNOPSYS for ES2 ecpd07 as the target CMOS process.*

5 Conclusion

We present a single chip implementation of a backpropagation network intended for image classification. The functional VHDL model and the synthesized gate level

$[65536-8-32]$	inputs - hidden neurons - output neurons
65 + 12 pads	signal pads + power pads
60 MHz	chip clock frequency (mixed functional / gate level simulation)
26 Mcps	maximum performance
0.6 Mcps / mm^2	connections per second chip area ratio (unrouted)
3.3 M pixels s^{-1}	input rate
50 images s^{-1}	gray scale of 256 levels, size up to 256×256 pixels

Table 3. Technical Data *of the NeNEB chip*

model of the NeNEB chip has been successfully proved by several classification problems (xor, decoder, character recognition) at specified target speed. Estimated chip area and speed are results of logic synthesis done with *SYNOPSYS* [14], which we use also to verify synthesized logic by mixed functional simulation and gate level simulation. Final chip layout and post layout timing simulation with *CADENCE DFW II* [15] are still pending.

References

[1] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning Internal Representations by Error Propagation", in J. A. Anderson and E. Rosenfeld (eds.) *Neurocomputing: Foundations of Research*, MIT Press 1988, pp. 673

[2] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning Representations by Back Propagating Errors", in J. A. Anderson and E. Rosenfeld (eds.) *Neurocomputing: Foundations of Research*, MIT Press 1988, pp. 696

[3] D. E. Rumelhart, J. L. McCelland, *Parallel Distributed Processing*, Vol. 1&2, MIT Press, 1986

[4] D. Pomerleau, "ALVINN: an Autonomous Land Vehicle in a Neural Network", *Carnegie-Mellon Univ: reports; CMU-CS-89-107*, 1989.

[5] D. A. Pomerleau, *Neural Network Perception for Mobile Robot Guidance*, Kluwer academic publishers, 1993.

[6] J. Cloutier, P. Y. Simard, "Hardware Implementation of the Backpropagation without Multiplication", in Proc. of the Fourth Int. Conf. on Microelectronics for Neural Networks and Fuzzy Systems, Turin, Italy, September 26-28, 1994, pp. 46-55, 1994.

[7] G. A. Baxes, *Digital Image Processing*, Principles and Applications, John Wiley & Sons, 1994.

[8] B. Jähne, *Digitale Bildverarbeitung*, Springer Verlag, 1991.

[9] *Analog/Digital and Digital/Analog Conversion Manual*, Motorola, 1984.

[10] *Altera Databook*, Altera Corporation, San Jose CA, 1993.

[11] A. Zell, *Simulation Neuronaler Netzwerke*, Addison-Wesley, 1994.

[12] P. W. Hollis, J. S. Harper, J. J. Paulos, "The Effect of Precision Constrains in a Backpropagation Learning Network", *Neural Computing 2*, pp. 363-373, 1990.

[13] M. Stevenson, R. Winter, B. Widrow, "Sensitivity of Feedforward Neural Networks to Weight Errors", *IEEE Transactions on Neural Networks*, Vol. 1, No.1, March 1990.

[14] *Synopsys VHDL Simulator Reference Manual*, Synopsys Inc., 1993.

[15] *Cadence Design Framework-II*, Cadence Inc., 1993.