Intelligent Edge Detection using a CUDA Simulator of Multilayer Neural Network Based on Multi-Valued Neurons

Jeff Wilson¹ and Igor Aizenberg¹
¹ Texas A&M University-Texarkana, Texarkana, TX, USA

Abstract - In this paper, we consider the edge detection problem using an intelligent approach. We use a multilayer neural network based on multi-valued neurons (MLMVN) as an intelligent edge enhancer. MLMVN is a complex-valued neural network and it has many advantages over classical neural networks. It significantly outperforms a classical multilayer feedforward neural network in terms of learning speed, number of parameters employed, and generalization capability. MLMVN has already shown its efficiency in solving edge detection problem. Here we significantly improve its performance employing a GPU software simulator of MLMVN, which allows parallelization of the learning process. This makes it possible to speed up the learning process significantly. Compared to a regular CPU (serial) software simulator, a parallel simulator, which is described in this paper, is about 10 times faster for a 3-layer network with 9 inputs, 45 hidden neurons, and 1 output neuron.

Keywords: edge detection, complex-valued neural network, CUDA, intelligent image processing

1 Introduction

In this paper, we consider the edge detection problem using an intelligent approach. There are likely more algorithms for detecting and enhancing edges in the image processing literature than for any other single subject [1], [2]. Edges form the outline of an object. If the edges in an image can be identified accurately, all of the objects of interest can be located visually, and their basic properties such as area, perimeter, and shape can be measured. Edge detection, in general, consists of two parts: edge detection, which is a process for calculating the edge magnitude at each pixel; and edge localization, which is a process for determining the exact edge location [3]. Once an edge is enhanced properly, the location of the edge can be identified accurately. The problem of edge detection in images taken from real-world scenes is that these images are usually noisy, while all edge detection algorithms are developed with no respect to noise. As a result, it is often very difficult to localize edges of the objects of interest against edges of a noisy texture. A similar problem appears when it is necessary to localize edges of an object of interest against edges of background textures. Thus, robust edge enhancement against edges of noisy and (or) preventing textures is quite important for edge detection algorithms.

In this paper, we suggest solving the edge detection problem using an intelligent approach. This approach employs a neural network to learn some edge detection operator from “teaching” images. After a learning session is finished, a trained machine learning tool can be used to detect edges on other images, which did not participate in the learning session. This idea was utilized in [3] using a classical multilayer feedforward neural network (MLF) with a modified output neuron. It was shown [3] that MLF can be used to learn the Sobel edge detector from a single real-world or synthesized image and then, after a learning process is complete, detect Sobel edges on the images which did not participate in the learning process. In [4], a multilayer neural network based on multi-valued neurons (MLMVN) [5] was used as an intelligent edge enhancer. This is a complex-valued neural network. The multi-valued neuron (MVN), which is a basic neuron of this network, has important advantages over a neuron with sigmoid activation function. These advantages follow from the MVN main properties [6]-[8]. This neuron operates with complex-valued weights. Its inputs and output are located on the unit circle. MVN may learn non-linearly separable input/output mappings, and its derivative-free learning is based on the error-correction learning rule [8]. MLMVN is the most successful applications of the MVN. This is a feedforward neural network with a traditional feedforward topology where neurons are integrated into layers, and the output of each neuron from the current layer is connected to the corresponding inputs of neurons from the following layer. However, MLMVN consists of multi-valued neurons, and this is its main distinction from a classical feedforward neural network. Its great advantage over other machine learning techniques is its original derivative-free backpropagation learning algorithm, which was suggested in [5] and described in detail in [8]. Its other important advantage is better performance in terms of generalization capabilities compared to a standard feedforward neural network, euro-fuzzy and kernel-based networks [5], [8].

It was shown in [4] that MLMVN with a single hidden layer can be successfully used not only to formally learn a certain edge detection operator, but also to learn how to detect edges corresponding to this operator on a noisy image, ignoring a noisy texture.

However, there is a one clear bottleneck, which reduces productivity of any traditional Central Processor Unit (CPU)-based (serial) software simulator of a neural network. The
learning process is iterative and it is time consuming. The larger is a network and the more learning samples are presented in the learning set, the more significant time is needed to train the network. This is especially sensitive for intelligent image processing applications where it is desirable to use learning sets containing thousands learning samples. The best way to speed the learning process up is to create a software simulator, which allows parallelization. In fact, error backpropagation and correction of the weights in multilayer feedforward neural network can be done in parallel for neurons from the same layer. The most suitable way for creation of such a parallel simulator is to use a Graphics Processing Unit (GPU) as an alternative to CPU and CUDA [9], a parallel computer architecture developed by NVIDIA. Several efficient CUDA-based simulators were developed for standard MLF (e.g., [10], [11] should be mentioned). However, no such simulators are known for complex-valued neural networks to our best knowledge. In this paper, we would like to cover this gap developing such a CUDA-GPU based simulator for MLMVN. This simulator makes it possible to drastically reduce that time, which is needed for learning. For example, for the network 9-45-1 (9 inputs, 45 hidden neurons and 1 output neuron), which we use for intelligent edge detection, it was succeeded to reach 10-times learning time reduction.

2 MVN and MLMVN

The discrete MVN was introduced in [6]. It implements a mapping between \( n \) inputs and a single output. This mapping is described by a threshold function of \( k \)-valued logic over the field of complex numbers. Let \( e^k_j = e^{j2\pi/k} \), where \( i \) is an imaginary unity, be the primitive \( k \)th root of unity. Let \( E^k = \{ e^k_0, e^k_1, e^k_2, \ldots, e^k_{k-1} \} \) (see Fig. 1). In \( k \)-logic over the field of complex numbers, values of logic are encoded by \( k \)th roots of unity, that is by elements of the set \( E^k \). Hence, the MVN input/output mapping is described by a function of \( n \) variables \( f(x_1, \ldots, x_n) \), which is either a function \( f: E^a_k \rightarrow E^k \) or a function \( f: O^a \rightarrow E^k \) (where \( O \) is a set of points located on the unit circle). Thus MVN performs the following input/output mapping [6]-[8]:

\[
f(x_1, \ldots, x_n) = P(w_0 + w_1 x_1 + \ldots + w_n x_n),
\]

where \( x_1, \ldots, x_n \) (\( x_j \in E^k \) or \( x_j \in O; j = 1, \ldots, n \)) are the neuron inputs, and \( w_0, w_1, \ldots, w_n \) are the weights. Evidently, \( f(x_1, \ldots, x_n) \in E^k \). \( P \) is the activation function of the neuron:

\[
P(z) = e^{j2\pi/k}
\]

Fig. 1. Geometrical interpretation of the discrete MVN activation function

\[
P(z) = \exp(i2\pi/j/k),
\]

if \( 2\pi j/k \leq \arg z < 2\pi (j+1)/k, \) (2)

where \( j=0, 1, \ldots, k-1 \) are values of the \( k \)-valued logic, \( z = w_0 + w_1 x_1 + \ldots + w_n x_n \) is the weighted sum, \( \arg z \) is the argument of the complex number \( z \).

Function (2) divides a complex plane into \( k \) equal sectors and maps the whole complex plane into a set of \( k \)th roots of unity (see Fig. 1). If the weighted sum is located in sector \( j \) then the neuron’s output is \( e^j \). Hence, the MVN’s output is determined by the argument (phase) of the weighted sum and does not depend on its magnitude.

The MVN learning is derivative-free. The most efficient MVN learning algorithm is based on the error-correction learning rule. The weights adjustment is completely determined by the neuron’s error, which is calculated as a difference between the desired and actual outputs. The error-correction learning rule is [7], [8]:

\[
W_{r+1} = W_r + \frac{C_r}{n+1} \left( e^q - e^s \right) \bar{X},
\]

and with a modification suggested in[5]:

\[
W_{r+1} = W_r + \frac{C_r}{n+1} \left| \frac{z_r}{z_r} \right| \left( e^q - e^s \right) \bar{X},
\]

where \( \bar{X} \) is the vector of neuron inputs with the components complex-conjugated, \( n \) is the number of neuron inputs, \( e^q \) is the desired output of the neuron, \( e^s = P(z) \) is the actual output of the neuron, \( r \) is the number of the learning step, \( W_r \) is the current weighting vector (to be corrected), \( W_{r+1} \) is the following weighting vector (after correction), \( C_r \) is the constant part of the learning rate (it may always be equal to 1),
and \(|z_j|\) is the absolute value of the weighted sum obtained on the \(r\)th learning step.

In [5], it was suggested to use the MVN as a basic neuron in a feedforward neural network - a multilayer neural network with multi-valued neurons (MLMVN). This is a multilayer neural network with a standard feedforward topology where neurons are integrated into layers, and output of each neuron from the current layer is connected to the corresponding inputs of neurons from the following layer. However, its basic neuron, MVN, determines a number of important distinctions and advantages of MLMVN over a standard MLF based on sigmoidal neurons.

The most important advantage of MLMVN is its derivative-free backpropagation learning algorithm, which was suggested in [5] and comprehensively observed in [8]. Let us recall how the MLMVN backpropagation learning is organized.

Let MLMVN contain one input layer, \(m-1\) hidden layers and one output layer. Let us use the following notations. Let \(D_{jm}\) be a desired output of the \(j^{th}\) neuron from the \(m^{th}\) (output) layer; \(Y_{jm}\) be an actual output of the \(j^{th}\) neuron from the \(m^{th}\) (output) layer. Then the global error of the network taken from the \(j^{th}\) neuron of the \(m^{th}\) (output) layer is calculated as follows:

\[
\delta_{jm}^* = D_{jm} - Y_{jm}. \tag{5}
\]

The MLMVN learning algorithm is derived [5], [8] from the consideration that the global error of the network depends on the local errors of all the neurons and therefore it must be shared among all the neurons because all of them contribute to this error by their local errors.

The backpropagation of the global errors \(\delta_{jm}^*\) through the network is used (from the \(m^{th}\) (output) layer to the \(m-1^{st}\) one, from the \(m-1^{st}\) one to the \(m-2^{nd}\) one, \ldots, from the \(2^{nd}\) one to the \(1^{st}\) one) in order to express the error of each neuron \(\delta_{js}, s=1,\ldots,m\) by means of the global errors \(\delta_{jm}^*\) of the entire network.

Let us use the following notations. Let \(w_{js}^i\) be the weight corresponding to the \(i^{th}\) input of the \(j^{th}\) neuron (\(j^{th}\) neuron of the \(s^{th}\) layer), \(Y_{js}\) be the actual output of the \(j^{th}\) neuron from the \(s^{th}\) layer (\(j=1,\ldots,m\)), and \(N_j\) be the number of the neurons in the \(s^{th}\) layer. It means that the neurons from the \(s+1^{st}\) layer have exactly \(N_j\) inputs. Let \(x_1,\ldots,x_n\) be the network inputs. The backpropagation learning algorithm for MLMVN is constructed in the following way [5], [8].

The global errors of the entire network are determined by (5). We have to distinguish the global error of the network \(\delta_{jm}^*\) from the local errors \(\delta_{jm}\) of the particular output layer neurons. The local errors are represented in the following way. The errors of the \(m^{th}\) (output) layer neurons are:

\[
\delta_{jm} = \frac{1}{t_m} \delta_{jm}^*, \tag{6}
\]

where \(jm\) specifies the \(j^{th}\) neuron of the \(m^{th}\) layer; \(t_m = N_{m-1} + 1\), i.e. the number of all neurons in the preceding layer (layer \(m-1\) where the error is backpropagated to) incremented by 1. The errors of the hidden layers neurons are computed as follows:

\[
\delta_{js} = \frac{1}{t_s} \sum_{j=1}^{N_{s-1}} \delta_{js+1}(w_{js+1})^{-1}, \tag{7}
\]

where \(js\) specifies the \(j^{th}\) neuron of the \(s^{th}\) layer (\(j=1,\ldots,m-1\)); \(t_s = N_{s-1} + 1\), \(s = 2,\ldots,m\) is the number of all neurons in the layer \(s-1\) (the layer where the error is backpropagated to, or in other words the number of inputs of the \(j^{th}\) neuron from the \(s^{th}\) layer) incremented by 1, and \(t_1 = n + 1\) (\(n\) is the number of network inputs).

The weights for all neurons of the network can be corrected after calculation of the errors. This correction can be done using the error-correction learning rules (3) or (4) adapted to MLMVN. There are the following learning rules presented in [5], [8] and slightly modified in [12]:

\[
\tilde{w}_{jm}^{\text{in}} = w_{jm}^{\text{in}} + \delta_{jm} \bar{Y}_{m-1}, \quad i = 1,\ldots,n, \tag{8}
\]

\[
\tilde{w}_{jm}^{\text{out}} = w_{jm}^{\text{out}} + \delta_{jm}, \tag{9}
\]

for the neurons from the \(m^{th}\) (output) layer (\(j^{th}\) neuron of the \(m^{th}\) layer),

\[
\tilde{w}_{js}^{\text{in}} = w_{js}^{\text{in}} + \frac{1}{|z_{js}|} \delta_{js} \tilde{Y}_{s-1}, \quad i = 1,\ldots,n, \tag{10}
\]

\[
\tilde{w}_{js}^{\text{out}} = w_{js}^{\text{out}} + \frac{1}{|z_{js}|} \delta_{js},
\]

for the neurons from the \(j^{th}\) neuron of the \(s^{th}\) layer (\(s=2,\ldots,m-1\)), and

\[
\tilde{w}_{j1}^{\text{in}} = w_{j1}^{\text{in}} + \frac{1}{|z_{j1}|} \delta_{j1} \tilde{x}, \quad i = 1,\ldots,n, \tag{11}
\]

\[
\tilde{w}_{j1}^{\text{out}} = w_{j1}^{\text{out}} + \frac{1}{|z_{j1}|} \delta_{j1},
\]

for the neurons of the \(1^{st}\) hidden layer.
3 CUDA-based MLMVN simulator

The most efficient, speedy use of CUDA parallel architectures across operations comes when computation is organized both into concurrent executing threads and into congruent and simultaneous operations whose evaluation may be started and completed in unison to prevent underuse of those synchronized computing elements when divergent execution paths must be serially executed [9]. The MLMVN learning algorithm was adapted to this end from the precepts given in [5], [8], beginning with its representation of the MVN’s activity mapping an input pattern vector \( X \) to a single output (see (1)). If a trivial element \( x_0 = 1 \) containing the multiplicative identity is added to the input pattern vector \( X \), then (1) can be modified as

\[
 f(x_0, x_1, \ldots, x_n) = P(w_0x_0 + w_1x_1 + \ldots + w_nx_n),
\]

where the special case of the weight \( w_0 \) is eliminated and all terms have a congruent form. This approach not only allows the output of a single MVN to be computed as the dot product \( z \) of vectors \( W \) and \( X \), to which the activation function \( P \) is applied, but is easily extended to produce the output of a serial layer of MVNs in the form of vector-column \( T \):

\[
 T = P(WX)
\]

where \( X \) is a vector-column containing the layer inputs (which are the same for all the neurons in a layer) and the matrix \( W \) is populated with the weights of the corresponding neurons (the 1st row contains the weights of the 1st neuron in the layer, etc.). Computations of this form are well supported by linear algebra libraries descended from BLAS [13] and in the case of the CUDA architecture by CUBLAS [9]. In a multilayer neural network, the output signal of neurons from the layer \( s \) are given by

\[
 T_s = P(W_sX_s)
\]

where \( Y_s = T_{s-1} \; s \geq 1 \), is a vector-column of the inputs of the \( s \)th layer neurons, which equal to the outputs of the preceding layer neurons, \( Y_0 \) is the prepended input pattern vector \( X = (1, x_0, \ldots, x_n) \), and \( W_s \) is a matrix containing in its rows the weights of the \( s \)th layer neurons.

To offset the proprietary nature of the CUDA architecture, we chose to supplement the manufacturer-supplied CUDA C environment with C++ for ease of portability and existing library support when constructing our new, parallelized MLMVN simulator. The desire for double precision floating point operations restricted us to using the Fermi hardware revision [14] and proceeded with GeForce GTX 450 and 550 Ti cards under Windows 7 and CUDA 4.0.

Our original approach involved replacing the serial vector and matrix operations with CUBLAS library calls, but this proved unduly burdened by transfer time overhead between the separate CPU host and CUDA device memory spaces. A process of revisions arrived at the “ship in a bottle” technique, an optimum model of fixed length structures and large, single-dimension arrays to allow infrequent, monolithic transfers into GPU device memory, where smaller, higher dimension, and variable length data structures are logically re-erected like the eponymous vessel’s rigging. For example, all weights for all layers were stored in a single, 2048-member array; the \( s \)th layer’s \( j \)th neuron’s \( i \)th weight would be found at

\[
 i + j \times \text{SignalQTY}[s]
\]

where \( \text{WtOffset} \) stores the index of the array element containing the first weight for a given layer \( s \) and \( \text{SignalQTY} \) is the number of inputs in the vector arriving from the learning set or previous layer \( s-1 \).

Our parallel simulation is twofold: serial procedures executed on the CPU are used to benchmark the parallel versions executed on the Fermi card GPUs. Where it is possible, identical code were used and independent operations performed in parallel with indexed thread assignment replaced conventional FOR loops as follows

```c
long INROWLEN=Net.iNeuronQTY[0];
for (int i=0; i<INROWLEN; ++i){
    Signals[Net.iNeuronOfst[i]+i]= Xinputs[IDX2C( i, s, INROWLEN )];
}
```

This proved to be very powerful for evaluation use, with the time to feed forward 10,000 samples scaling up only 20-30% from the time taken by a single network evaluation. Backpropagation proved less tractable, with simple parallelism limited to the magnitude of input vectors for each layer. RMSE calculation had medial improvement by making the deviation measurement and squaring part of the evaluation procedure, with summation being implemented as an array reduction operation in \( O(\log_2 n) \) time.

4 Edge Detection: Simulation Results

To use MLMVN for edge detection and enhancement, we have to train the network using a learning set. After the learning process is completed, MLMVN can process those images, which have not participated in the learning process. To process an image using MLMVN, it is necessary to perform this transformation, we have to choose some
\(k > m\), and then the intensity value \(j \in \{0,1,\ldots,m-1\}\) can be transformed to \(e^{j2\pi / k}\). It is important to choose \(k > m\), to avoid a collision, which may occur when the “black” intensity \(e^0 = 1\) and the “white” intensity \(e^{2\pi (m-1)/k}\) will be located on the unit circle too close to each other. In all our experiments with 8 bits/pixel grayscale images with the dynamic range 0,...,255 we used \(k = 384\). The same value of \(k\) was used in the activation function (2), respectively.

To prepare a learning set, we have to take two images. One of them is an image where we want to detect edges. Another one is its ideal edged image obtained with some edge detector. The learning samples are formed in the following way: We randomly select a pixel with the coordinates \((i,j)\) in the first (original) image. Then we take a 3x3 window around it, which forms 9 inputs for MLMVN, and the corresponding desired output from the \(j^{th}\) pixel of the ideal edged image

\[
\begin{pmatrix}
  x_{i-1,j-1} & x_{i-1,j} & x_{i-1,j+1} \\
  x_{i,j-1} & x_{i,j} & x_{i,j+1} \\
  x_{i+1,j-1} & x_{i+1,j} & x_{i+1,j+1}
\end{pmatrix} \rightarrow y_{ij} \tag{11}
\]

In all our experiments we used a single image-teacher (and the corresponding ideal edged image) to create a learning set with 10,000 learning samples. We used in our simulation a threshold Boolean filtering (TBF) edge detector introduced in [4].

A TBF detector employs edge detecting Boolean functions, which analyzes a 3x3 local window around each pixel of a binary image. The main property of edge detecting Boolean functions is that they differ between upward and downward brightness jumps. An upward brightness jump occurs when there is a “hill” in the center of the window (1), and there is at least one 0 value around it. A downward brightness jump occurs in the opposite situation – when there is a “gap” in the center of the window (0), and there is at least one 1 value around it. It was shown in [4] that TBF upward and downward edge detectors are equivalent to the following spatial domain edge detecting filters for downward brightness jumps

\[
y(i,j) = \begin{cases} 
  m = \max \{x(i,j) - x(i \pm 1, j \pm 1), \text{if} \ m > 0 \} \\
  0, \quad \text{otherwise}
\end{cases} \tag{12}
\]

and upward brightness jumps

\[
y(i,j) = \begin{cases} 
  m = \max \{x(i \pm 1, j \pm 1) - x(i,j), \text{if} \ m > 0 \} \\
  0, \quad \text{otherwise}
\end{cases} \tag{13}
\]

where \(i,j\) are the coordinates of the pixel of interest, \(x(i,j)\) is the intensity value in the pixel of interest, \(x(i \pm 1, j \pm 1)\) are the intensity values in the adjacent pixels in a 3x3 window, \(y(i,j)\) is the intensity value in the resulting edged image, which is equal to \(m\), if an edge is detected, and to 0, if an edge is not detected.

A great advantage of TBF edge detectors is that all edges are detected “as they are”. Their intensity equal to a maximal intensity jump detected in the corresponding pixel, and the “width” of an edge is always 1 pixel, because if there is an upward or downward brightness jump detected in some pixel, a brightness jump of the same type cannot exist in either of the adjacent pixels. The example of TBF edge detection is shown in Fig. 2.

To prove that MLMVN can learn how to detect edges according to a certain edge detection operator, we trained MLMVN to detect edges corresponding to TBF upward and TBF downward edge detectors. To create learning sets for both edge detectors according to (11), a single test image “Lena” (Fig. 3) was used. So, the network inputs were taken from “Lena”, while its desired outputs were taken from the edged image obtained from “Lena” with TBF determined by (12) for the first learning set, and with TBF determined by (13) for the second learning set, respectively.

We used MLMVN with 9-54-1 topology (9 inputs, 54 hidden neurons in a single hidden layer, and a single output neuron). It took 457 iterations to learn the TBF upward edge detection with the accuracy 3.39 (in terms of root mean square error – RMSE), and 102 iterations to learn the TBF downward edge detection with the RMSE accuracy 3.34.

Repeating learning process starting from the different random weights we got about the same number of iterations to reach the same accuracy. So this result is stable.

After the learning process was completed, we used MLMVN to detect edges on those images which did not participate in the learning process. The results for the “Boat” test image are shown in Fig. 4. There is practically no visual distinction between the images in Fig. 4a vs. Fig. 2b, and Fig. 4b vs. Fig. 2c. While the first images from these pairs were obtained using MLMVN trained with the “Lena” image, the second images from the same pairs were obtained in the straightforward way, with the corresponding edge detection operators. This experiment clearly confirms than MLMVN learns exactly that particular edge detection operator, which is used for its training. The distinction between those images created by MLMVN and the ideal edged images that were obtained in the straightforward way is mostly is caused by the level of edge enhancement. This leads to some accumulation of RMSE.

To detect “clean” edges on noisy images using MLMVN, it is possible to train the network to detect ideal edges on a noisy “teacher”-image. To create a learning set, we added zero-mean Gaussian noise with the variance 0.3\(\sigma\) (where \(\sigma\) is a variance of the original image) to the original image “Lena” and created a noisy “teacher”-image. This image was used to create teaching inputs, while the ideal edged images were used to create the corresponding teaching outputs according to (11). The learning sets containing 10,000 samples were created. Then the learning sessions were started.
Fig. 2 Edge detection using threshold Boolean filtering

(a) TBF upward edge detection on the clean "Boat" image (see Fig. 2a) using MLMVN trained on "Lena". PSNR=32.78, RSME=5.85

(b) TBF downward edge detection on the clean "Boat" image (see Fig. 2a) using MLMVN trained on "Lena". PSNR=32.93, RSME=5.76

Fig. 3. Image “Lena” used for learning

Fig. 4 Edge detection on a clean image with MLMVN
We used MLMVN with the following topologies 9-45-1, and 9-54-1. It was succeeded to drop the learning RMSE to 8.41 for the network 9-45-1 (60 iterations), while this network learned the TBF upward edge detection. It was also succeeded to drop the learning RMSE to 8.44 (5817 iterations) for the network 9-45-1, while this network learned the TBF downward edge detection.

Performance of the CUDA MLMVN simulator is characterized by the following processing times (see Table 1), which we got for 1 iteration with 10,000 learning samples (the Fermi hardware revision on a GeForce GTX 550 Ti card under Windows 7 and CUDA 4.0 was used to run the CUDA-based parallel simulator and the Pentium 4 650 2.93 GHz processor with the same OS and CUDA 4.0 was used to run a regular serial simulator).

<table>
<thead>
<tr>
<th>Type of simulator</th>
<th>Network 9-45-1</th>
<th>Network 9-54-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMSE Evaluation, ms</td>
<td>CUDA 3.5</td>
<td>Serial 720 4.1</td>
</tr>
<tr>
<td>Classification, ms</td>
<td>CUDA 3.7</td>
<td>Serial 600 4.3</td>
</tr>
<tr>
<td>Backpropagation, ms</td>
<td>CUDA 270</td>
<td>Serial 2700 220</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As it is seen from Table 1, the MLMVN CUDA simulator significantly outperforms a traditional serial simulator. The learning process with the CUDA MLMVN 9-45-1 simulator is about 10 times faster and the learning process with the CUDA MLMVN 9-54-1 simulator is about 13 times faster than the ones with a serial simulator. This comparison shows high advantages of CUDA-based MLMVN simulator over a regular serial MLMVN simulator.

5 Conclusions

In this paper, we have considered how the edge detection problem can be solved using a neural network. We employed CUDA simulator of the multilayer neural network based on multi-valued neurons to learn edge detection operators and then to detect edges corresponding to these operators. The most important conclusions are the following: 1) The MLMVN CUDA simulator is a highly efficient tool for speeding up learning processes where a large neural network and thousands learning samples are involved; 2) MLMVN can learn different edge detection operators from a single example and then it can process those images that did not participate in the learning process detecting edges specifically corresponding to the learned operator with a high accuracy.

6 References