GPU Simulator of Multilayer Neural Network Based on Multi-Valued Neurons

Christian Hacker
Texas A&M University-Texarkana
Texarkana, TX, USA
christian.hacker@ace.tamut.edu

Igor Aizenberg
Manhattan College
Riverdale, NY, USA
igor.aizenberg@manhattan.edu

Jeff Wilson
Dallas, TX, USA
jwilson@clueland.com

***Abstract*— In this paper, we consider principles of design and basic fundamentals of a GPU simulator of the multilayer neural network with multi-valued neurons (MLMVN). Slowing down a learning process due to a big learning dataset and/or a big neural network needed for solving a certain problem is a potential bottleneck preventing the use of neural networks for solving some challenging problems. The same is related to deep learning. MLMVN is a feedforward complex-valued neural network, which has a number of advantages when compared to real-valued neural networks. These advantages include derivative-free learning and significantly better generalization capability. To extend applicability of MLMVN, its GPU-based software implementation shall be considered. We present basic principles of the GPU simulator of MLMVN and how matrix algebra operations are specifically employed there. It is shown that the bigger the network is, the more beneficial is its GPU implementation. It is shown that up to 32x acceleration can be achieved for the MLMVN learning process. Some applications, which could not be even considered without a GPU simulator, are also presented.**

Keywords*— *Complex-Valued Neural Networks, Multi-Valued Neuron, Multilayer Neural Network with Multi-Valued Neurons, MLMVN, Intelligent Filtering

I. Introduction

In this paper, we will consider basic principles, organization and applications of a GPU-based simulator of the multilayer neural network with multi-valued neurons (MLMVN). MLMVN is a complex-valued feedforward neural network that is a valuable member of the rapidly developing complex-valued neural networks (CVNN) family. A good review of the state of the art in the CVNN area is given, for example, in [1]-[3]. CVNNs have successfully been used for solving various real-world problems. Just a few of them are, for example, landmine detection [4], forecasting of wind profiles [5], and medical image analysis [6], forecasting of oil wells productivity [7], and decoding of signals in EEG-based brain-computer interfaces [8]. We also would like to mention some other contributions. In [9] and [10], a classical backpropagation learning algorithm for the complex-valued case was generalized. In [11], another approach to complex domain error backpropagation was considered. A new optimization technique for learning in complex domain was developed in [12]. Solving classification problems using various complex-valued neural networks was recently considered in [13]-[15].

MLMVN was first suggested in [16] and then presented in detail in [17]. MLMVN has a traditional feedforward topology where neurons are grouped in layers, and outputs of all neurons in a hidden layer are connected to the corresponding inputs of all neurons in the following layer. All specific properties of MLMVN and its advantages over a classical feedforward neural network with sigmoidal neurons are determined by the use of the multi-valued neuron (MVN) as its basic unit.

MVN is the first historically known compex-valued neuron. It was initially called "an element of multiple-valued threshold logic", which was introduced by N.N.Aizenberg *et al*. in 1973 in [18]. It was re-introduced as a discrete MVN in [19]. While MVN weights can be arbitrary complex numbers, its inputs and output are located on the unit circle. Discrete MVN inputs/outputs are the exact k^{th} roots of unity (where k is a positive integer). Continuous MVN inputs/outputs can be arbitrary numbers located on the unit circle. It follows from these properties of both discrete and continuous MVN outputs that both discrete and continuous MVN activation functions depend only on the argument (phase) of the neuron's weighted sum. They map the set of complex numbers onto the set of k^{th} roots of unity (the discrete case) or onto the unit circle (the continuous case).

A comprehensive study of MVN and MLMVN, their error-correction learning, and other properties are presented in detail in [3]. One of the most important properties and advantages of both a single MVN and MLMVN is their learning algorithm, which is based on the error-correction learning rule. This learning rule is a complex-valued generalization of the classical F. Rosenblatt's error-correction learning rule [20]. Thus, MVN and MLMVN error-correction learning is derivative-free.

MLMVN was successfully employed in many applications. It was used, for example, for type of blur and blur parameters recognition in image deblurring [21], prediction of component reliability and level of degradation of the railroad equipment [22], long term forecasting of oil wells productivity [7] and long term financial time series prediction [3], decoding of signals in EEG-based brain-computer interfaces [8], satellite data inversion for determination of meteorological data profiles in the atmosphere [23], and solving various classification problems [3], [17], [24]. To improve MLMVN generalization capability when solving classification problems, a learning algorithm with soft margins was introduced in [24].

However, there is a common bottleneck which reduces productivity of any traditional Central Processor Unit (CPU)-based (serial) software simulator of a neural network. Any learning process is iterative and it is time consuming. The larger a network is (the more hidden neurons are employed)

978-1-5090-0620-5/16/$31.00

and the bigger a learning set is (the more learning samples are presented there), the more significant time is needed to train the network. This can be especially sensitive for any deep learning applications and any intelligent image processing applications where it is desirable to use learning sets containing thousands of learning samples. The best way to speed up the learning process is to create a software simulator which allows parallelization. In fact, error backpropagation and correction of the weights in a multilayer feedforward neural network can be done in parallel for neurons from the same layer. The most suitable way for the creation of such a parallel simulator is to use a Graphics Processing Unit (GPU) as an alternative to CPU, and CUDA [25], a parallel computer architecture developed by NVIDIA. Several efficient CUDA-based simulators were developed for the classical multilayer perceptron (e.g., [27], [28] should be mentioned). The first attempt to design a GPU simulator for MLMVN was done in [29]. However, that simulator, being a good step ahead, had a limited applicability because it was not suitable for simulation of a really big neural network containing thousands of neurons.

In this paper we present a general approach for design of a GPU-based MLMVN simulator, which really makes it possible to speed up both a learning process (up to 32x times as it follows from our experiments) and data processing (as it will be seen from the intelligent filtering application). Our simulator is designed in Python, using the PyCUDA [32] library. The structure of the paper is as follows. Some MVN and MLMVN fundametals are briefly recalled for the reader's convenience in Section II. Then our approach to design a GPU-based MLMVN simulator is presented in Section III. Simulation results are presented in Section IV. Conclusions and further directions of this work are given in Section V.

II. MVN AND MLMVN

Let us recall very briefly some MVN and MLMVN fundamentals for the reader's convenience.

A. MVN

MVN is a neuron with n inputs and one output, all located on the unit circle. The discrete MVN was proposed in [19]. A discrete MVN input/output mapping is described by a function of n variables $f(x_1, ..., x_n)$, which is either a function $f: E_k^n \to E_k$ or a function $f: O^n \to E_k$ (where $E_k = \{1, \varepsilon_k, \varepsilon_k^2, ..., \varepsilon_k^{k-1}\}$ is the set of the k^{th} roots of unity, $\varepsilon_k = e^{i2\pi/k}$ is the primitive k^{th} root of unity, i is the imaginary unit, k is a positive integer determining k-valued logic, and O is a set of points located on the unit circle). This function is a threshold function of k-valued logic [3] and therefore it can be represented using n+1 complex-valued weights as follows [3]

$$f(x_1, ..., x_n) = P(w_0 + w_1 x_1 + ... + w_n x_n) \qquad (1)$$

where $x_1, ..., x_n$ ($x_j \in E_k, j = 1,...,n$) are the neuron inputs and $w_0, w_1, ..., w_n$ are the weights. P is the activation function of the neuron:

$$P(z) = \varepsilon_k^j = e^{i2\pi j/k}, \text{ if } 2\pi j/k \le \arg z < 2\pi(j+1)/k, \qquad (2)$$

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

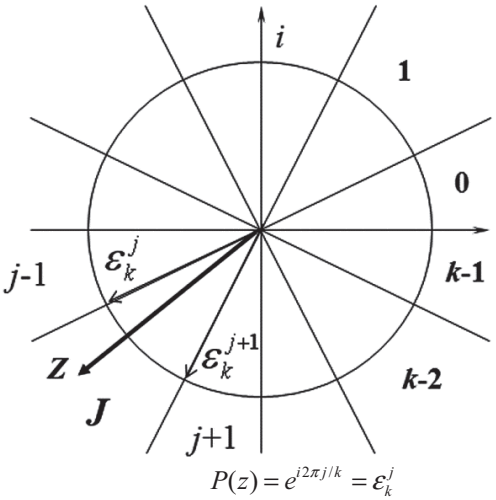

Fig. 1. Geometrical interpretation of the discrete MVN activation function

Function (2), which was suggested in the seminal paper [30], divides the complex plane into k equal sectors and maps the whole complex plane onto the set E_k of the k^{th} roots of unity (see Fig. 1). If the weighted sum is located in the sector j, then the neuron's output is ε_k^j.

When an MVN input/output mapping is described by a function $f: O^n \to E_k$, the neuron inputs are not necessarily k^{th} roots of unity, but rather arbitrary points located on the unit circle. It is important to mention that if there is a real-valued input/output mapping defined on the bounded subdomain $D^n \subset \mathbb{R}^n$ and $f(y_1, ..., y_n): D^n \to K = \{0,1,...,k-1\}$, where $y_j \in [a_j, b_j], a_j, b_j \in \mathbb{R}, j = 1,...,n$, then it can be easily transformed to $f: O^n \to E_k$ by a linear transformation applied to each $y_j, j = 1,...,n$:

$$y_j \in [a_j, b_j] \Rightarrow \varphi_j = \frac{y_j - a_j}{b_j - a_j}\alpha \in [0, 2\pi[, \quad j = 1,...,n; 0 < \alpha < 2\pi, \qquad (3)$$

and then $x_j = e^{i\varphi_j} \in O, j = 1,2,...,n$ represents a complex number located on the unit circle.

The MVN error-correction learning rule is presented and analyzed in detail in [3], where a modified proof of the convergence of the learning algorithm has also been presented (some remarks and minor corrections to this proof are given in [36]).

The continuous MVN performs an input/output mapping described by a function $f: O^n \to O$. The continuous MVN activation function is $P(z) = e^{iArg(z)} = z/|z|$.

The weights adjustment for both discrete and continuous MVN is completely determined by the neuron's error, which is the arithmetic difference $\delta = D - Y$ between the complex numbers D (a desired output) and Y (an actual output) located on the unit circle.

The error-correction learning rule is [3]

$$W_{r+1} = W_r + \frac{C_r}{(n+1)}\delta \bar{X}, \tag{4}$$

and the rule with a self-adaptive learning rate is

$$W_{r+1} = W_r + \frac{C_r}{(n+1)|z_r|}\delta \bar{X}, \tag{5}$$

where $\bar{X}$ is the input vector with the components complex-conjugated, n is the number of neuron inputs, r is the number of the learning step, W_r is the current weighting vector (to be corrected), W_{r+1} is the updated weighting vector (after correction), C_r is the constant part of the learning rate (it should be complex-valued in general, but in all known applications, including those considered in this paper, it was set equal to 1), and $|z_r|$ is the absolute value of the weighted sum obtained on the r^{th} learning step (self-adaptive part of the learning rate).

It should also be mentioned that it might be reasonable to calculate the error not as the difference between the desired ($D = \varepsilon_k^q$) and actual ($Y = \varepsilon_k^s$) outputs, but as the difference between the desired output D and the projection $z/|z|$ of the current weighted sum z onto the unit circle. Evidently, in such case $\delta = D - z/|z|$. This is useful, for example, to learn highly nonlinear input/output mappings.

B. MLMVN

MLMVN has a traditional feedforward topology.

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

$$\delta_{jm}^* = D_{jm} - Y_{jm}; j = 1,...,N_m. \tag{6}$$

The MLMVN learning algorithm is derived ([3], [17]) from the consideration that the global error of the network depends on the local errors of all neurons and therefore must be shared among all neurons, since all of them contribute to this error by their local errors.

The backpropagation of the global errors δ_{jm}^* through the network is used to express the error of each neuron $\delta_{jr}, r = 1,...,m$ by means of the global errors δ_{jm}^* of the entire network.

Let us use the following notations. Let w_i^{jr} be the weight corresponding to the i^{th} input of the jr^{th} neuron (j^{th} neuron from the r^{th} layer), Y_{jr} be the actual output of the j^{th} neuron from the r^{th} layer (r=1,…,m), and N_r be the number of neurons in the r^{th} layer. Subsequently, the neurons from the r+1$^{\text{st}}$ layer have exactly N_r inputs (r=1,…,m). Let $x_1,...,x_n$ be the network inputs.

The global errors of the entire network are determined by (6). To obtain the local errors for all neurons, the global error must be shared with these neurons through the backpropagation process. This process works as follows. The errors of the output layer neurons are:

$$\delta_{jm} = \frac{1}{t_m}\delta_{jm}^*; j = 1,...,N_m, \tag{7}$$

where jm specifies the j^{th} neuron of the m^{th} (output) layer; $t_m = N_{m-1} + 1$, i.e. the number of all neurons in the preceding layer (layer m-1 where the error (7) will be then backpropagated to) incremented by 1.

Then the errors of the hidden layers neurons are

$$\delta_{ir} = \frac{1}{t_r}\sum_{j=1}^{N_{r+1}} \delta_{j,r+1}(w_i^{j,r+1})^{-1}, \tag{8}$$

where ir specifies the i^{th} neuron of the r^{th} layer (r=1,…,m-1); $t_r = N_{r-1} + 1, \quad r = 2,...,m, \quad t_1 = n + 1$ (n is the number of network inputs) is the number of neurons in layer r-1 (or equivalently the number of inputs of the j^{th} neuron from the r^{th} layer) incremented by 1. Equations (7)-(8) determine the MLMVN error backpropagation.

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 (4) for hidden neurons and (5) for output neurons adapted to MLMVN.

The standard MLMVN learning algorithm consists of the sequential checking, for all learning samples, whether the actual output of the network coincides with the desired output. If for some sample there is no coincidence, then the errors (6) must be backpropagated according to (7)-(8), and the weights must be adjusted. The learning process continues either until the zero-error is reached or the root mean square error (RMSE) criterion is satisfied.

For MLMVN with discrete output neurons, RMSE should be applied to the errors in terms of the numbers of sectors (see Fig. 1), thus not to the elements of the set $E_k = \{\varepsilon_k^0, \varepsilon_k, ..., \varepsilon_k^{k-1}\}$, but to the elements of the set $K = \{0,1,...,k-1\}$. The local errors for the r^{th} learning sample in these terms are calculated as

$$\gamma_r = (\alpha_{j_r} - \alpha_r) \bmod k; \alpha_{j_r}, \alpha_r \in \{0,1,...,k-1\} \quad (9)$$

($\varepsilon_k^{\alpha_{jr}}$ is the desired output and $\varepsilon_k^{\alpha_r}$ is the actual output). For MLMVN with continuous output neurons, error (9) is calculated simply as an absolute angular difference between the arguments of the desired and actual outputs.

Let Δ_r be the square error of the network for the r^{th} learning sample (r=1,…, M, hence the learning set contains M learning samples). For MLMVN with a single output neuron it is

$$\Delta_r = \gamma_r^2, r = 1,...,M \quad (10)$$

(γ_r is the local error taken from (9)), and for MLMVN with N_m output neurons (m is the output layer index) it is

$$\Delta_r = \frac{1}{N_m}\sum_{j=1}^{N_m}(\gamma_{jr})^2; r = 1,...,M\ . \quad (11)$$

The MLMVN learning process (under RMSE criterion) continues until RMSE drops below some pre-determined acceptable minimal value λ:

$$RMSE = \sqrt{\frac{1}{M}\sum_{r=1}^{M}\Delta_r} \le \lambda\ . \quad (12)$$

The convergence of this learning algorithm is proven in [3] (some remarks and minor corrections to this proof are given in [36]).

III. GPU-BASED MLMVN SIMULATOR AND ITS DESIGN

A. Basic Considerations

In neural networks, the weights for individual neurons are treated as values within a matrix - this allows simulation software to manipulate an entire network exclusively through linear algebra routines. The run-time of a simulation is proportional to the size of a network; a large network will run slowly on a CPU due to its inherently sequential architecture and limited number of processing cores. Specifically, CPU-bound linear algebra operations on large weight matrices quickly becomes intractable and requires numerous memory transfers. This issue is only exacerbated for the MLMVN, which must iteratively operate on every input/output mapping within its learning set and process twice as much data as a real-valued network. A CPU-bound MLMVN is thus limited to small-scale simulations and otherwise would invoke run-times of several weeks. Because we desire to apply the MLMVN paradigm to more complicated problems with large networks or learning sets, it is necessary to develop a non-serial, or parallel, version of the simulator software.

General-purpose GPUs have rapidly gained popularity as a solution to serially-intractable problems, including large-scale neural networks. By dividing a single operation into several smaller ones that can be executed simultaneously across hundreds (or thousands!) of processing cores, a GPU can theoretically out-perform a CPU on a similar task. The two most popular GPU frameworks, OpenCL [26] and CUDA [25], allow users to write a single instruction, or kernel, which is then executed on every core over a portion of some pertinent data. Such a process is called single instruction, multiple data (SIMD). As an overtly-simple example, an element-wise matrix multiplication kernel might specify how to multiply a single element in a matrix by a scalar; each processing core in the GPU may then apply this kernel to a different element within the matrix. Kernel application happens in tandem over the available cores, whereas a CPU processor would need to evaluate the instruction iteratively. This doesn't mean that GPU parallelism is a panacea for slow-running algorithms; problems that aren't 'embarrassingly parallel' or that require frequent memory transfers between the GPU (device) and CPU (host) memory may result in poorer performance than a strictly CPU-bound program. Finding an acceptable compromise between speed and memory optimization is a standing issue in SIMD programming, and designing efficient kernels can be a daunting task.

To mitigate these problems, several resources exist which provide convenient GPU utilities. The CUBLAS library [33] re-implements the BLAS linear algebra routines for NVIDIA's CUDA cards, and the PyCUDA & PyOpenCL [34] modules give high and low-level control of GPUs for the Python interpreted programming language. The latter two resources are particularly interesting due to their reliance on Run-Time Code Generation (RTCG) and metaprogramming. In short, a GPU kernel is automatically optimized prior to compilation based on the chosen hardware and program parameters, and it is only compiled when first called by the interpreter. Furthermore, these modules offer a view-based interface for GPU arrays that is almost identical in functionality to NumPy [31], the de facto standard for CPU-bound linear algebra and matrix manipulation in Python. This gives users the best of both worlds: a high-level interpreted language and a powerful, massively-parallel system for crunching numbers. A programmer can choose to let the PyCUDA/PyOpenCL libraries handle optimizing kernels and managing GPU memory - with normally negligible performance tradeoffs. The advantages and disadvantages of autonomous GPU programming may be observed in our own attempts to implement a GPU-bound MLMVN simulator.

To make large-scale MLMVN experiments possible, it is necessary to leverage the parallelism of GPUs. Our first successful attempt [29] relied on custom kernels designed for NVIDIA's Fermi architecture, written in C++ and CUDA-C. By constraining our software to only use the fast cache and texture memory of a GPU, we attained a dramatic speedup

factor of ~195 for the classification phase of the MLMVN learning algorithm, with a network topology of [9, 54, 1]. This method was limited by the relatively small size of the cache memory, resulting in a cap on the total number of neurons in the network. To address this constraint, here we use PyCUDA and the derivative scikit-cuda [35] libraries to provide optimized, deterministically-generated kernels and automatic memory management. While enabling the MLMVN simulator to utilize significantly more of GPU memory - and thus allowing larger network topologies - PyCUDA [32] introduces some new caveats, the consequences of which are elucidated in our simulation results.

B. Implementation

The output of a single MVN as it follows from (1), is nothing else as the dot product of vectors W and $\bar{X}$, to which an activation function P is applied. Let us use the same approach, which was used in the earlier work [29]. This approach is expanded to calculate all outputs within a given layer of MLMVN as a vector-column:

$$T_s = P\,(W_s \mathbf{Y}_s)$$

where $\mathbf{Y}_s = \mathbf{T}_{s-1}$ $s \geq 1$, is a vector-column of the inputs of the sth layer neurons, which equal to the outputs of the preceding layer neurons, $\mathbf{Y}_0$ is the prepended input pattern vector $\mathbf{X} = (1, x_0, ..., x_n)$, and W_s is a weights matrix for network layer s.

Our previous approach for storing network weights and other variables involved the allocation of large one-dimensional arrays in device memory, within which data structures such as weight arrays were stored and logically reconstructed when needed. In its current Python implementation, the MLMVN simulator declares all required data structures individually and tasks PyCUDA with managing them efficiently. When the Python interpreter encounters CUDA linear algebra routines in the learning algorithm, the appropriate RTCG kernel is invoked and pertinent data within device memory is accessed. Thus, the Python interpreter is responsible for all GPU function calls. While this implies that the simulator is indeed still CPU-bound, the most computationally-costly operations, namely those involving large matrices - are handled by the GPU. It would be more accurate to describe this method as a GPU-accelerated simulator as opposed to a GPU-bound one, but it nonetheless delivers considerable performance gains for large network topologies.

IV. Simulation Results

A. Benchmarking Topologies

To evaluate the potential performance gains of the GPGPU-accelerated MLMVN simulator, we tested 55 unique network configurations and compared their CPU and GPGPU-accelerated run-times. Topology used the form [S, NxL, 1], where S is the number of inputs neurons (i.e. number of learning samples), N is the number of neurons per hidden layer, and L is the number of hidden layers. For example, if N = 500 and L = 3, the tested topology would be [75, 500, 500, 500, 1]. Networks as small as [75, 5, 1] and as large as [75, 10000x5, 1] were trained on the 75-sample "Iris" learning set [37] for a maximum of 10 iterations; the elapsed run-times were used to find the mean run-time per iteration. CPU and GPGPU simulations were handled by a dual-core Intel i7-620M processor and a Tesla C2075 card, respectively. Fig. 2 shows the ratio of CPU vs. GPGPU mean iteration run-time as a logarithmic function of both N and L. Immediately apparent is the disparity between GPGPU performance for very small and very large topologies - while the former group produces abysmally long run-times on a GPGPU, networks with higher N and L values showed a speedup factor of up to 32x. Performance gains decrease slightly for the largest networks tested, likely due to limitations in the memory capacity of the test machines or in the simulation software itself. Granted, these are modest gains compared to our previous CUDA-based simulator, but they were achieved for network sizes in excess of 30,000 neurons as opposed to a 64-neuron network. The substantial variation in our results is likely caused by PyCUDA functions under-utilizing GPGPU resources for smaller network topologies, as well as some inefficient device-to-user memory transfers in the learning algorithm.

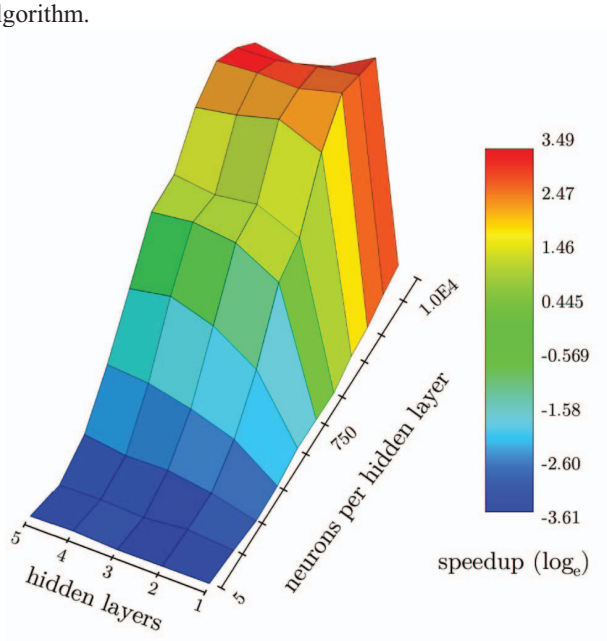

Fig. 2. Ratio of CPU vs. GPGPU mean iteration run-time

B. Image Filtering with MLMVN

A practical application of the GPGPU-accelerated MLMVN is based on the approach to intelligent image filtering presented in [38]-[40]. This approach is based on learning, using a neural network, how to create a clean patch from a noisy one using the large number of patches taken from many different images. Then a trained network is expected to filter patches from other images, which were not presented in

a learning set. Hence we need to create a learning set from a number of n x n patches randomly taken from different images. Noisy patches will be used as inputs (therefore a network will have n^2 inputs) and the corresponding clean patches will be used as desired outputs (therefore the network will have n^2 outputs). Then we need to train a neural network. After a network is trained, it should be used for filtering an image through dividing it into n x n overlapping patches, filtering all pixels in each patch simultaneously and averaging then the results for overlapping parts of patches. Our learning set is generated from a greyscale image database (100 different images) [41], with which a MLMVN simulator is then trained. The learning set is constructed by randomly selecting a pixel 'patch' from a specific image, distorting the patch with additive Gaussian noise of standard deviation 0.2σ where σ is a standard deviation of the image, and storing the resulting 'noisy' and 'clean' patches as a row-wise input/output vector of length n^2. This process is repeated several times for each image in the database (starting coordinates of patches are generated as random numbers), and the results are compiled into a learning set. We employ MLMVN whose hidden neurons have a continuous activation function, while output neurons have the discrete activation function (2) with $k = 288$. This is important to avoid closeness of the "white" and "black" parts of the intensity range $I = \{0,1,...,255\}$ on the unit circle. This value was chosen experimentally. While, for example, for $k = 264$ and $k = 384$ a learning process goes much slower, for $k = 288$ it goes much faster. MLMVN is then trained to find a mapping between the noisy and corresponding clean patches. Our filtering experiments used $n = 15$ (thus 15x15 patches), and had 7000 to 10000 samples in a learning set. The training results and tested topologies are listed in Table 1. Best minimized RMSE for each network calculated according to (12) was found through trial-and error, as a free-run simulation would often 'bottom out' at a certain RMSE and not improve beyond it.

TABLE I
TRAINING RESULTS FOR VARYING TOPOLOGIES & SAMPLE SETS

Topology	Samples	Iterations	Learning RMSE
[225, 4096, 225]	7000	1750	3.836
[225, 4096, 225]	10000	500	5.590
[225, 512, 4096, 225]	7000	300	8.234
[225, 512, 4096, 225]	10000	300	9.574
[225, 4096x4, 225]	7000	309	5.993
[225, 4096x4, 225]	10000	400	6.621

The benefits of the GPU-accelerated simulations were readily apparent: for example, for a topology of [225, 512, 4096, 225] and a 10000-sample learning set, it took ~23 hours to complete 300 iterations. Conversely, it was estimated that our CPU simulator would require approximately 222 hours for the same task - nearly a tenfold increase.

After training, we tested the MLMVN's filtering ability with 5 noisy images omitted from the learning set. As before, Gaussian noise with standard deviation 0.2σ, where σ is a standard deviation of the image, was added to the target images; these noisy versions were then passed patch-wise to the trained network as vectorized learning samples with dummy output values. We used overlapped patches with offset 1, then averaged the filtering result for each pixel over all overlapped patches it belonged to. The network outputs were then used to reconstruct the filtered image, which was statistically compared to the original via Peak Signal-to-Noise Ratio (PSNR). Fig. 3 thru Fig. 8 show characteristic filtering results for the three largest topologies. Interestingly, the best results attained with the [225, 512, 4096, 225] network (10,000 learning samples) had the worst RMSE value during training (see Table 1), while the largest network [225, 4096x4, 225] effectively removed noise but normalized extreme pixel intensities, resulting in a 'washed-out' appearance and the overall poorest results. A correlation between learning set size and filtering quality consistent with observations in [39]-[40] was seen as well. As for the quality of filtering in terms of PSNR, the best result is shown with the network [225, 512, 4096, 225]: the average PSNR over 5 test images is 32.76 (from 30.86 for the image shown in Fig. 8 to 34.62), which is comparable to or even better than the one gotten using order-statistic filters and only slightly yields to BM3D filter [42], commonly recognized as the best nonlinear filter.

V. DISCUSSION AND CONCLUSIONS

MLMVN is a powerful tool for machine learning; its use of MVNs with complex-valued weights allows it to achieve classification accuracy that rivals and often exceeds that of many other machine learning methods. Furthermore, MLMVN requires fewer total neurons and, typically, fewer iterations of its learning algorithm to minimize a learning error for the network. However, a reliance on a serial learning algorithm and complex-valued numbers introduces a significant increase in simulation run-time, especially for large network topologies and big learning sets. By utilizing the massive-parallelism of GPGPUs, we can arbitrarily scale our simulations without invoking unfeasible run-times. Benchmark tests demonstrate that the GPGPU-accelerated simulator functions optimally with sufficiently-large networks that can efficiently utilize computing resources. Some practical applications for MLMVN are shown, including the first-known example of image denoising with different MLMVN topologies (including a quasi-deep topology [225, 4096x4, 225]). We intend to further apply this model to image recognition problems, which may employ deep learning as well as increase its potential utility for the scientific community.

REFERENCES

[1] A. Hirose, *Complex-Valued Neural Networks, 2nd Edn.*, Springer, Berlin, Heidelberg, 2012.

[2] D. Mandic and V. Su Lee Goh, *Complex Valued Nonlinear Adaptive Filters: Noncircularity, Widely Linear and Neural Models*, John Wiley & Sons, 2009.

[3] I. Aizenberg, I., *Complex-Valued Neural Networks with Multi-Valued Neurons*. Berlin: Springer-Verlag Publishers, 2011.

[4] Y.Nakano, A.Hirose, “Improvement of Plastic Landmine Visualization Performance by use of ring-CSOM and Frequency-Domain Local Correlation,” *IEICE Transactions on Electronics*, vol. E92-C, Issue 1, Jan. 2009, pp. 102-108.

[5] S. L. Goh, M. Chen, D. H. Popovic, K. Aihara, D. Obradovic and D. P. Mandic, "Complex Valued Forecasting of Wind Profile," *Renewable Energy*, vol. 31 , Sep. 2006, pp. 1733-1750.

[6] A. Handayani, A.B.Suksmono, T.L.R.Mengko, and A.Hirose, “Blood Vessel Segmentation in Complex-Valued Magnetic Resonance Images with Snake Active Contour Model,” *International Journal of E-Health and Medical Communications*, vol. 1, issue 1, Jan. 2010, pp. 41-52.

[7] Aizenberg I., Sheremetov L., Villa-Vargas L., and Martinez-Muñoz J., "Multilayer Neural Network with Multi-Valued Neurons in Time Series Forecasting of Oil Production", *Neurcomputing*, vol. 175, part B, pp. 980-989, Jan. 2016.

[8] N.V.Manyakov, I. Aizenberg, N. Chumerin, and M. Van Hulle, “Phase-Coded Brain-Computer Interface Based on MLMVN”, book chapter in *Complex-Valued Neural Networks: Advances and Applications* (A. Hirose – Ed.), Wiley, 2012, pp. 185-208.

[9] H. Leung and S. Haykin, "The Complex Backpropagation Algorithm", *IEEE Transactions on Signal Processing*, vol. 39, No 9, 1991, pp. 2101-2104.

[10] G.M Georgiou and C. Koutsougeras "Complex Domain Backpropagation", *IEEE Transactions on Circuits and Systems CAS- II. Analog and Digital Signal Processing* , vol. 39, No 5, 1992, pp. 330-334.

[11] T. Nitta, "An Extension of the Back-Propagation Algorithm to Complex Numbers", *Neural Networks*, vol. 10, No 8, 1997, pp. 1391-1415.

[12] S. Fiori, "Learning by Criterion Optimization on a Unitary Unimodular Matrix Group", *Journal of Neural Systems*, vol. 18, No 2, pp. 87-103.

[13] M.d.F. Amin, M.d.M. Islam, K. Murase, "Ensemble of Single-Layered Complex-Valued Neural Networks for Classification Tasks", Neurocomputing, vol. 7 No 10-12, 2009, pp. 2227-2234.

[14] R. Savitha, S. Suresh, N. Sundararajan, and H. J. Kim, “A Fully Complex-valued Radial Basis Function Classifier for Real-valued Classification”, *Neurocomputing*, vol. 78, no. 1, 2012, pp. 104-110.

[15] R. Savitha, S. Suresh and N. Sundararajan, "Projection-Based Fast Learning Fully Complex-Valued Relaxation Neural Network", *IEEE Transactions on Neural Networks*, vol. 24, No 4, April 2013, pp. 529-541.

[16] I. Aizenberg, C. Moraga, and D. Paliy, "A Feedforward Neural Network based on Multi-Valued Neurons", In *Computational Intelligence, Theory and Applications. Advances in Soft Computing* , XIV, (B. Reusch - Ed.), Springer, Berlin, Heidelberg, New York, 2005, pp. 599-612.

[17] I. Aizenberg, and C. Moraga, “Multilayer Feedforward Neural Network based on Multi-Valued Neurons (MLMVN) and a backpropagation learning algorithm,” *Soft Computing*, 11, issue 2, , Jan. 2007, pp. 169-183.

[18] N. N. Aizenberg, Yu. L. Ivaskiv, D. A. Pospelov, and G.F. Hudiakov, "Multivalued Threshold Functions. Synthesis of Multivalued Threshold Elements", Cybernetics and Systems Analysis, vol. 9, No 1, January 1973, pp. 61-77.

[19] N.N. Aizenberg and I.N. Aizenberg, "CNN Based on Multi-Valued Neuron as a Model of Associative Memory for Gray-Scale Images", *Proceedings of the Second IEEE International Workshop on Cellular Neural Networks and their Applications,* Munich, October 14-16, 1992, pp.36-41.

[20] F. Rosenblatt On the Convergence of Reinforcement Procedures in Simple Perceptron. *Report VG 1196-G-4. Cornell Aeronautical Laboratory*, Buffalo, NY, 1960.

[21] I. Aizenberg, D. Paliy, J. Zurada, and J. Astola, "Blur Identification by Multilayer Neural Network based on Multi-Valued Neurons", *IEEE Transactions on Neural Networks*, vol. 19, No 5, May 2008, pp. 883-898.

[22] O. Fink, E. Zio, U. Weidmann, “Predicting Component Reliability and Level of Degradation with Complex-Valued Neural Networks”, *Reliability Engineering & System Safety*, vol. 121, 2014, pp. 198–206.

[23] I. Aizenberg, A. Luchetta and S. Manetti, S, “A modified learning algorithm for the multilayer neural network with multi-valued neurons based on the complex QR decomposition,” *Soft Computing*, vol. 16, issue 4, 563-575, Apr. 2012.

[24] I. Aizenberg, “MLMVN with Soft Margins Learning”, *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, No 9, September 2014, pp. 1632-1644.

[25] “The NVIDIA® CUDA® Toolkit”, NVIDIA Corporation, available online at https://developer.nvidia.com/cuda-toolkit

[26] J. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *Comput Sci Eng*, vol. 12, no. 3, pp. 66–73, 2010.

[27] X. Sierra-Canto, F. Madera-Ramirez; V. Uc-Cetina, “Parallel Training of a Back-Propagation Neural Network using CUDA”, *Proc. of 2010 9th IEEE International Conference on Machine Learning and Applications* (ICMLA-2010), 2010, pp. 307 – 312.

[28] C. Oei, G. Friedland, and A. Janin, Parallel training of a Multi-Layer Perceptron using a GPU”, International Computer Science Institute at Berkeley, Technical report, available online at http://www.icsi.berkeley.edu/pubs/techreports/TR-09-008.pdf

[29] J. Wilson and I. Aizenberg, “Intelligent Edge Detection using a CUDA Simulator of Multilayer Neural Network Based on Multi-Valued Neurons”, Proceedings of the 2012 International Conference on Image Processing, Computer Vision & Pattern Recognition (IPCV-WORLDCOMP’12) (H.A. Arabnia and L. Deligiannidis – Eds.), CSREA Press, Las Vegas, July 2012, pp. 291-297.

[30] N. N. Aizenberg, Yu. L. Ivaskiv, and D. A. Pospelov, "About one generalization of the threshold function" *Doklady Akademii Nauk SSSR* (*The Reports of the Academy of Sciences of the USSR*), vol. 196, No 6, 1971, pp. 1287-1290 (in Russian).

[31] S. van der Walt, S. C. Colbert, and G. Varoquaux, “The NumPy Array: A Structure for Efficient Numerical Computation,” *IEEE Computing in Science & Engineering*, 2011.

[32] “PyCUDA Documentation. Pythonic access to Nvidia‘s CUDA parallel computation API”, available online at http://documen.tician.de/pycuda/

[33] “CUDA Toolkit 4.0 CUBLAS Library”, NVIDIA Corporation, available online as part of the GPU Computing SDK, http://developer.nvidia.com/object/gpucomputing.html

[34] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, “PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation,” *Parallel Comput*, vol. 38, no. 3, pp. 157–174, 2012.

[35] Lev Givon et al.. (2015). scikit-cuda 0.5.1. Zenodo. 10.5281/zenodo.40565

[36] I. Aizenberg, " Adjustments to the proofs of the convergence theorems", available online at http://www.eagle.tamut.edu/faculty/igor/CVNN-MVN_book_Convergence_Proofs_Adjustments.htm (2013).

[37] A. Asuncion and D. J. Newman, “UCI Machine Learning Repository”. Available: http://www.ics.uci.edu/~mlearn/MLRepository.html . Irvine, CA: University of California, School of Information and Computer Science, 2007.

[38] H. C. Burger, C. J. Schuler, and S. Harmeling, “Image Denoising: Can plain Neural Networks compete with BM3D?”, in *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition* (CVPR-2012), 2012, pp. 4321-4328.

[39] H. C. Burger, C. J. Schuler, and S. Harmeling, “Image Denoising with Multi-Layer Perceptrons, Part 1: Comparison with Existing Algorithms and with Bounds”, available online at http://arxiv.org/pdf/1211.1544.pdf, 2012.

[40] H. C. Burger, C. J. Schuler, and S. Harmeling, “Image Denoising with Multi-Layer Perceptrons, Part 2: Training Trade-offs and Analysis of their Mechanisms, available online at http://arxiv.org/pdf/1211.1552.pdf, 2012.

[41] Test Images. University of Granada Image Data Base. Available Online http://decsai.ugr.es/cvg/dbimagenes/

[42] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian, “Image Denoising by sparse 3-D Transform-Domain Collaborative Filtering”. *IEEE Transactions on Image Processing*, vol. 16, no 8, pp. 2080–2095, 2007.

Fig. 3
The "Train Station" image, which was not used in the learning sets (the original image)

Fig. 4
The "Train Station" image from Figure 10 corrupted by additive Gaussian noise with the standard deviation $\sigma_{noise} = 0.2\sigma$. Testing PSNR=25.561

Fig. 5
The image from Fig. 4 filtered with MLMVN [225, 4096x4, 225] trained using the 7000 patches taken from 100 images. Learning RMSE=5.993 Testing PSNR=25.224

Fig. 6
The image from Fig. 4 filtered with MLMVN [225, 4096, 225] trained using the 7000 patches taken from 100 images. Learning RMSE=3.836 Testing PSNR=27.807

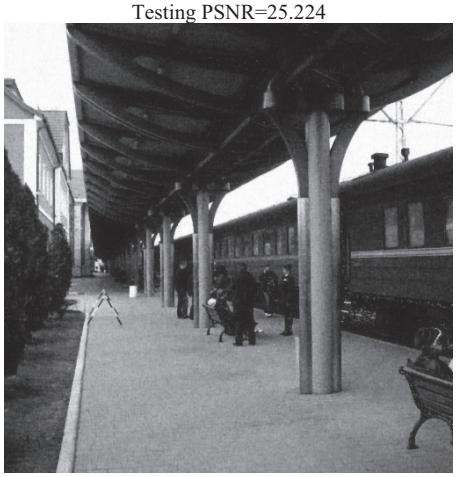

Fig. 7
The image from Fig. 4 filtered with MLMVN [225, 512, 4096, 225] trained using the 7000 patches taken from 100 images. Learning RMSE=8.234 Testing PSNR=29.548

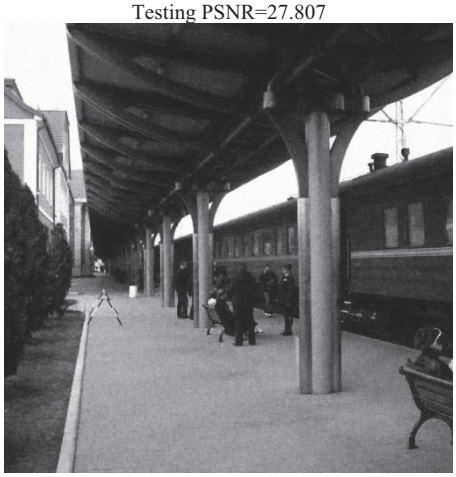

Fig. 8
The image from Fig. 4 filtered with MLMVN [225, 512, 4096, 225] trained using the 10000 patches taken from 100 images. Learning RMSE=9.574 Testing PSNR=30.86