Survey of Implementation and Visualization of Neural Networks

Nate Gauntt
University of New Mexico
Dept. of Computer Science
Email : negaunt@cs.unm.edu

***Abstract* - Modern Graphics Processing Units (GPUs) are cheap, ubiquitous, and increasing in performance at a rate two to three times that of traditional CPU growth. As GPUs evolve in function and become ever more programmable each generation, they are increasingly put to use in various high throughput parallel calculations, such as solving fluid flow equations or simulating plate tectonics. In this paper, we discuss the recent trend in implementing classical Neural Network architectures on the GPU, allowing in some cases real time input processing, and in other cases dramatic speedup of traditional algorithms. Also, we discuss a broad survey of techniques developed within the last 20 years to analyze the often non-linear dynamics of neural systems.**

1 Introduction

In the early 1940's and throughout the 50's, paritally as a result of American and British codebreaking activities, the theory of computation became an area of active research in the scientific community. In a 1943 paper published by Warren McCullogh and Walter Pitts [2], a formal system is created that mathematically describes the behavior of biological neurons and networks of neurons. Using this system, they show that, with sufficient neurons and proper connections, any computable function can be represented by such a network of neurons. In 1969, Minsky and Papert [3] explain the serious mathematical limitations of a single artificial neuron, including it's inability to learn the trivial XOR boolean function. This symbolist critique cast doubt on the efficacy of using neural networks (NNs) for learning tasks for more than a decade until a revival of the technique in the 1980's [4], when an efficient method of error feedback learning called back-propogation was discovered.

Just as connectionist methodology flourished in the 80's and early 90's after efficient error feedback was discovered, modern graphics hardware provides another efficiency revolution, allowing neural networks to be applied easily and cheaply to certain real-time problems and other throughput-sensitive domains. It is only within the last several years that Graphics Processing Units (hereafter GPUs) have had the ability to store results efficiently to 32-bit textures (GeForce 8, 2006), have been able to be programmed by high level languages such as Cg (2002), or have had the tool support to be used together in robust high-performance architectures like CUDA (2006) [5, 7, 8]. There has been much research in the last several years describing advancements in implementation of neural networks using such parallel hardware, which we will address presently.

2 GPU Economic Advantages

Previously, we discussed the possibility of networking several GPUs connected together in the same computer system to perform calculations. In [1], Meuth describes how this technique has been used in industry to make high performance 'cluster of clusters', often taking the commodity aspect further by using clusters of game consoles as cost-effective supercomputers. Little success was achieved by networks of the first-generation XBox [9], though the University of Illinois - Urbana Champaign developed a productive 64 node Playstation 2 cluster for $15,000. Using data from [10], this translates into $39/GFLOP, as opposed to the estimated price of using CPUs (Pentium4-3200), $312/GFLOP. Thus, in 2005, there is already an order of magnitude price difference per GFLOP in using GPUs vs CPUs for at least some cluster applications. Meuth summarizes the FLOPS per dollar ratio of modern CPUs vs various console generations of GPUs in figure 1.

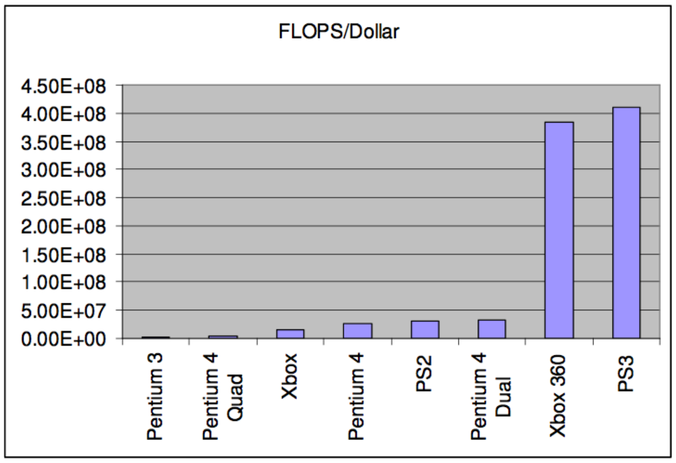


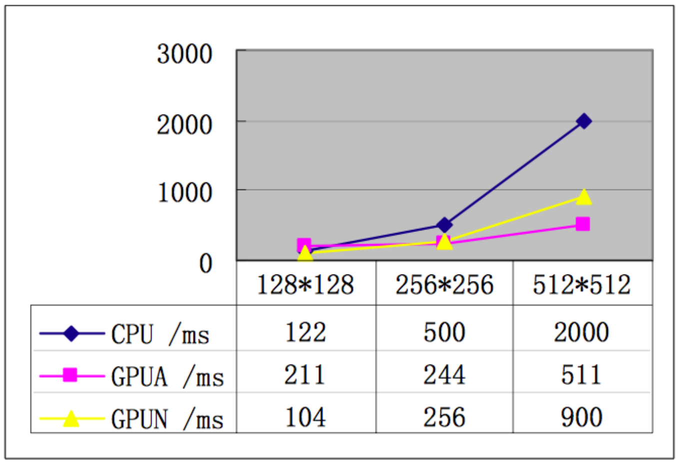
Figure 1: Shows the FLOPS per dollar ratio of the past two generations of game consoles and Intel Processor-based systems, from Meuth [1]

3 Neural Network Implementations

Neural Network implementations on the GPU have evolved with the changing state of the underlying hardware. One of the earlier implementations is by Bohn [13], who implements Kohonen feature mapping on an SGI workstation. His efficient implementation in OpenGL is possible through accelerated GL functions glColorMatrix, and glminmax, and various texture blending functions. This behavior was possible utilizing expensive SGI workstations; however, the modern trend was to sacrifice accelerated 2D functions for better 3D performance. Given the minor programmability of hardware at the time, Bohn relies heavily on OpenGL blending functions, tying such an implementation to vendor support of an accelerated high-level API.

In the early 2000's, much effort was spent investigating increasingly programmable GPUs, and their application to scientific computing [14, 15]. Fast linear algebra on the GPU requires fast matrix multiply implementations, and this work led to several neural network GPU implementations utilizing matrix multipy routines. Rolfes [12] does this, using a blend of DirectX and hand-tuned GPU assembly code. Unlike Bohn, Rolfes relies less on accelerated DirectX calls, but instead relies on fast implementation of certain GPU instructions and hand-tuned assembly code. Rolfes' implementation is somewhat more portable and less fragile than Bohn's, but still relies on implementation details, which are constantly in flux, in order to achieve good performance. The main contribution of this implementation techinique is to accelerate multi-layer perceptron (MLP) neural networks on commodity graphics hardware. For another matrix-based computation of MLP, see Kyoung-Su's [**?**] implementation, used for real-time text detection in a video stream. Kyoung-Su achieves a 20-fold performance increase over CPU implementation.

As the language of high level shader programming became standard and GPU compiler optimization improved, implementors began moving to higher level implementations of neural networks. Zhongwen et. al. [11] used Cg to implement both self organizing map (SOM) and MLP neural networks. Despite the high level language, some knowledge of GPU instructions is needed to construct efficient algorithms. For example, to find the minimum value (or best match) in the SOM, Zhongwen uses a recursive algorithm dividing the activation space into 4 at each step, until only scalar 2x2 matrices are left. This may seem puzzling without the knowledge that most GPUs have an efficient **min** function that computes the minimum of 2 4-vectors. Zhongwen achieves a 2 to 4 fold performance advantage in SOM over the CPU, and 200 fold performance advantage over CPU implementation. It is not known what CPU implementation is used, or if it is even optimized, so it is important to carefully consider performance claims. That said, Zhongwen's MLP was able to compute an object tracking function in 46 ms, which they note is theoretically fast enough to be useful for real-time response. Zhongwen also notes several caveats of high level implementations. In one case the same Cg code yielded correct results on the ATI card and incorrect results on the NVidia card, so a code change was needed to work around a probable hardware defect. The performance of GPU code is very sensitive to the number of rendering passes and data transfer to the CPU, which they minimized where possible. Also they note that 32-bit floating point textures were critical to the accuracy of the MLP, a feature that is not fully supported on many graphics cards, and may also suffer from SGI patent restrictions [6].



	128*128	256*256	512*512
CPU /ms	122	500	2000
GPUA /ms	211	244	511
GPUN /ms	104	256	900

Figure 2: SOM training time, CPU vs ATI / NVidia GPU, from Zhongwen [11]

While Cg provides a language low level enough to be portable across graphics APIs (such as DirectX and OpenGL), but high enough to avoid hardware-dependant and cumbersome GPU assembly code, it is not the only language used for writing flexible shader programs. Recently, Davis explores the use of BrookGPU [17], a stream programming framework designed for use on the GPU. BrookGPU is slightly higher level than Cg, abstracting the details of shader and vertex programs running on the GPU, instead focusing on input streams being operated on by special functions called kernels. The compiler enforces limitations inside kernel functions to ensure only efficient operations are executed on the GPU. The main benefit, as Davis explains in [4], is portability, as a single BrookGPU program can be targeted to DirectX or OpenGL. Similarly, BrookGPU allows the programmer to focus on algorithm design, rather than on handling graphics API calls or decyphering shader language syntax. Davis was able to quickly implement Hopfield, MLP, and Fuzzy ART neural networks, and compare the BrookGPU to CPU implementations. For MLP and Hopfield networks, ATLAS BLAS optimized linear algebra suite was used in the CPU implementation, and complete code for both BrookGPU and CPU implementations is given. Davis was able to realize a modest 2 fold performance increase over CPU for MLP and Hopfield networks; however, the Fuzzy ART implementation was dramatically slower than the CPU. The reason given for this is that BrookGPU's stream model doesn't support certain useful texture memory updates, and presumably other low level optimizations, that Cg allows. Similar to Zhongwen, Davis finds significant performance differences between NVidia and ATI performance for the same code (see figure 3), suggesting that high level implementations are sensitive to hardware differences. Another explantion is that the NVidia and ATI shader program compilers have signficant differences in optimizing code. In either case, programmer caution is advised.

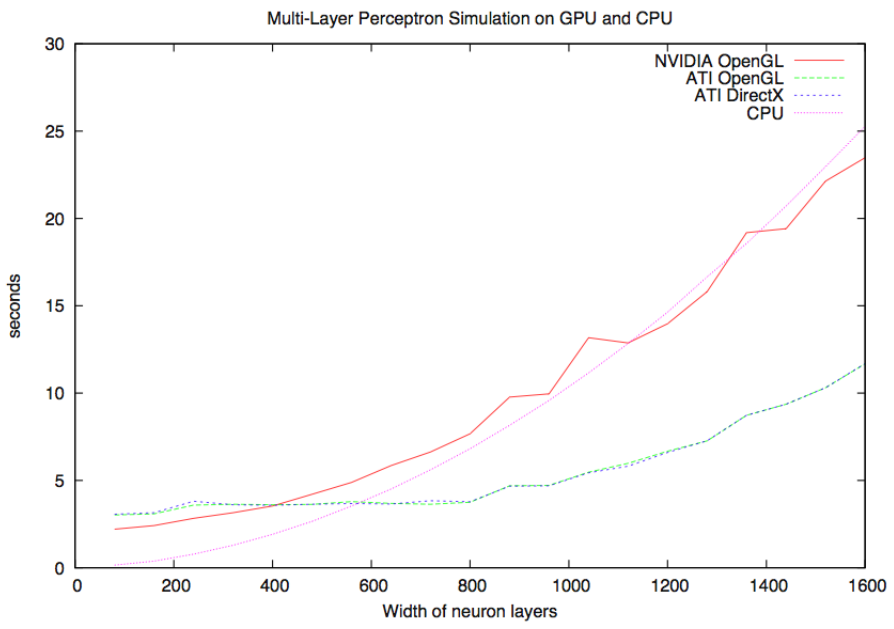


Figure 3: Calculation time in MLP vs. network layers, from [4]. Red NVidia and green ATI BrookGPU code are better than CPU as layers increase

Spiking neural networks are often more interesting to neurobiologists than the MLP neural network, as they preserve more information about the dynamics of neurons and more closely resemble biological neurons. With this fidelity comes cost, and where MLP neurons are updated on each processor timestep, spiking networks may take up to 100 timesteps to update a neuron. To this end, several researchers have developed GPU algorithms for simulating these networks more efficiently [18, 19]. In [18], Bernhard uses spiking networks to solve an image segmentation problem. Neurons are connected locally for exitation for regions of similar color, and are connected globally for inhibition regardless of color. Reading locally from a few number of exitatory neurons in parallel is efficient for fast GPU texture memory, but reading all other neurons for global inhibition is not. Bernhard solves this by adding an additional pass where a pixel is written to only if a neuron has fired. Then, using an *occlusion query*, he counts the number of pixels written, and this is the same as the global inhibition for the neuron. Since GPUs accelerate occlusion queries, this results in a 5 to 7 times speed increase versus CPU timings. An unrelated benefit of simulating neural networks on the GPU is ease of visulization. Figure 4 is almost literally successive memory dumps of the GPU algorithm as it executes.

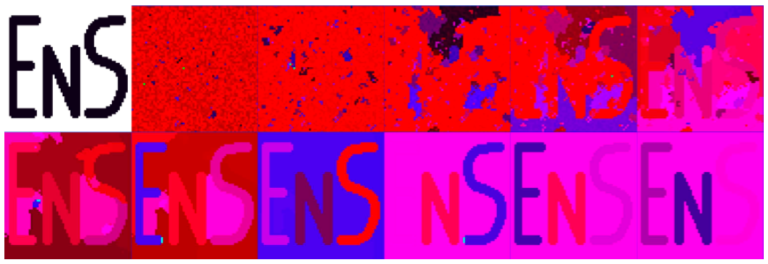


Figure 4: Time series evolution of spiking neurons for image segmentation, from [18]. Areas of similar color are similar activation level

Nageswaran's approach also models spiking neural networks, though at a significantly greater level of biological plausibility. In order to do this, he uses CUDA, similar in some ways to BrookGPU in that it abstracts the GPU as a multiprocessor, capable of running lightweight threads simultaneously and accessing shared memory through texture memory. Threads in CUDA can be scheduled in groups by the on-GPU hardware scheduler, such that if a thread in the group stalls reading GPU texture memory, another non-stalled group can run. Similar to parallel distributed systems literature, complex CUDA algorithms have the same kind of problems with synchronization and optimization. Despite this, Nageswaran et. al. were able to simulate 100K neurons with 50M synaptic connections with a 25 fold speedup over a CPU implementation, overall being only 1.5 times slower than real-time (ie. one update per timestep). It is interesting to note that in both spiking network implementations that learning takes place on the GPU, which is rare for GPU neural networks. Part of why it is efficient to do "learning updates" for spiking networks is that instead of weights being represented by contents of memory addresses, they are represented by firing frequency, implicitely calculated by the simulation itself.

Fuzzy ART is perhaps a middle ground between the minimal analytic model of MLP networks and the biologically correct spiking networks, and it's implementation on the GPU derives in part from various clustering algorithms on the GPU, such as SOM in [11, 13]. Martinez-Zarzuela et. al. implement Fuzzy ART on the GPU, in two parts, in [20]. The first part is a GPU ART network optimized for learning, and another GPU ART network optimized for testing. The learning ART parallelizes at calculating input activations and sorting for the best category match. The sorting for best category match closely resembles the best match reduction Zhongwen uses for SOM, and the updating of ART weights is accomplished by rendering into a sub-region of a texture via scissoring. Where training calculates neuron response and best match in parallel for a single input vector, testing calculates category for several input vectors simultaneously. Testing calculates the first category for n input vectors, and the first category response for each input vector is stored in parallel to n textures. These n textures are fed back into the network, and if the second category response is greater, it will overwrite the category number. This process continues until all categories have a chance to respond and compete. Unfortunately, training updates tended to be the performance bottleneck, with the GPU training network running 10 to 100 fold slower than a CPU implementation, whereas the GPU testing network had no such bottleneck and ran 25 to 46 fold faster than the CPU. The authors note that un-optimized memory transfer from CPU to GPU is a potential source of improvement for the testing network. See [21, 22] for GPU treatment of related non-neural clustering algorithms.

4 Neural Network Visualization

Since the connectionist counter-revolution of the 1980's, researchers have been using neural networks productively for many and varied tasks, such as approximating atmospheric lighting in [23]. Unfortunately, the non-linear nature of even small, feed-forward networks makes them difficult to formally analyze, and may have been the source of the original (erroneous) critique of general MLP networks by Papert and Minsky. If the result of the neural network is primarily visual, or if the algorithm performs image or video processing, the visual debugging or presentation of the neural network can be straightforward, such as in figures 4, 5, or in [23]. If the result of the network is an arbitrary classification task; however, the visual interpretation is far from obvious.

An earlier work in visual analysis of neural networks is the Hinton diagram in [24]. In figure 6, each black square is a network of neurons responding to the input, which can be thought of as a query. The response is shown by the size of the squares, which when viewed en masse, show which part of a large network is responding to the input. This is also an early example of *semantic zooming*, where on one plate, Hinton represents a network with squares corresponding to neuron activation, and on the next plate, he represents a network of networks, with squares corresponding to average network activation.

Another earlier work by Dennis [25] uses principal component analysis (PCA) and canonical discriminant analysis (CDA) to plot the hidden layer activations of a feed-forward network as it learns a classification task. PCA plots the data points along axes which maximize variance and preserve, to some extent, distance between points in the original space and the transformed space. CDA requires points to be labelled by cluster group, and plots the data points along axes which cluster points within groups while maximizing distance between

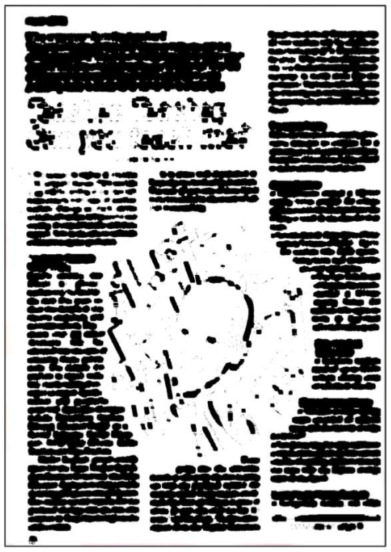


Figure 5: Visual debugging: black pixels are ones categorized as containing text, from [16]

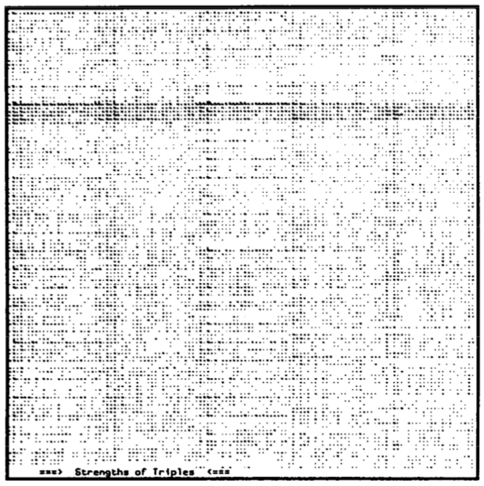


Figure 6: Hinton Diagram: squares are memory networks, size is average network response to input. Darker rows represent a stored pattern in memory.

groups. Unlike PCA, CDA cannot work on unlabelled data, and CDA doesn't necessarily preserve distance between points in the original space and the transformed space. CDA, in figure 7 gives a more meaningful picture of the hidden unit state evolution, but requires a priori knowledge about the task, which is probably unsuitable for networks using unsupervised learning. The historical discussion of automatic techniques like PCA versus guided techniques like CDA is important, as there is a tension in the literature as to (1) what level of a priori knowledge it is appropriate to assume and get unbiased results, and (2) how to differentiate between meaningful clustering and accidental clustering in complicated statistics involving high dimensional data.

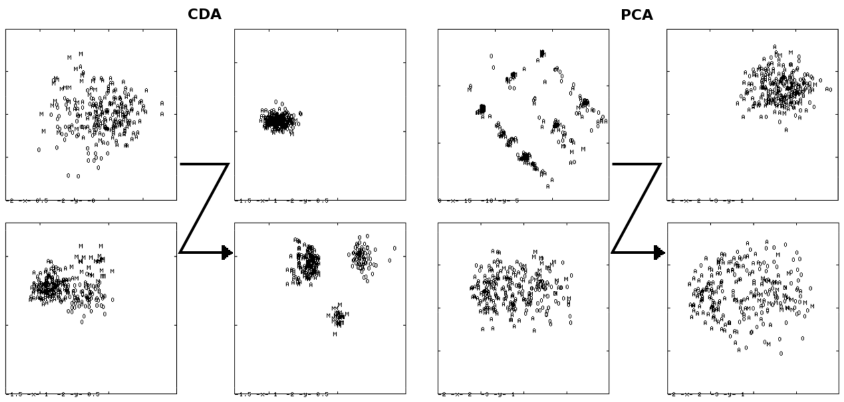


Figure 7: Left: CDA plot of hidden layer   Right: PCA plot of hidden layer
Visualization of hidden layer weights in MLP as network learns, from [25]

MLPs are by far the most popular network visualized in the literature, Craven collects and summarizes several techniques in [26]. These include a version of Hinton diagrams modified to show polarity and typically spatially organized to show directional connections, see figure 8.

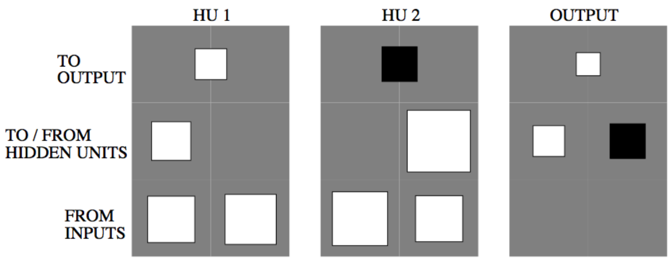


Figure 8: Modified Hinton diagram: size and color are magnitude and polarity of activation, spatial position is connection geometry, from [26]

Wejchert and Tesauro develop another way of visualizing MLP networks in [27]. In figure 9, a bond diagram is displayed, its namesake deriving from weights being displayed as variable length and colored "bonds", where the visible fraction of the length corresponds to the weight magnitude and the color correponds to the weight polarity. Also, the size of the neurons (white circles) that originate the bonds corresponds to the magnitude of the *bias* weight. Craven notes that the advantage over Hinton diagrams is that bond diagrams show network topology in a more obvious way. The disadvantage of this technique, notes Craven, is that it is difficult to compare the magnitude of the *bias* with the magnitude of the *bonds*, as one is represented primarily by area and the other by relative length.

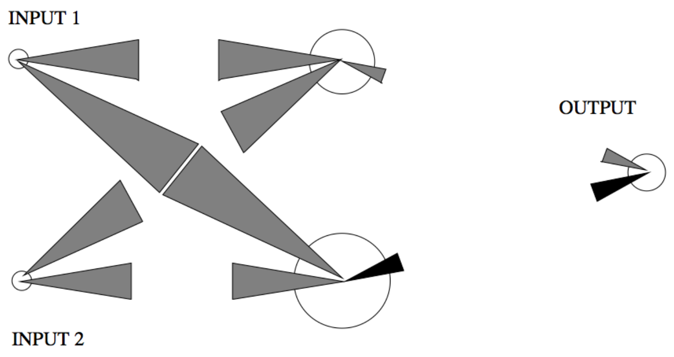


Figure 9: Bond diagram: The white circles represent neurons, circle size is bias magnitude, bond length is connection weight, bond color is weight polarity, from [26]

They develop another visualization technique in the same paper, called *trajectory diagrams* reminding one of Charles Joseph Minard's diagram of Napoleon's eastern Europe campaign in [30]. Figure 10 shows such a diagram, where the axes are ranges over two of the input weights to this neuron, and the thickness of the line is proportional to error.

Pratt and Nicodemus develop a tool to display moving hyperplanes in the frame of the input space as the neurons learn, see figure 11 for details. Though they discuss mostly feature spaces of length 2, this method could be applied to arbitrary length input vectors by using PCA or a related statistical technique to find a meaningful 2D representation. Like CDA, however, animated hyperplanes remain meaningful only when working with labelled classifier data, and so obey the same limitations of CDA or other high dimensional clustering techniques.

Lang developed a similar technique in the late 80's to display hyperplane response to input stimulus in [29]. The x and y axis represent the full range of two components of an input vector, and the gray level magnitude is the activation, with white and black representing polarity, see figure 12.

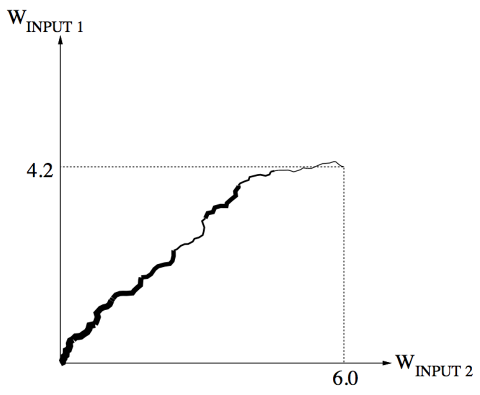


Figure 10: Trajectory diagram: x and y are two input weights, line thickness is error, from [26]

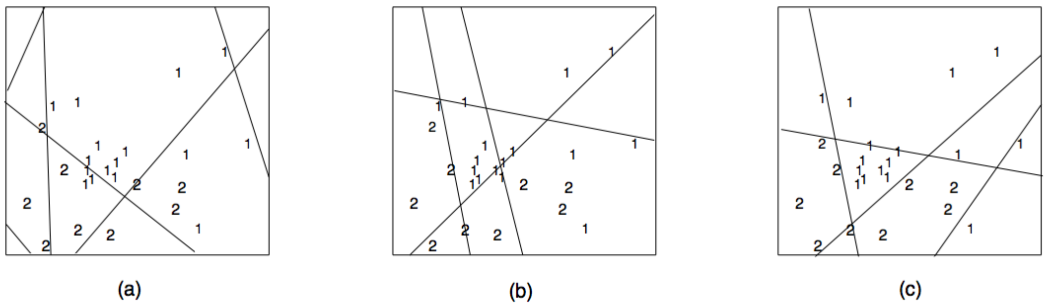


Figure 11: Animated Hyperplanes: Left to right: (a) no training, (b) training with 1/2 epochs, (c) training with full number of epochs, from [28]

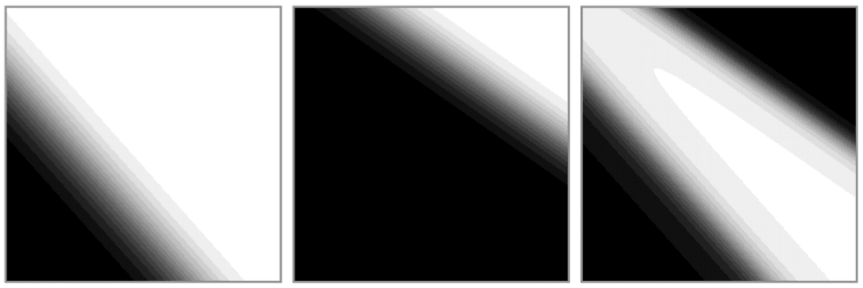


Figure 12: Response function plots: x and y are input ranges, gray level is activation. Left to Right: three hidden unit responses to input, from [26]

More recently, Gallagher [31] provides a comprehensive review of neural network visualization, focusing on using PCA to analyze neural networks in his own work. Figure 13 is an elegant way of looking at the error surface in terms of principal components, projected into three dimensions. His work only analyzes MLP networks trained with back-propogation. Indeed such as diagram as in figure 13 would not be smooth or even meaningful without the smooth weight change of back-propogation, or a similar learning algorithm that smoothly changes weight vector.

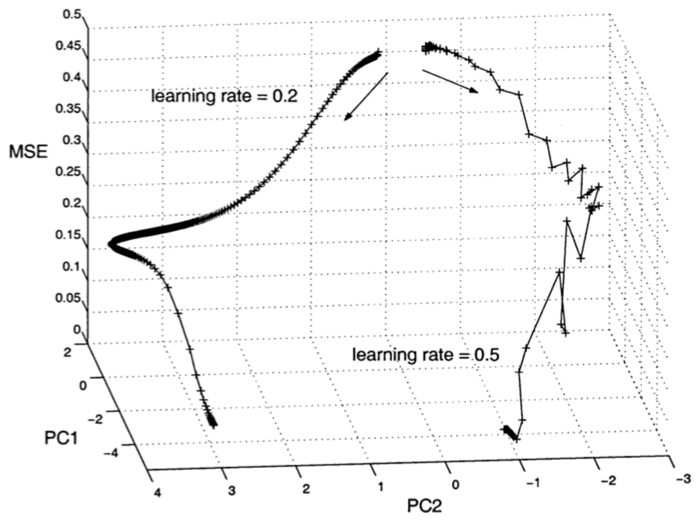


Figure 13: Plot of error surface relative to 2 principal components of input space, from [31]

There are many other kinds of neural networks than MLP networks, and novel visualization methods have been developed to address these. In [32], Duch examines hidden node visualization using primarily radial basis function neurons in the hidden layer. If there are 2 classes in a classification task, then for an arbitrary (normalized) input vector $i = [0.38, 0.64]$, the network will map i onto one of the classes, say $k_2 = [0, 1]$. The hidden layer, then, maps input vectors into clusters in the non-discrete classification space, say onto two clusters, k'_1 centered around $[0.8, 0.1]$, and k'_2 centered around $[0.2, 0.9]$. These clusters must be linearly seperable to be accurately processed by the output neurons, and the degree to which they actually are seperable can show the generalization of a network. This can be extended to arbitrary dimensions, practically limited to 5 dimensions for graph structure clarity. In figure 14, the 4 and 5 dimension lattices that the hidden nodes are mapped onto are shown. For the 2D case, the lattice that k'_1 and k'_2 would be mapped onto would be $L_2 = \{(0, 0), (1, 0), (0, 1), (1, 1)\}$, for example.

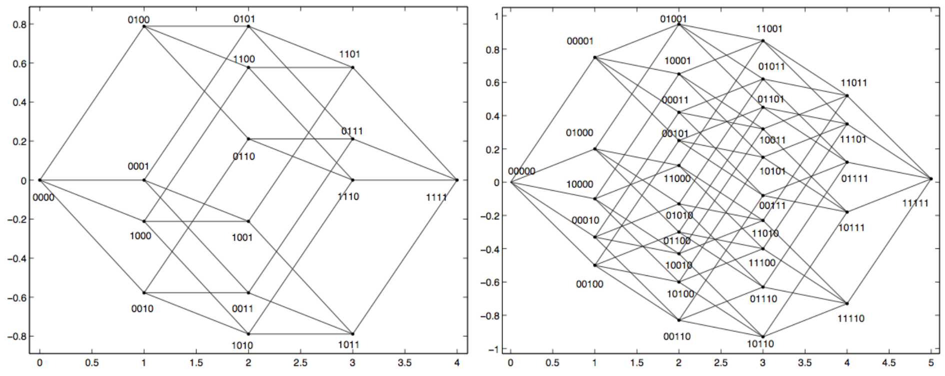


Figure 14: Left to right: 4 and 5 dimensional lattices for input space, from [32]

Mapping the hidden layer onto the lattice occurs automatically. First the input space is rotated such that the vector $K = [1, 1, \ldots, 1]$ points at the reader, out of the page. For the output space, this would not hide any correctly classified datums, as $k_{zero} = [0, 0, \ldots, 0]$ and $k_{one} = [1, 1, \ldots, 1]$ would be mapped to the same 2D point, but no output point should be labelled with no class or with all classes. For hidden spaces, however, this can obscure intermediate points along that line. Second, the rotated input space is rotated again around the PCA vector of greatest variance, to spread out the hidden images of the input space. This is shown in figure 15, for trained RBF networks that either solve well or solve poorly the noisy XOR problem of seperating 4 non-overlapping Gaussians.

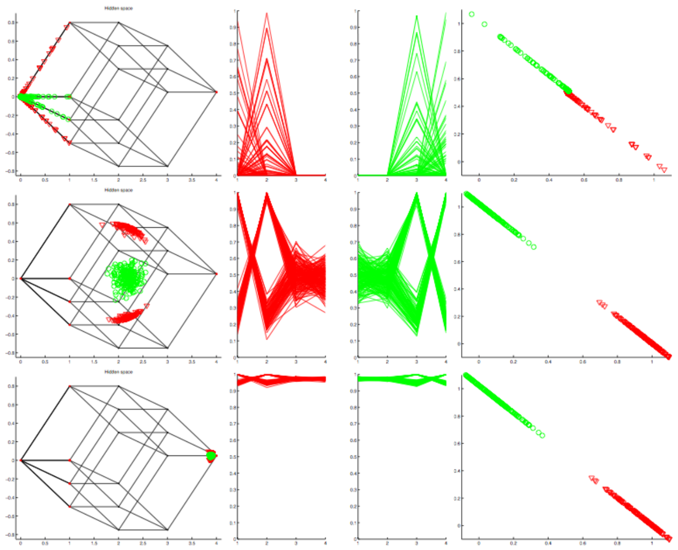


Figure 15: (top) Gaussian neuron falloff too slow, $\sigma = 1/7$ (mid) optimal falloff, set to $\sigma = 1$, (bot) conservative falloff, set to $\sigma = 7$. (left) hidden layer input image mapped onto hypercube, (mid) parallel coordinate representation, (right) output space, from [32]

Another type of non-MLP network useful to researchers has been the ART network and its derivatives. In [33], Bartfai et. al. introduce two useful visual displays, *circle diagrams* and *input relative diagrams*, for describing templates and input presentation response, respectively. Each projection of a k dimensional feature vector is represented by short line segments on the 2D plane, with a unique angle per segment inside of the projection. For example, a 2D feature vector $[x, y]$ where $0 < x, y < 1$ could be represented uniquely by a pairs of short line segments whose angles from a reference line would be $[x * 360, y * 360]$. Further, all the segments in x would have unique angles, and all the segments in y would have unique angles, but x would not necessarily be unique to y with respect to the angle. Also, if the feature space of x ranges smoothly from 0 to 1, the line segments of x placed co-terminally form a circle, hence circle diagrams, see figure 16.

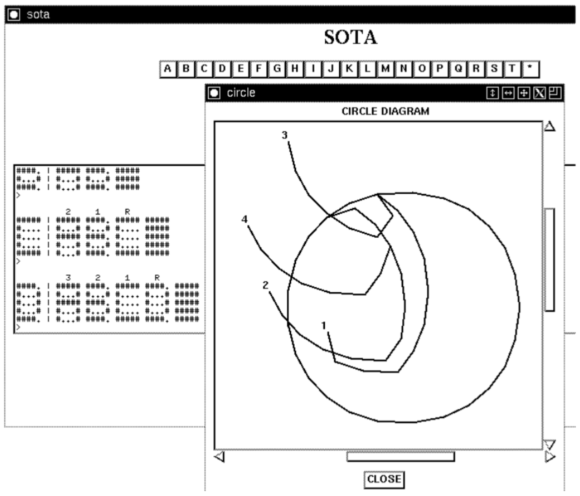


Figure 16: Circle diagram: numbered arcs are categories, arc length is number of input features matched by category, circular arc is uncommitted category nodes. Vigilance is 1, after presenting A..D there are 4 categories, from [33]

Where circle diagrams show the state of the ART network, input relative diagrams show the process by which new templates are committed and the effect of vigilance on selection, see figure 17.

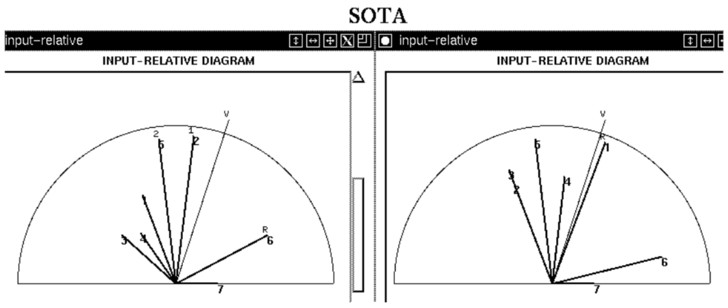


Figure 17: Input relative diagram: bold rays (numbered) are categories, ray length is activation, semicircle is maximum activation, ray angle is (template ∩ input)/|input|, and 'v' ray is vigilance, from [33]

5 Neural Network Teaching Tools

Here we present a few more works that could fall under visualization; however, some representation choices were made specifically to serve as user interface elements, and so fit better in their own category.

N^2VIS, in [34], is a testbed where neural network students can design, run, and modify MLP networks as they run. In figure 18, a MLP is displayed as the simulation runs. Black circles are neurons, and they are filled with white as a sigmoid function of activation. Weights are edge colors, where the colored portion, similar to bond diagrams, is the weight magnitude and the color is the weight polarity. The small lines intersecting with the edges are both user interface elements to change the weights, and also the colored fraction represents weight variance calculated over several timesteps of the simulation. In the lower right corner is a *compact matrix* representation of the same MLP, described later.

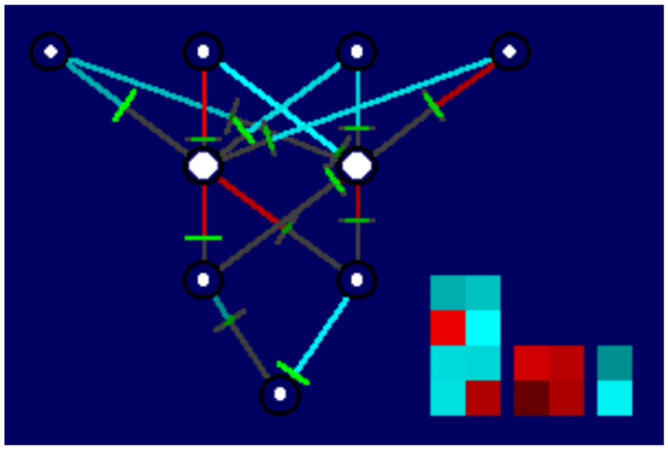


Figure 18: MLP display with slider bars to adjust weights, from [34]

N^2VIS can also run large numbers of competing MLPs that modify themselves with a basic genetic algorithm over some fitness criteria. In order to display many networks at once on the screen, the *compact matrix* representation was created. In figure 19, a genetic inheritance hierarchy of MLPs is displayed in compact format. The compact matrix representation is initially similar to a colorful Hinton diagram, though weights are represented instead of activations. It consists of a number of colored tilings grouped into a container square, where the number of tilings is equal to the number of layers, the height of a tiling is the layer width in neurons, and the width of the tiling is the number of forward connections per neuron in a layer. Like Hinton diagrams, it is useful to display a gestalt property over a large number of networks.

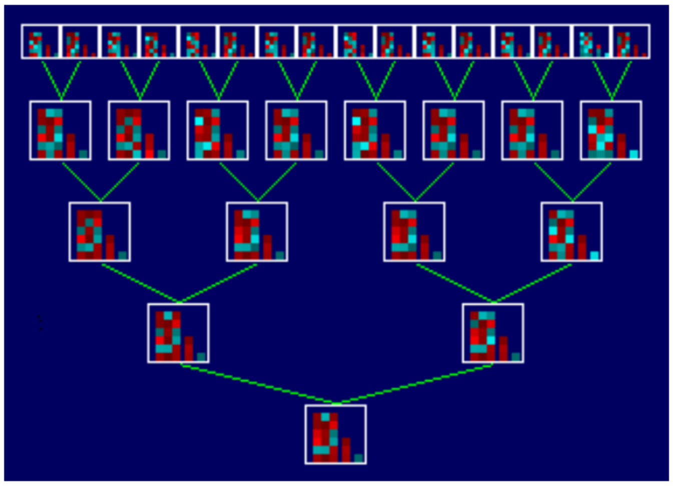


Figure 19: Genetic inheritance hierarchy of MLP population, from [34]

Simbrain [35] is primarily designed to be a pedagogical tool. It has a user interface to view, design, and connect neural networks of various types, and a Java API for writing more complex weight update and activation functions than is possible with the user interface. Also, views exist to create simple worlds to embody the designed neural networks and generate input vectors as the networks move around the space. Finally, there is a view that gives students access to various high-dimensional projection libraries, to view the progress through weight space that their embodied intelligences make. A screenshot of these UI windows can be seen in figure 20.

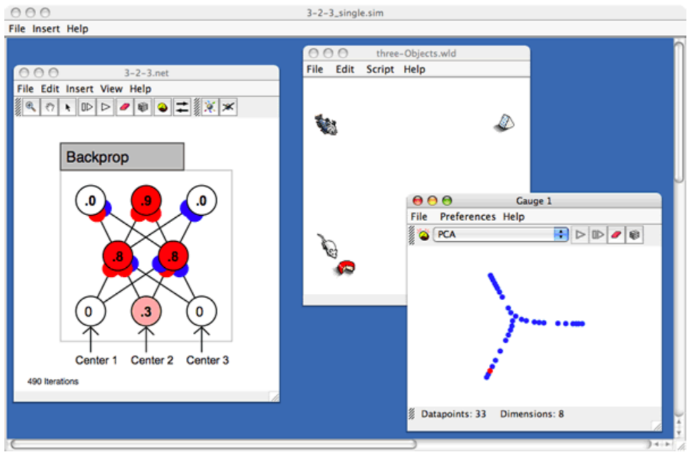


Figure 20: Left to right: network designer, world designer, state space viewer

The default visualization of networks is straightforward. Neurons are colored circles, where the color is polarity and the number inscribed is the rounded activation value. Connections are lines, and weights are represented as variable-sized colored circles positioned below (and partly occluded by) the neuron they belong to. The size of the smaller circles is the magnitude of the weights, and the color is again the polarity. All of these properties are shown in figure 21.

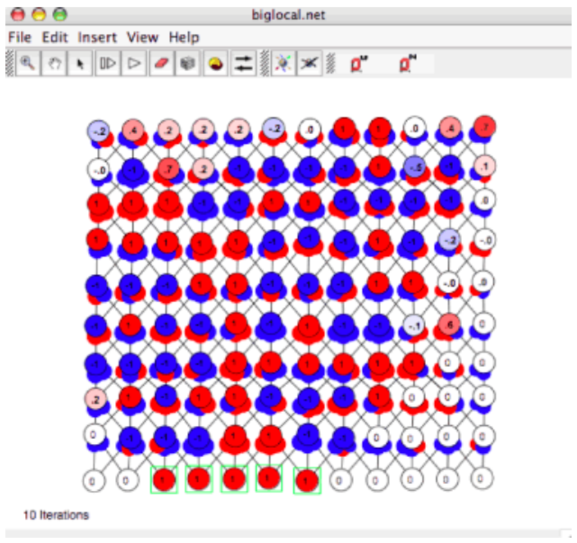


Figure 21: Network designer and simulator: medium size feed-forward MLP is shown, from [35]

The user interface has a photoshop like quality, where mouse select, cut, and paste can be used, much as in image editing, to construct network topologies. A wide variety of networks can be created with this tool, including MLP, Hopfield, and Izhikevich spiking neural networks. Also, there is API support for creating networks of continuous-valued neurons that integrate differential equations for their operation.

6 Future Work

Neural networks have benefited much from intense study over the last 20 years. Within the last 5 to 10 years, neural networks of various types have been ported to run on cheap, ubiquitous hardware that exists on practically every workstation available to the average college student. Though running artifical neurons on dedicated hardware is not a new idea by any stretch, never have so large a fraction of the research community had access to these machines, and unlike specially designed ANN electronics, GPUs are cheap, programmable, and increasingly high throughput. As CPU designers attempt to push the performance curve by adding parallelism instead of clock speed, it becomes more important to develop robust, fault tolerant computational paradigms that can make use of this new power. Neural networks are a fault tolerant, robust, inherantly parallel computational system that can fill this developing niche. Developing novel GPU implementations of autonomous agents using neural networks is the next step in not only enhancing the devices that occupy our daily lives, but also in answering the most important, the most deep questions concerning the nature of intelligence and the status of humans, as intelligent creatures, in our world.

References

[1] Meuth, Ryan; Wunsch, Donald "A Survey of Neural Computation on Graphics Processing Hardware", *IEEE 22nd International Symposium on Intelligent Control* pp. 524-527, 2007

[2] McCullogh,W; Pitts,W "A Logical Calculus of the Ideas Immanent in Nervous Activity" *Bulletin of Mathematical Biophysics* 1943

[3] Minsky, M; Papert, S "Perceptrons: an Introduction to Computational Geometry", *MIT Press, 2nd edition* 1969

[4] Davis, C "Graphics Processing Unit Computation of Neural Networks" *Master's Thesis, UNM Press* 2001

[5] "OpenGL NVidia depth_buffer_float Documentation"
http://www.opengl.org/registry/specs/NV/depth_buffer_float.txt,
accessed 12/09

[6] "OpenGL ARB texture_float Documentation"
http://www.opengl.org/registry/specs/ARB/texture_float.txt,
accessed 12/09

[7] "Cg Homepage"
http://developer.nvidia.com/page/cg_main.html,
accessed 12/09

[8] Halfhill, Tom "Parallel Processing with CUDA"
Microprocessor Report [Reprint] January 2008

[9] arch3angel@gmail.com "12 Node Xbox Linux Cluster"
http://www.xl-cluster.org/index.php,
accessed 12/09

[10] Levine,B; Schroeder,J; et. al. "PlayStation and IBM Cell Architecture"
http://www.mcc.uiuc.edu/meetings/2005eab/presentations/PlaystationIBMCell.ppt,
accessed 12/09

[11] Zhongwen, Luo; Hongzhi, Liu; Xincai, Wu
"Artificial Neural Network Computation on Graphics Process Units"
IEEE Intl. Joint Conference on Neural Networks 2005 Vol. 1 pp. 622-626

[12] Rolfes, T "Artificial Neural Networks on Programmable Graphics Hardware" *Game Programming Gems 4* 2004 pp. 373-378

[13] Bohn, C.A. "Kohonen Feature Mapping Through Graphics Hardware"
Proceedings of Third International Conferance on Computational Intelligence and Neurosciences 1998

[14] Larsen, E.S.; McAllister, D "Fast Matrix Multiplies Using Graphics Hardware" *Super Computing 2001 Conference* Denver, CO 2001

[15] Kruger, J; Westermann, R "Linear Algebra Operators for GPU Implementation of Numerical Algorithms" *SIGGRAPH 2003 Conference Proceedings*

[16] Kyoung-Su Oh; Keechul Jung "GPU implementation of Neural Networks" *Pattern Recognition* Vol. 37, Issue 6, June 2004

[17] "BrookGPU" *Stanford University Graphics Lab* http://graphics.stanford.edu/projects/brookgpu, accessed 12/09

[18] Bernhard, F; Keriven, R "Spiking Neurons on GPUs" *International Conferance on Computational Science*, 2006

[19] Nageswaren, J; Dutt, N; Krichmar, J "Efficient Simulation of Large-Scale Spiking Neural Networks Using CUDA Graphics Processors" accepted in *International Joint Conferance on Neural Networks*, 2009

[20] Martinez-Zarzuela, M; Pernas, F "Fuzzy ART Neural Network Parallel Computing on the GPU" *IWANN 2007, LNCS 4507* pp. 463-470

[21] Hall, J; Hart, J "GPU Acceleration of Iterative Clustering" *Manuscript Accompanying Poster at GP$\hat{2}$: The ACM Workshop on General Purpose Computing on Graphics Processors* and *SIGGRAPH 2004 Poster*, 2004

[22] Harris, C; Haines, K "Iterative Solutions Using Programmable Graphics Processing Units" *The 14th IEEE Intl. Conf. on Fuzzy Systems (FUZZ) 2005* May 22-25 pp. 12-18, 2005

[23] Pietras, K; Rudnicki, M "GPU-based Multi-Layer Perceptron as Efficient Method for Approximation Complex Light Models in Per-Vertex Lighting" *Studia Informatica* Vol. 2(6) pp 53-63, 2005

[24] Touretzky, D; Hinton, G "A Distributed Connectionist Production System" *Cognitive Science* Vol. 12, pp. 423-466, 1988

[25] Dennis, S; Phillips, S "Analysis Tools for Neural Networks" *Technical Report 207, Dept. of Computer Science, U. of Queensland, Austrailia*, 1991

[26] Craven, M; Shavlik, J *International Journal on Artificial Intelligence Tools* Vol. 1 pp 399-425, 1992

[27] Wejchert, J; Tesauro, G "Neural Network Visualization" *Advances in Neural Information Processing Systems* Vol. 2 pp 465-472

[28] Pratt, L; Nicodemus, S "Case Studies in the Use of a Hyperplane Animator for Neural Network Research" *Proceedings of the International Conference on Neural Networks, IEEE World Congress on Computational Intelligence* Vol. 1 pp 78-83, 1994

[29] Lang, K.J.; Witbrock, M.J. "Learning to Tell Two Spirals Apart" *Proceedings of the 1988 Connectionist Models Summer School* pp 52-59, 1988

[30] Marey, E.J. "La Methode Graphique", published in Paris, 1885
Reprinted in Tufte, E.R. "The Visual Display of Quantitative Information", 1983

[31] Gallagher, M; Downs, T "Visualization of Learning in Multi-layer Perceptron Networks using PCA" *IEEE Transactions on Systems, Man, and Cybernetics: Part B: Cybernetics* Vol. 33, No. 1, Feb. 2003

[32] Duch, Wlodzislaw "Visualization of Hidden Node Activity in Neural Networks" *International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, 2004

[33] Bartfai, G; Ellife, M "Implementation of a Visualization Tool for ART1 Neural Networks" *Proceedings of the Intl. Two-Stream Conference on Artificial Neural Networks and Expert Systems, Dunedin, New Zealand* 1993

[34] Streeter, M; Ward, M "N$\hat{2}$ VIS: An Interactive Visualization Tool for Neural Networks" *Technical Report*, *Dept. of Computer Science, Worcester Polytechnic Institute*, 2000

[35] Yoshimi, J "Simbrain: A Visual Framework for Neural Network Analysis and Education" *Brains, Minds, and Media: Journal of New Media in Neural and Cognitive Science and Education*, 2008