AIAA Paper No. 2008-0885, presented at the AIAA 46th Aerospace Sciences Meeting, Reno, NV, Jan., 2008, Copyright, 2008, Lyle N. Long

Biologically-Inspired Spiking Neural Networks with Hebbian Learning for Vision Processing

Lyle N. Long[1] and Ankur Gupta[2]
The Pennsylvania State University, University Park, PA 16802

This paper describes our recent efforts to develop biologically-inspired spiking neural network software (called JSpike) for vision processing. The ultimate goal is object recognition with both scale and translational invariance. This paper describes the initial software development effort, including code performance and memory requirement results. The software includes the neural network, image capture code, and graphical display programs. All the software is written in Java. The CPU time requirements for very large networks scale with the number of synapses, but even on a laptop computer billions of synapses can be simulated. While our initial application is image processing, the software is written to be very general and usable for processing other sensor data and for data fusion.

I. Introduction

THIS paper describes our initial progress in developing biologically-inspired neural network software using object-oriented programming. The focus of the paper is neural networks for vision processing, but the software and algorithms developed have implications for more general intelligent systems as well. While there has been a lot of discussion about computers reaching human levels of intelligence (e.g. [1], [2], etc.), these often over-simplify the difficulty in developing intelligent machines. Supercomputers are now close to the memory and speed of the human brain, but there are important system issues that are often neglected (e.g. wiring diagrams, software development, algorithms, learning, sensory input, and motor-control output). In addition, computers and brains are very different devices. Computers are extremely precise and fast, while brains are imprecise and the neurons are quite slow. A truly intelligent machine will not just be an isolated computer, it will be a complex system of systems embedded in the real world. The key to building intelligent machines, however, is machine learning and emergent properties, since humans will not be capable of completely specifying, wiring, and programming the entire system. The machines must be able to learn on their own, and then we can expect remarkable properties to emerge.

Figure 1 compares the number of synapses in several living systems [3], [4] to the number of bytes in some current computing devices. The largest computer in the world is the IBM BlueGene [5], with 212,992 processors, 74 terabytes of memory, and a speed of roughly 596 trillion floating point operations/second (teraflops). The human brain has roughly 10^{15} synapses and 10^{11} neurons [4]. It is the number of synapses (not neurons) that are important for computing and memory. The software described in this paper requires only 1 byte/synapse and roughly 1 floating point operation/synapse/timestep. In addition, these neural networks can be trained efficiently using Hebbian-style unsupervised learning. They can also be made to be scalable on massively parallel computers [6]. Supercomputers are approaching the speed and memory of humans, but few people have regular access to more than a few hundred processors on most supercomputers. Even more interesting is the fact that a laptop has roughly the same computing power and memory as a cockroach and a small PC cluster is on the order of a rat, both of which have far more "intelligence" than any robot man has ever built and are readily available to almost all researchers.

In addition to computer memory and speed, however, intelligent machines will need to be embedded in the real world with significant input/output capabilities, feedback loops, and the ability to learn. The human sensory system (touch, vision, smell, hearing, and taste) uses hundreds of millions of cells, and there are roughly 600 muscles in the human body. The DARPA Urban Challenge vehicles [7] are examples of some of the most capable robotic systems that have been built to date, but they have only a few sensor systems (e.g. lasers, cameras, and radar) and only a few motor-control output channels. They also required about 500,000 lines of code. Cognitive architectures (e.g. Soar

[1] Distinguished Professor of Aerospace Engineering, Bioengineering, and Mathematics, AIAA Fellow. 229 Hammond Bldg., LNL@psu.edu, http://www.personal.psu.edu/lnl
[2] Graduate Research Assistant and Ph.D. Candidate, Computer Science and Engineering, 229 Hammond Bldg.

and ACT/R) have been implemented on mobile robots also [8], but these too have very low memory, speed, learning ability, and input/ouput. While Moravec [2] compares living creatures and man-made systems in terms of memory and speed, these are only two of the requirements for an intelligent machine.

We may be able to achieve human-level processing power on current supercomputers, but it is not clear how to build (or "wire") the complex networks or how the networks should be connected to the sensors and motor-control devices, and manually designing and building these is beyond the capabilities of humans. It is unlikely that traditional artificial intelligence approaches will lead to truly intelligent machines. Rule-based systems and cognitive architectures require humans to program the rules, and this process is not scalable to billions of rules. The machines will need to learn on their own by experimenting in the real world. Human-level intelligence (and possibly consciousness) will most likely be produced as an emergent property of an embedded learning machine.

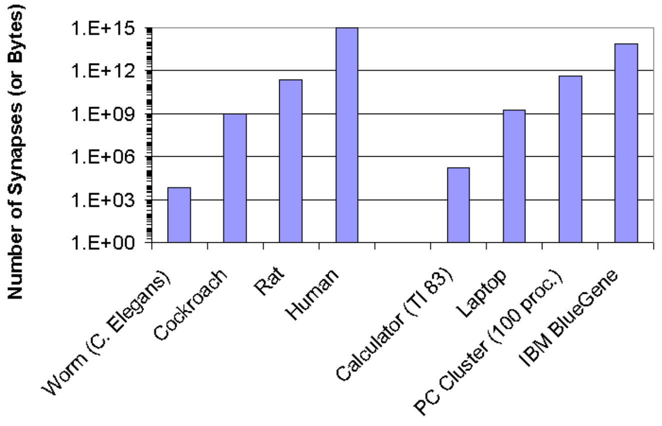


Figure 1. Comparison of the number of synapses (or bytes)

Even computer vision systems are very far from being able to match human vision capabilities in many scenarios. In order to develop better computer vision systems, it is imperative that we learn from the human system. So, in the next few paragraphs we summarize the important components and processing of the mammalian vision system, which uses 25% of the human brain [9].

A. Biological Vision Systems

The human vision system is a remarkable system. Wandell [10] and Koch [11] give very good overviews of human vision, which we briefly review here. In mammals, light enters the eye through the lens and impinges on the retina. The output of the retina goes through the optic disk via 1.5 million retinal ganglion cell axons [11]. The signals from these cells go into about 20 visual streams, and they all get a fairly complete image from the retina. Each one specializes in a different type of visual information processing, and they send their spatiotemporal information to the appropriate regions of the brain for processing. There are 3 main visual streams: one associated with eye movement, one for spatial variations (ventral stream), and one for temporal variations (dorsal stream). There are also special regions for color processing.

There are roughly 5 million cones and 100 million rods in each retina. The cones are used in high light levels (photopic) and the rods are for low light (scotopic). In the center of the retina there is the fovea which has the highest concentration of cones, and no rods. It has a diameter of approximately 1.5 mm, which corresponds to about 5.2 degrees of view angle (about half the size of a fist at arms length). A cone is about 2.5 microns in diameter and there are about 50,000 in the fovea (roughly 225 by 225). There are a variety of cones for different wavelengths which have peak responses at wavelengths of roughly 575, 540, and 450 nanometers (yellow, green, and blue have wavelengths of 575, 540, and 480 nm, respectively).

The most important information in the visual stream is contrast, not light levels, and neurons change their response to achieve good measures of contrast. Humans can see over 6 orders of magnitude in light level (moonless night to bright sunny day). The system is incredibly efficient, and is able to achieve this performance with a system that encodes information using approximately 1 byte (or less) at a time. This occurs partly through pooling the signals from many rods to enhance the sensitivity.

The retinas (of mammals) are primarily connected to an area of the brain called the lateral geniculate nucleus (LGN), which has six layers. The retinal ganglion cells (RGC) encode information into neuronal action potentials. (~50 Hz for a uniform level of light). Most exhibit on-center, off-surround or off-center, on-surround behavior. The RGC responses are not separable in space-time. The "difference of Gaussians," has two 2-D Gaussian filters one for the center and one wider one for surrounding area is a good model for these cells, but the surround portion responds slower than center portion.

After the LGN, the signals are sent to the cortex, which is about 1400 cm^2 and is about 2 mm thick. Roughly 55% of the human neocortex (and 25% of the entire brain [9]) is devoted to vision (somatosensory, motor, and auditory use about 11%, 8%, and 3%, respectively) [12]. Signals from the LGN go mainly to Area V1 or the primary visual cortex (PVC). V1 has about 10^8 neurons (24 cm^2) in 6 layers, and the LGN has about 10^6 neurons. In V1 layers 2, 3, and 4 (mainly 4) receive input from LGN. Layer 6 sends output back to the LGN. Also, signals in LGN and V1 remain retinotopically arranged. Artificial electrical stimulation of V1 can cause sensations of vision.

Hubel and Wiesel (1959-1982) [13] [14], performed very clever experiments and discovered simple and complex cortical cells (they received a Nobel prize for this work in 1981). Simple cells respond to small spot stimuli, but complex cells are nonlinear and do not satisfy superposition. Simple cells have adjacent excitatory and inhibitory regions. These are found mainly in layers 2 and 3 of V1. They also found that groups of neurons respond to bars and lines at different angles (a response that is similar to using a Gabor filter [15]). Also, some neurons in V1 respond to bars moving in a certain direction, but not to bars moving in other directions (or even in the opposite direction). The "wiring diagram" beyond V1 is extremely complex (see: [12]), which is why machine learning is so crucial. It is unlikely that we will ever be able to manually build and "wire" a human brain, but there is a chance that an artificial brain can evolve and learn, if built properly.

B. Computer Vision

In the software developed here, we are trying to roughly model how the human vision system operates (obviously it is not a complete model and is greatly simplified). The present discussion is not complete, however, unless we present some topics in computer vision [16] which are often used to solve problems such as edge detection, corner detection, and object recognition. In the next few paragraphs we present some important computer vision concepts and algorithms, and give some key references. This section is not essential for the reader, but any biologically inspired vision system will need to compete with state of the art computer vision systems which use algorithms such as these.

The Sobel operator [16] can be used to detect horizontal or vertical edges, but the Canny edge detector [17] is one of the most well known edge detectors in the computer vision community. It formalizes the steps of: 1) Noise smoothing, 2) Edge enhancement, and 3) Edge localization, to design an optimal edge detector. Canny proposed a linear continuous filter that maximizes these desirables, and looks very similar to a derivative of Gaussian filter. Corner detection is another important problem in vision which is applicable in stereo vision, image stabilization etc. Harris corner detection [18] is one of the well known algorithms in this area. In this algorithm, one computes an "R-score" and sets a threshold above which all pixels are classified as corners. The Hough transform [19] can be used to find simple shapes (e.g. lines, circles, etc.) in images.

Gabor filters [15, 20] are another important class of filters in vision as they represent properties of certain cells in the mammalian visual cortex. Mathematically the response can be defined as a product of a cosine and a Gaussian:

$$g(x,y;\lambda,\theta,\sigma,\gamma)=\cos\left(\frac{2\pi(x\cos\theta+y\sin\theta)}{\lambda}\right)\exp\left(-\frac{(x\cos\theta+y\sin\theta)^2+\gamma^2(-x\sin\theta+y\cos\theta)^2}{2\sigma^2}\right)$$

Here, λ is the wavelength of the cosine, θ is the orientation, γ is the aspect ratio defining the ellipticity of the response, and σ defines the extent (standard deviation). These have been used in many vision applications such as edge detection, target detection, and texture segmentation [21, 22]. This filter is named after Dennis Gabor, who won a Nobel prize in 1971 for inventing holography.

In using functions such as Gabor filters for image processing, it is crucial for performance that the function be separable into two 1-D filters. For the case where γ=1 we can show that the Gabor filter is separable. If the cos function is replaced using a trigonometric identity and the exponential terms are expanded and simplified, then it can be rewritten as:

$$g(x,y;\lambda,\theta,\sigma)=\left[\cos\left(\frac{2\pi x\cos\theta}{\lambda}\right)\cos\left(\frac{2\pi y\sin\theta}{\lambda}\right)-\sin\left(\frac{2\pi x\cos\theta}{\lambda}\right)\sin\left(\frac{2\pi y\sin\theta}{\lambda}\right)\right]\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

which can be rewritten in a separable form:

$$g(x,y;\lambda,\theta,\sigma)=\cos\left(\frac{2\pi x\cos\theta}{\lambda}\right)\exp\left(\frac{-x^2}{2\sigma^2}\right)\cos\left(\frac{2\pi y\sin\theta}{\lambda}\right)\exp\left(\frac{-y^2}{2\sigma^2}\right)-$$
$$\sin\left(\frac{2\pi x\cos\theta}{\lambda}\right)\exp\left(\frac{-x^2}{2\sigma^2}\right)\sin\left(\frac{2\pi y\sin\theta}{\lambda}\right)\exp\left(\frac{-y^2}{2\sigma^2}\right)$$

The standard Gabor filter would require O(N^2) operations per pixel, where N is the size of the filter used (e.g. 7 to 35). The separable form would only require O(2N) operations. The above can be used to find regions in images that correlate to Gabor filters by performing convolutions of the image pixel data with Gabor filters. We also plan to use these to set the weights in some of our neural networks (which is similar to what occurs in human vision systems).

Optic flow algorithms are useful for avoiding obstacles, pattern recognition, video compression, etc. The Lucas-Kanade method [23] is one of the well known differential algorithms for computing optical flow. It is assumed that the pixel intensity is almost constant from one frame to another. The algorithm works well for estimating flow at corners but does not produce good results near low texture regions, or regions with edges due to the aperture problem.

An object recognition algorithm to recognize known objects from unknown viewpoints is based on Scale Invariant Feature Transforms (SIFT)) keys or features [24]. In this method, image features are transformed into local feature coordinates that are invariant to rotation, translation, and scaling. This involves four major steps consisting of 1) Scale-space extrema detection (searching through scale and space to find potential interest points in the scene) 2) Key-point localization (fitting of a detailed model to determine location and scale) 3) Orientation assignment (local image gradient directions are used to assign one or more orientations to each key-point location), and 4) Key-point descriptor (the local image gradients are measured at the selected scale in the region around each key-point and transformed into representation that allow for local shape distortion and illumination changes). A large number of keys can be extracted from an image, resulting in recognition of small objects even when occluded.

Another object recognition approach is based on Principal component analysis (PCA) [25]. In this method each of the templates to be matched is stored in a compact form for all possible scales, rotations etc. As the number of templates can be very large they can be stored in a compact form using Eigenanalysis by exploiting the fact that they tend to be very similar. For face recognition, a similar approach can be followed, computing a mean face and storing Eigenfaces and coefficients obtained from PCA. Thus, all the faces can be stored compactly and each face can be approximately reconstructed. However, such methods have problems with occlusion and noise.

Recently, Riesenhuber and Poggio [26] and others have developed image recognition software and algorithms that are both scale and position invariant. In this approach there are multiple layers, which perform operations such as Gabor filtering and MAX pooling operations. These have been very effective for object recognition problems. We plan to attempt to duplicate this sort of approach, but using biologically-inspired neural networks.

II. Neurons and Neural Networks

A. Introduction to Neural Networks

Artificial neural networks (ANN) can broadly be classified into three generations. The first generation consisted of McCulloch and Pitts neurons which restricted the output signals to discrete '0' or '1' values. They have been used for building networks such as multi-layer perceptrons, and can model any function with a binary output using a single hidden layer [27]. The second generation models use a continuous activation function allowing for the output to take values between '0' and '1'. This makes them more suitable for analog computation, and at the same time requires fewer neurons for digital computation than the first generation models. They are more powerful than their digital counterparts as they can also approximate any analog function [27]. As they use a continuous activation function, the training can generally be achieved by using a gradient descent algorithm such as backpropagation [28]. Backpropagation is a supervised learning technique that allows for adjustment of the weights given the desired output. One way to think of the analog outputs between 0 and 1 is in terms of normalized firing rates or frequencies in a certain time interval. This can be called as a rate-coding scheme and implies a hidden averaging mechanism. As can be seen, this is conceptually different from the workings of real cortical neurons which work in terms of spikes.

In a previous paper [6], we showed how these 2nd generation models could be made scalable and run efficiently on massively parallel computers. In that work, we developed an object-oriented, massively-parallel ANN software package SPANN (Scalable Parallel Artificial Neural Network). MPI was used to parallelize the C++ code. The back-propagation algorithm was used to train the network. The software was used to identify character sets consisting of 48 characters and with various levels of resolution and network sizes. The code correctly identified all the characters when adequate training was used in the network. The training of a problem size with 2 billion neuron weights on an IBM BlueGene/L computer using 1000 dual PowerPC 440 processors required less than 30 minutes. Various comparisons in training time, forward propagation time, and error reduction were also made. These, however, are not biologically plausible neural networks, even though they have proven useful in numerous engineered systems.

Another relevant approach is the Neocognitron [29]. This approach is supposed to emulate the findings of Hubel and Wiesel, including simple and complex cells. It is a multi-layered neural network with unsupervised learning capabilities.

Spiking neural networks belong to the third generation of neural networks and like their biological counterparts use spikes or pulses to represent information flow. They can use spatial-temporal information in communication and computation similar to biological neurons. This information is encoded both in the timing as well as the rate of pulsation, i.e. pulse coding. The motivation behind exploring the spiking neuron models is that temporal information can also be encoded in the input signals and multiplexing can be achieved using pulse coding [27]. This makes the spiking neuron models more versatile and powerful than their non-spiking counterparts. Also, spiking ANNs are more biologically plausible than traditional ANNs. The software required for these neural networks are no more difficult to parallelize than that used for the 2nd generation networks.

In [30], we developed a spiking neural network model to identify characters in a character set. The network is a two layered structure consisting of integrate-and-fire and active dendrite neurons. There are both excitatory and inhibitory connections in the network. Spike time dependent plasticity (STDP) was used for training. The winner take all (WTA) mechanism was enforced by the lateral inhibitory connections. Most of the characters were recognized in a character set consisting of 48 characters. The network was trained successfully with various resolutions of the characters. Also, the addition of uniform random noise did not decrease its recognition capability.

B. Neuron Models

The neural networks used here are biologically plausible time-dependent spiking neural networks, which emulate the complex circuitry of the human brain. The most widely known and detailed model of a biologically-realistic neuron is due to Hodgkin and Huxley (H-H) [31, 32] (they won a Nobel prize in 1963). This model is often used in neuroscience, but is seldom used in engineering applications since it is very expensive to simulate. The H-H equations are:

$$C\frac{dv}{dt} = -\sum_k I_k(t) + I(t)$$

$$\sum_k I_k = g_{Na}m^3h(v-E_{Na}) + g_K n^4(v-E_K) + g_L(v-E_L)$$

$$\dot{m} = \alpha_m(v)(1-m) - \beta_m(v)m$$

$$\dot{n} = \alpha_n(v)(1-n) - \beta_n(v)n$$

$$\dot{h} = \alpha_h(v)(1-h) - \beta_h(v)h$$

Here v is the voltage across the membrane, C is the capacitance, ΣI_k is the sum of the ionic currents which pass through the cell membrane, Na, K, and L denote the three types of channels; m, n, and h are called the gating variables; E, g, α, and β are other parameters which are listed below:

$$E_{Na} = 115mV, E_K = -12mV, E_L = 10.6mV$$

$$g_{Na} = 120mS/cm^2, g_K = 36mS/cm^2, g_L = 0.3mS/cm^2$$

$$\alpha_n = \frac{(0.1-0.01v)}{e^{(1-0.1v)}-1}, \alpha_m = \frac{(2.5-0.1v)}{e^{(2.5-0.1v)}-1}, \alpha_h = 0.07e^{-v/20}$$

$$\beta_n = 0.125e^{-v/80}, \beta_m = 4e^{-v/18}, \beta_h = \frac{1}{e^{(3-0.1v)}+1}$$

These equations are quite complex and very expensive to solve numerically, as discussed later. Another well known, but much simpler, model of a neuron is a leaky integrate and fire (LIF) neuron [32]. For neuron i in a system of neurons, this would be modeled by:

$$\frac{dv_i}{dt} = \frac{1}{RC}\left((I_{input} + I_i)R - v_i\right)$$

where v is voltage, R is resistance, C is capacitance, and I_{input} is a possible input current (usually zero), and I_i is the current from the synapses. Incidentally, in biology, it is more common to use the term conductance, $g = 1/R$, rather than resistance, R. The current from the synapses can be modeled as:

$$I_i = \alpha \sum_{j=1}^{N} w_{ij} \sum_{k=1}^{M} \delta(t - t_{jk})$$

Where N is the number of presynaptic neurons and M is the number of times the j^{th} presynaptic neuron has fired since the i^{th} neuron fired. The coefficients w_{ij} and α, represent the synapse weights and the increment of current injected per spike, respectively. The above is a fairly simple differential equation, and an exact solution (for a single neuron and constant I) is [32]:

$$v(t) = I R - \left(I R - v_o \right) e^{-t/\tau}$$

Where $\tau = RC$ and v_o is the initial condition. Typically in the LIF model once the voltage crosses some threshold a spike is inserted, followed by decay, a step decrease, and/or a refractory period. There are also more complex models [33], but the LIF model is fast and captures the essence of many applications.

From the above solution, one can show that the firing rate of the neuron will be given by [32]:

$$f = \frac{1000}{T_{th} + T_{ref}} \quad \text{where,} \ T_{th} = -\tau \ln\left(\frac{V_{th} - I R}{V_r - I R} \right)$$

T_{ref} is the refractory period and all the times are in milliseconds. This is useful for computing the frequency of spiking given the characteristics of the neuron: current (I), threshold (V_{th}) and resting voltages (V_r), resistance R, capacitance C, refractory time T_{ref}, and τ (which is $dt/(R*C)$). It can also be used to verify that the neuron simulation is being computed properly. The maximum frequency will be $1000/T_{ref}$, or 500 Hz if T_{ref} is 2 ms. One can also show if the current is 10% above the threshold current, the frequency will be given by:

$$f_{I/10} = 1000 / (T_{ref} + 2.4\tau)$$

(assuming V_r=0). For T_{ref} =2 ms and τ = 8 ms, this yields 50 Hz. So for neurons with these properties, the firing rate will range from roughly 50 to 500 Hz. Traditional 2nd generation rate-based neural networks often implement a sigmoid function at the neurons that ensures that the output will always remain between some upper and lower bounds, for all range of inputs. In effect, the spiking neurons have this same feature. If the current is below the threshold current (no matter how small), the firing rate will be zero. In addition, no matter how large the current is, the firing rate will not be above the maximum rate.

Figure 2 shows the voltage time history from a Matlab simulation of a single neuron. It compares the behavior of the Hodgkin-Huxley model and the Leaky Integrate and Fire (LIF) model. A simple first-order Euler method is used to integrate the equations, with a time step size of 1 millisecond. It shows the response to a step increase and decrease in input current. The start-up and stopping transients are significantly different, but the periodic signals are very similar. The current is turned on at 3 seconds (6 seconds for LIF) and is turned off at 35 seconds. The LIF model also has a linear decay after the spike, but the LIF model can easily include an exponential decay model as well. The agreement between the two models is quite remarkable, especially considering how simple the LIF model is compared to the H-H model. The H-H simulation required roughly 120 CPU seconds and the LIF simulation required only 0.02 CPU seconds. The H-H model requires computing (at every time step) several exponentials, which are very expensive. H-H also requires significantly more memory, since there are three ODE's and many variables to store. The details of the neuron behavior is probably not crucial if one is primarily interested in developing engineering neural network systems, and so LIF is used in this application.

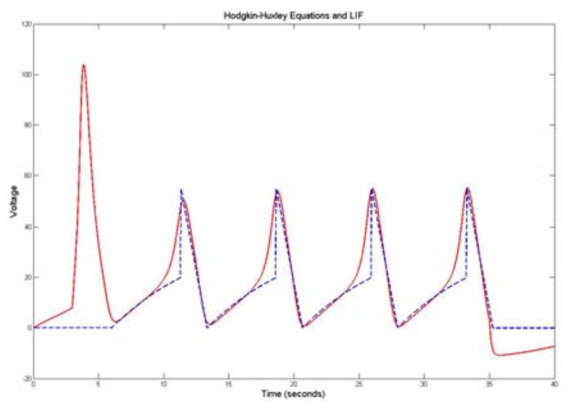


Figure 2. Voltage time history of the Hodgkin-Huxley Neuron (solid line) compared to simple Leaky Integrate and Fire Neuron (dashed line).

In some spiking neural network papers, there is little mention of the time step size used or its importance. A small time step size is crucial to obtaining an accurate, converged, and repeatable solution. In computational fluid dynamics and computational aeroacoustics [34] algorithms have been developed that minimize phase and amplitude errors in numerical simulations to partial differential equations. Often these are fourth order accurate (or higher) schemes, which means the error is proportional to Δt^4. An Euler explicit scheme is O(Δt), which means very small time steps must be used to obtain an accurate solution.

Figure 3 shows the well-known frequency vs. current curve for a single LIF neuron. The solid line is the theoretical solution, and the other lines show the solution from an explicit Euler algorithm for three different time step size (1.0, 0.1, and 0.01 milliSec.). Only the smallest time step agrees well with the theoretical solution. At the highest frequency (about 200 Hz) the period is 5 mSec, and a time step size of 0.01 mSec corresponds to 500 time steps per period. Using a time step size of 1 mSec. corresponds to 5 points per period, and cannot accurately simulate 200 Hz. Higher order accurate methods (e.g. Adams-Bashforth) would be difficult to implement, since they usually do not work well with very large gradients and often are not self starting (and the neuron signal is continually restarting). Using the exact solution over that portion of the waveform where it is applicable does not really help very much either. This would give an algorithm such as:

$$v(t+\Delta t) = IR - (IR - v(t))\, e^{-\Delta t/\tau}$$

If R=40 megaOhms and C=0.2 nF, then τ=8 mSec. Expanding the exponential term in a Taylor series expansion yields:

$$v(t+\Delta t) = v(t) + \left(\Delta t/\tau + (\Delta t/\tau)^2 + \cdots\right)(IR - v(t))$$

which is essentially the same as the explicit Euler algorithm:

$$v(t+\Delta t) = v(t) + \left(\Delta t/\tau\right)(IR - v(t))$$

for small values of $\Delta t/\tau$, which will usually be the case at frequencies near 200 Hz (since the period itself is 5 mSec.). Even if you only use 10 steps per period, then $\Delta t/\tau=0.0625$ and $(\Delta t/\tau)^2=0.004$. Therefore, using the exact solution produces results essentially the same as those shown in Figure 3; and it requires very small time step sizes as well. Also, the exponential is very expensive to compute. A good rule of thumb is that the we need roughly 100 steps per period, or $\Delta t < 10/f$, where f is the maximum frequency (Hz) expected and Δt is in milliseconds. It should also be mentioned here that during one period of the neuron firing rate there could also be complex phenomena occurring at even smaller scales due to the input from the synapses.

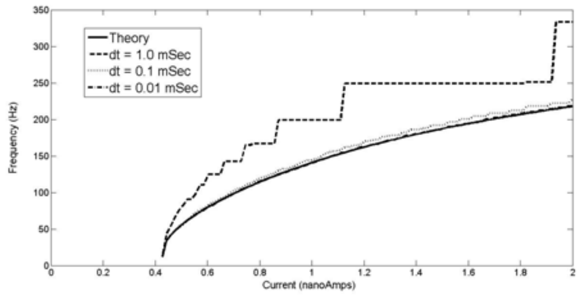


Figure 3. Numerical solutions for a single LIF Neuron (using different time step sizes) compared to Theory.

C. Spiking Neural Network Software

A number of systems have been developed in the past for simulation of spiking neural networks [35]. They vary in the degree to which they represent biologically realistic neurons, support parallel computing, their complexity, operating system support, performance, etc. The next few paragraphs briefly discuss some of these systems.

GENESIS [36] was designed to be a generic simulation system for building a biologically realistic neural network simulator. It has been used for biochemical reactions as well as more realistic small and large neural networks [37]. It includes a variety of multi-compartmental models with many Hodgkin Huxley (H-H) or calcium dependent conductance models. Simple neuronal models, such as Integrate & Fire (IF) and those by Izhikevich [33], are not realistic enough for the intended use of GENESIS, these are not included in GENESIS by default (although one can construct them). It also offers its graphical interface XODUS. The GENESIS core code and XODUS are both written in C. To get maximum benefit one has to learn and write code in the GENESIS scripting language. PGENESIS [35] is the parallel version of GENESIS and has support for both MPI and PVM. It can be run on a variety of platforms including Windows with Cygwin, OS/X, Linux on both 32 and 64 bit architectures.

The neo-cortical simulator (NCS) [38] was designed to be mainly a mammalian brain simulator with large networks of H-H type biologically realistic neurons. Its current version (version 5) is written in C++ and MPI. NCS can run on any Linux cluster and has also been run on a 8000 processor Swiss EPFL IBM Blue Brain machine. The largest simulation run on NCS had about 1 million single-compartment neurons connected by 1 trillion synapses and required about 30 min on 120 CPUs to simulate one biological second [37].

The Neural Simulation Tool (NEST) [39] initiative [35] was started to build networks of large neurons with biologically realistic connectivity and a small number of compartments. The simulator can model different neuron types including IF, H-H, and also different types of synapses including the STDP learning method. The user needs to learn a stack oriented simulation language to build simulation networks. It is written in C++ and MPI and supports parallelization by muti-threading or message passing. It can be compiled and run in a Linux platform with certain required libraries mentioned in the NEST initiative website. For a simulation network of 10000 neurons and 1 million synapses, the speedup obtained was almost linear up to 8 processors.

The spikeNET approach [40] was developed for building large networks of spiking neurons with simple pulse-coupled IF models. The original code was for research purposes and future versions are now commercialized [35]. The source code of the research version of spikeNET can still be downloaded from the spikeNET research webpage. SpikeNET was originally developed for processing only one spike per neuron and it cannot implement synaptic delays. The idea was that the brain could use rank order coding (the order in which neurons fire) to encode information rather than rate coding. The main emphasis was on doing the computation in real time for a variety of image processing tasks such as real time object tracking, face recognition etc. SNVision technology [35] which is based on spikeNET can do tasks such as real time detection in natural scenes, recognition of multiple targets, other recognition tasks etc. The spikeNET code for research purposes was written in C++ and runs on Linux.

NEURON [41] was developed initially to be a simulator for modeling cells with complex ionic channels or with cable properties. A number of papers in the literature have used NEURON as a simulation tool for problems involving cells that obey certain complex branched anatomies or biophysical properties such as multiple ion channels etc. One of its most important capabilities is that it allows modelers to work on a higher level without worrying about computational issues by offering a "natural syntax" such as the concept of a section. Sections are basically un-branched neurites and can be assembled into branched trees. Models are created based on an interpreted language (similar to C) [37], and one can write additional functions in the NMODL language. It offers a GUI interface and modelers without any knowledge of programming can use the GUI to build complex models but often one needs to do some programming to exploit its full capabilities. It also has parallel support and can be run on Beowulf clusters, the IBM BlueGene, and the Cray XT3. A thalamocortical network model by [35] with roughly 5 million equations, 3000 cells, and 1 million connections, showed almost linear speedup up to 800 processors. It is claimed that the speedup obtained is generally linear with CPUs unless each CPU is solving fewer than 100 equations.

As can be seen from the above discussion, present systems do use networks or brain simulators which are medium to high in biological realism [21]. Systems which are more biologically meaningful tend to be more complex and thus require more memory and/or are very slow. We do not want to model the brain but want to build simple engineering solutions for image processing (and other signal processing) tasks such as object recognition using simple LIF models.

III. Software Development

A. Introduction

The software developed here (called JSpike) uses an object-oriented programming approach, and is programmed in Java. We are, however, simultaneously developing a C++ code also, which will be able to run efficiently on massively parallel computers. Java is a very powerful language, and the OOP approach (encapsulation, polymorphism, and inheritance) allows one to develop very understandable and maintainable code. It has many of the advantages of C++, but without many of the problems of C++. It also has some similarity to Ada and allows one to develop more reliable code. Java has many features built into the language that few other languages can claim, such as threads, exception handling, OOP, graphics, graphical user interfaces, and remote method invocation (RMI). In addition, due to the rigorous OOP nature of Java (unlike C++), the language is easily extended and people can easily share libraries.

While Java was initially fairly slow, due to immature compilers and being run in interpreted mode, today the speed is often equal or close to C++. Even if it were a factor of two slower than C++, the advantages of Java would still make it worth using over C++ for many applications. We performed some performance tests of Java. Figure 4 compares the speed of Java and C++ using different computers and different compilers. Even for matrices of size 15000x15000 Java is very close to the performance of C++, and this is without using a just-in-time compiler or a native compiler. On the IBM Power5 computer Java was 18% slower than GNU c++ (g++) (but Java from Sun and and the IBM xlC compiler were the same speed), and on the Intel Xeon 3Ghz [21], Java was only 7% slower than g++. This is quite acceptable to us, since Java offers so many advantages over C++. There are also great free compilers and integrated development environments (IDE), such as Javac (from Sun), gcj (from GNU), Eclipse and NetBeans.

Using an OOP approach allows one to efficiently develop very complex software systems. In this case, we were able to first develop a Neuron class, and we could thoroughly debug that piece before moving on to the higher level functions of the code. This approach has worked well for us in other applications as well, for example a flapping wing aerodynamics C++ code [42], a Monte Carlo C++ code for gas dynamics [43], and a Java robotics code [8]. All of these codes would have taken far longer to develop in FORTRAN or C, and would have resulted in unmaintainable code when completed. Given the power of Matlab, there is really no reason to use FORTRAN, since FORTRAN cannot produce software with the maintainability or features of Java or C++. On the other hand, far too many researchers never venture beyond Matlab, and this often severely limits what they can accomplish. Matlab is fantastic for small codes and for prototyping, but researchers will be limited if it is their only programming option. Long [44] discusses the importance of software engineering, and the need for increased education in this area.

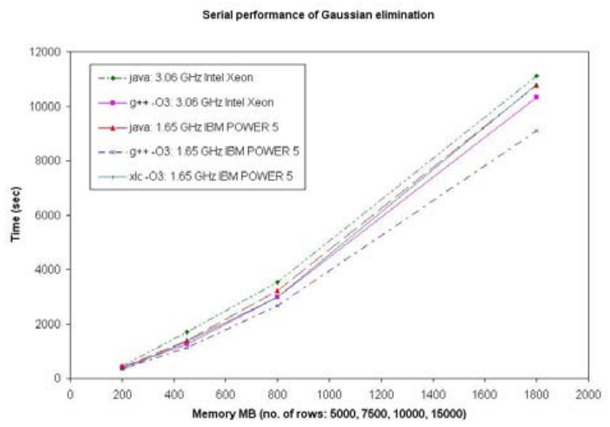


Figure 4. Comparison of Java and C++ for Gaussian elimination on several types of computers using different compilers.

The ability to encapsulate data and methods in Java, and in particular the ability to encapsulate a main method with every class, is especially useful in Java. For each Class we develop we write a main method that can be used to test the Class, and this main method forever stays with the Class. So if changes are made at a low level, they can be tested at a low level first, and then gradually incorporated into the higher classes as it is verified and validated.

B. Neuron Class

The Neuron class in JSpike models a single neuron. A list of the key data and methods associated with the Neuron class is shown in Table 1. This class includes about 300 lines of code, including comments and blank lines. The file size of the code is only 14K bytes. The use of static variables is very useful to making the code more understandable and reduces memory requirements. Each neuron stores the data required for the LIF type neuron (e.g. resistance, capacitance, resting potential, threshold potential, refractory time, time step size, and the alpha value used in synapses). Each neuron also keeps track of the last two times it fired (timef and timefold), how many times it has fired (numFirings), and whether it fired during the last time step (spiked). The Neuron object only requires about 40 bytes of data per neuron, which means that roughly 25 million neurons would require only 1 GB of computer memory. The synapses in a network, however, will also require memory, which is discussed later. The primary method is the "step" method, which implements the time marching of the LIF neuron equation. The step method only requires about 5 floating point operations per neuron per time step. A "Poisson" method is used to include noise in the simulation in some cases. This is done by randomly changing the threshold voltage of each neuron, but when the input is from real sensors (e.g. cameras) there is usually no need to add noise. The "getFreqFromCurrent" simply implements the equation [32] presented earlier that relates current to frequency. The "main" method is used to test the Class. One can give the neuron a variety of input currents and the results are displayed in a pop-up graphical window showing the voltage vs. time. There are also a few Get and Set methods, which are typically used in OOP codes.

Neuron Data	Neuron Methods
Static: **Float**: R, C vRest, tau, tref, currentMax, dt, alpha, factor Non-Static: **Float**: v, vThreshold, timef, tinefold, current, **Int**: numFirings Boolean: spiked	Neuron constructor step poisson getFreqFromCurrent main

Table 1. Data and Methods used in the Neuron Class.

Figure 5 shows a simulation from the Neuron Class. For this simulation, u_{th}=140 mV, u_r=20 mV, R=38.3 megaOhms, C=0.207 nanoFarads, t_r=2 mSec, and the time step size was 0.1 mSec. The lower figure shows the input current to the neuron, and the upper figure shows the resulting voltage. This shows the neuron fires more often when the current is larger, and it shows how the voltage decays when the current is set to zero. On the far right of

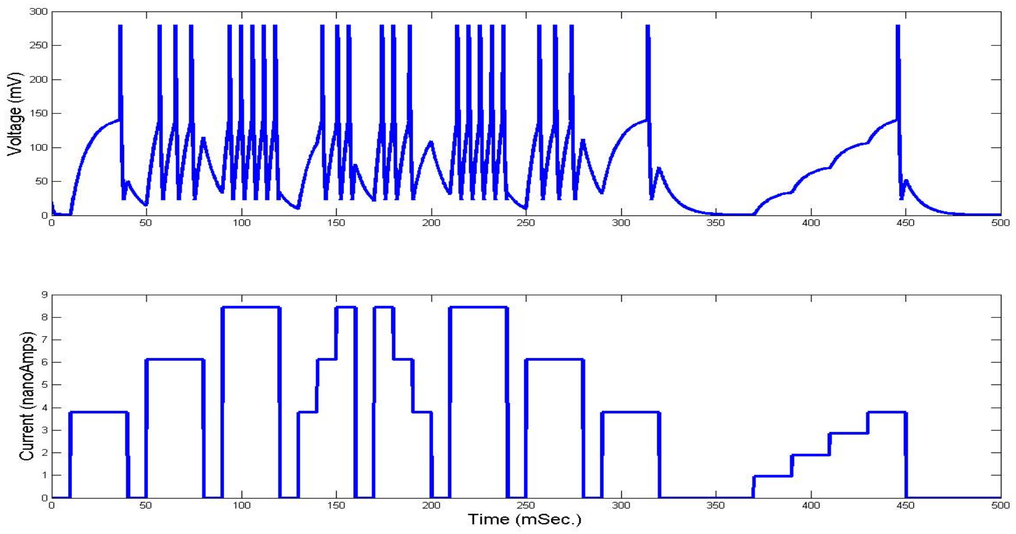


Figure 5. Single neuron response to input current.

the figure it shows the behavior that occurs due to input current from the synapses. Typically a small amount of current is transmitted to the post-synaptic neuron (from pre-synaptic neurons that have fired), and if enough of these occur, the postsynaptic neuron will fire. This simulation uses an exponential decay function after a spike. It also illustrates how complex the output can be, even for rather simple input currents. In the neural network code, the current would come from either the inputs to the network or from the synapses, or an external current can be injected into a neuron as is done in some physiological experiments.

C. Layer Class

Another key part of a neural network are the layers, so we have developed a Layer Class. Each Layer object has a 2-D array of Neurons. This Class contains 800 lines of code (including blank lines and comments). The file size is only 27K bytes. Table 2 shows the primary data and methods of the Layer Class. The main data are the array of neurons and the 4-D array of synapse weights. The weights arrays usually represent the majority of the memory used by the code, so they are stored as bytes (from -127 to 127) and converted to float when needed in the synapse calculations. In most cases, the synapses dominate both the CPU time and the memory required for a neural network simulation. Each synapse has a weight (1 byte), and requires roughly 1 floating point operation per time step (3 if a presynaptic neuron has fired recently). If two layers of size (100x100) were connected using an AllToAll connection, this would require 20,000 neurons (roughly 1 MB) and 100 million synapses (roughly 100 MB). There are several different synapses methods, to accommodate a wide variety of different synapse connections (e.g. all-to-all, one-to-one, stencil, etc.). These methods define the layer-to-layer connections and initial synapse weights. This is essential in order to develop biologically-inspired neural networks, since they are not simply layers of all-to-all neurons.

Layer Data	Layer Methods
Neuron: neurons[][] **Int**: neurX, neurY, synapseType, numNeurons **Byte**: weights[][][][] **Boolean**: doWTA	Layer constructor step stepLearn stepWTA setCurrent setWeights synapses applySTDP applyWTA main

Table 2. Data and Methods used in the Layer Class.

The Layer class is also where learning is implemented. During the learning phase of the run if a neuron fires, Hebbian learning can be used to adjust the weights via unsupervised learning. Hebbian refers to the concept that Hebb proposed in 1949 [45]:

"When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."

Experimental evidence has proven the above idea to be quite accurate [46]. He goes on to say later in the book:

"The old, long-established memory would then last, not reversible except with pathological processes in the brain; less strongly established memories would gradually disappear unless reinforced."

Two difficult issues in developing robust Hebbian learning algorithms are synaptic competition and balancing the growth and decay of the weights. It is important that the synapses compete with each other, but if not done properly all the synapses will converge to the minimum or maximum value.

Hebbian learning is often implemented as spike-time dependent plasticity (STDP) in spiking neuron codes [47]. Previously, we implemented STDP for character recognition [30]. The STDP algorithm modifies the synaptic weights using the following algorithm:

$$\Delta w = \begin{cases} A^{+} e^{\left(\Delta t / \tau^{+}\right)} & , \quad \Delta t \le 0 \\ - A^{-} e^{\left(-\Delta t / \tau^{-}\right)} & , \quad \Delta t > 0 \end{cases}$$

and

$$w = \begin{cases} w_{old} + \Delta w \left(w_{\max} - w_{old} \right) & , \quad \Delta w \ge 0 \\ w_{old} - \Delta w \left(w_{\min} - w_{old} \right) & , \quad \Delta w < 0 \end{cases}$$

Where the time constant τ=20 ms, A^{+}=0.005, A^{-}=0.006, and $\Delta t=(t_{pre} - t_{post})$ is the time delay between the presynaptic spike and the postsynaptic spike. If the presynaptic spike occurs before the postsynaptic spike, it probably helped cause the postsynaptic spike, and consequently we should encourage this by increasing the synaptic weight. And if the presynaptic spike occurs after the postsynaptic spike, then we reduce the weight of the synapse since there was no cause and effect in this case. STDP can be used for inhibitory or excitatory neurons. This is one example of Hebbian-style learning, the increase in synapse weights is larger if the two neurons fire closer together.

The above algorithm is not necessarily the optimal learning approach for spiking networks, even though it has worked well on a number of applications. One issue with STDP involves causality. When a post-synaptic neuron fires in a time-marching code, it is unknown (at that time) whether one of the presynaptic neurons will fire in the future (and at what time). In the laboratory, current can be injected near individual neurons after they fire, but this is not necessarily an efficient computational algorithm. The above algorithm can be easily implemented for very small systems where memory storage is not an issue, but for millions of neurons and billions of synapses we need extremely efficient algorithms that use minimal computer operations and storage, and minimal gather/scatter operations.

Hebbian learning is implemented very efficiently in the JSpike software. It is implemented in essentially an "event driven" manner. That is, if a neuron fires, then the learning method is called by the neuron that fired. This neuron has ready access to all the presynaptic neurons that connect to it. When the postsynaptic neuron fires (and learning is turned on), it can then loop thru all the presynaptic neurons and compute which ones also fired during the time interval between this and the previous postsynaptic firing. Since the current is reset each time a neuron fires, we simply need to know which presynaptic neurons fired between the last two firings of the postsynaptic firings. These are the neurons that connect to the synapses that are strengthened. Any neuron that has not contributed to the postsynaptic neuron firing has its weight decreased. This approach is spike-time dependent, but it is different than STDP. The weight updates are done using:

$$w = \begin{cases} w_{old} + A^{+} e^{\left(-\Delta t / \tau^{+}\right)}, & t_{post_2} \le t_{pre} \le t_{post_1} , \quad \Delta t = t_{post_1} - t_{pre} \\ w_{old} - A^{-} e^{\left(-\Delta t / \tau^{-}\right)} , & t_{pre} < t_{post_2} , \quad \Delta t = t_{post_2} - t_{pre} \end{cases}$$

We have used $\tau^{+} = \tau^{-} = 20$ ms, $A^{+} = A^{-} = 0.02$, In addition, the weights are restricted to the range 0.0 to 1.0. We will continue to refine this, but this approach has worked well in tests so far.

D. Network Class

The Network Class data and methods are shown in Table 3. This Class is 1200 lines of code and 39K bytes. The Network has a 1-D array of Layers (which have 2-D arrays of Neurons), and each Layer can have a different number of Neurons. The Neurons in each Layer can be connected in a

Network Data	Network Methods
<u>Layer</u>: layers[] **<u>Float</u>**: time, inputLayerCurrent[][], **<u>Int</u>**: numLayers, numNeurons, numWeights, numSteps, numLearn, epochs **<u>Boolean</u>**: doWTA[], doLearn[], firings[][],	Network constructor initWeights run step writeWeights writeFiringsRates writeVoltage writeSpikes main

Table 3. Data and Methods used in the Network Class.

wide variety of ways, but for the time being a Layer is only connected to the Layer below it (this would not be hard to change though). The Booleans doWTA and doLearn are used to define which layers will perform winner-take-all and Hebbian learning, respectively. There is an array to hold the input current data (for the first layer). There is also the option to store the simulation results (voltages, weights, firings, etc.). These can be displayed using Java or Matlab.

E. Other Java Classes

A Camera Class is used to get images and pixel data from a webcam. It uses the javax media and imageio libraries from Java. A framegrabber writes the image to a buffer, and then the pixel data can be easily obtained and manipulated (color or gray-scale). The images can be displayed on the screen or written to a file also. This Class is only 500 lines of code, and includes several methods that aren't essential to the neural network vision task.

Since JSpike uses unsupervised learning, the camera can be used to train the network by simply pointing it at images. The pixel data is converted to simulate neuron input currents, somewhat like what occurs in mammalian retinas. The pixel data is typically a byte and ranges from 0 to 255, these are converted to floating point values with a range from some minimum current to some maximum current.

There are also a few other Java classes that we have developed to display graphics and images on the screen. As the neural network is running, we can display spike trains and neuron voltages. This is useful primarily for small systems of neurons and for debugging. These use the Java Swing library, and in particular JPanels and JFrames. For larger simulations the neuron spike histories, synapse weights, and neuron voltage time histories are saved to files. These are displayed after the run using Matlab. The output data files of simulations such as these can be enormous. A one million neuron simulation with a time step size of 0.1 ms would require 40 gigabytes of disk space just to store the neuron voltage time histories for 1 second of real time. The code can do this, but often we simply want to know when each neuron fired. So there is an option in the code to store the time and neuron number when a neuron fires. Since firings occur relatively infrequently, this can save a great deal of storage. The voltage time history data shows more detail (such as the rising voltage before a spike), but a simple scatter plot of firings is almost as useful. Examples of these are presented in Section IV. Storing the synapse weight information is also challenging. Storing a billion weights would require 1 gigabyte of disk space, and storing many of these to study the learning process would require a gigabyte each time. Visualizing all this data is difficult to say the least. Software packages used to visualize computational physics simulations (such as Tecplot, IDL, Ensight, and Fieldview) are almost essential.

And, finally, a Vision Class brings all the objects together into the vision processing system. This consists primarily in a large Main method, but uses all the Objects described above. It first creates a camera object, then it creates a Display object to show the image from the camera on the screen. We can also coarsen the image, so that fewer pixels can be used. The Vision Class then creates a neural network object (which includes Layers and Neurons). The number of layers, neurons per layer, and type of synapse connections can be specified at this time. The number of inputs is determined by the size of the image that will be processed. The Vision class is only 300 lines of code. The main time-stepping loop is inside the Vision Class main method and performs these steps each time step:

1. Get image from Camera
2. Display image on screen
3. Convert pixel data to gray scale
4. Convert gray scale pixels to neuronal input currents
5. Compute current from all synapses
6. Step neurons forward one time step
7. Write data to files (optional)

We could also use color images and feed the RGB data into the network. The use of OOP technology makes this system much more readable, maintainable, and extendable. Most of the code is fairly clear and easy to understand.

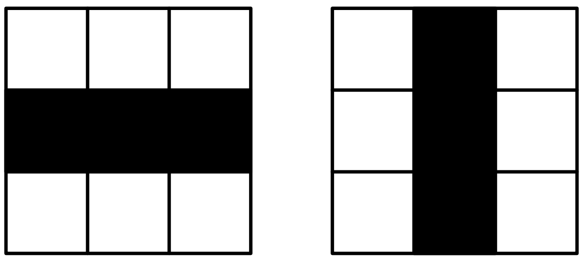


Figure 6 Simple input training patterns.

IV. Results

This section presents the results of some of the simulations. The first results to be shown are simple examples to illustrate the how the code works and how the results can be displayed and interpreted. These results also illustrate the Hebbian learning feature of the code for a simple example. In this example the network consists of 9 input neurons (in a 3x3 array). These input neurons are all connected to one output neuron. The weights of the connections between the input neurons and the output neuron were trained to recognize these bars using the Hebbian approach described above. This is a simple example to illustrate how the approach works. The network was trained on two input patterns: a horizontal bar and a vertical bar, as shown in Figure 6. The network weights before and after learning are shown in Figure 7.

The weights of the input neurons are all set to 1.0, while the synapse weights between the input and output layers are randomly set to a value near 0.5 initially. The input synapses are numbered 1 through 9, while the synapses that were trained are numbered 10 through 18. Synapse 14 corresponds to the center of the 3x3 grid, and this input neuron is always on in the training sets, consequently its weight becomes 1.0. Neuron sets 13 & 15 and 11 & 17 are on roughly 50% of the time in the training data, and those weights become roughly 0.6. The other weights (the corner neurons: 10, 12, 16, and 18) are not on in the training sets so they become zero or very small.

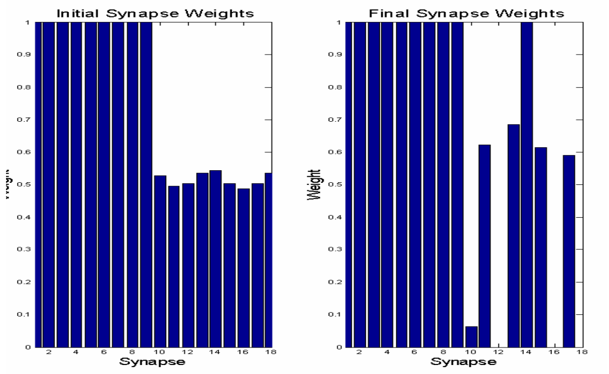


Figure 7 Network weights before and after learning.

Once the above network is trained, it can be given different input data to analyze. Figure 8 shows a few examples of this. It shows how the network responds when shown training and non-training input patterns. The firing rates shown for neurons 1 through 9 illustrate which input pattern was shown to the network. The first two cases are the training data, and the output neuron (neuron number 10) is firing at a high rate (approximately 210 Hz). All nine of the input neurons could be firing, but at different rates. The six pixels that are white do produce a small amount of current, they are not "off." The black pixels produce quite high current values. The third case shown is a diagonal pattern, and the output is approximately 145 Hz. The fourth case even less like the training data (since it is not centered), and its output neuron is spiking at approximately 110 Hz.

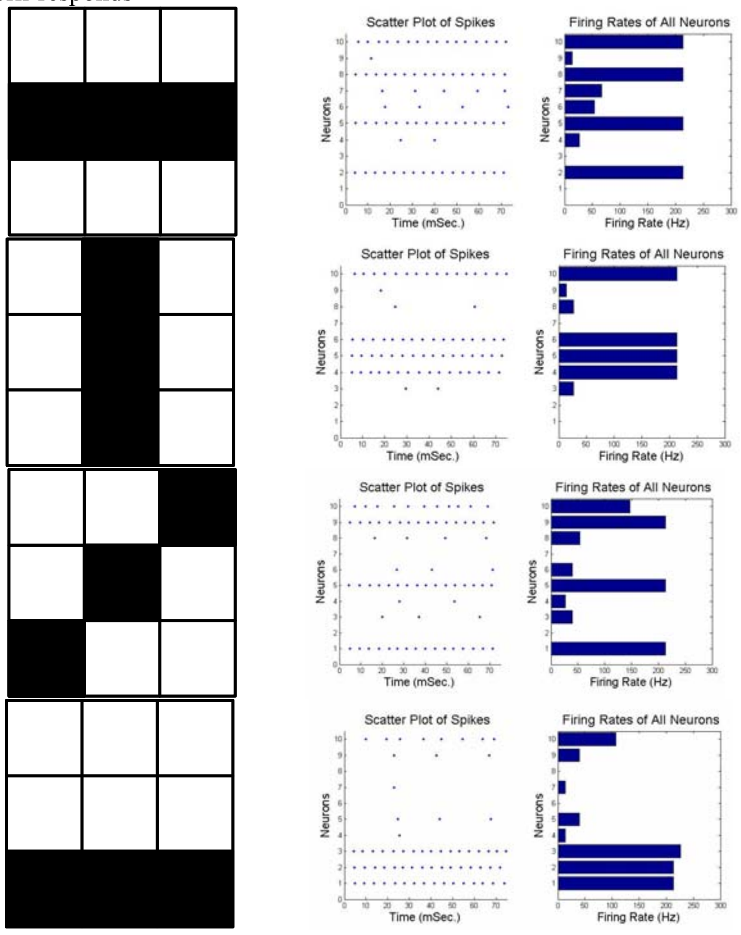


Figure 8. Application of trained network to different input patterns.

Learning can also be applied selectively in these spiking neural networks. We can select which layers should be trained and which should remain constant. We can also train layers one at a time, and build up the network. This is may be especially useful in the first few layers of the network. Just as in mammalian vision systems, where in the early stages of the image processing (in area V1 or before V1) the synapse weights are tuned to simple features such as bright spots and edges. In our networks, the weights of the first few layers can be initialized to Gabor filters as in the approach of [26]. Other layers can perform winner take all pooling operations.

The above results did not use the camera as input, since they used simple arrays of steady-state inputs. The next set of results are for a case that used the camera, but the image was coarsened to 30x30 array of pixels. Figure 9 shows the original image from the camera and two coarsened images (200x200 and 30x30). These were fed to the neural network (the images were taken at the Beckman Institute at Caltech). The software can grab the image from the camera, coarsen it, and then send it to the neural network. The 30x30 pixel data was fed into a neural network with two layers. The first layer had 30x30 neurons (to match the pixels) and the second layer had only one neuron that was connected to all the neurons in the first layer. Figure 10 shows a contour plot of the time histories of the neuron voltages as well as the spike train generated. The spike train data simply shows a dot when a neuron fires, these spikes can be seen in the voltage contour plot also (as red spots).

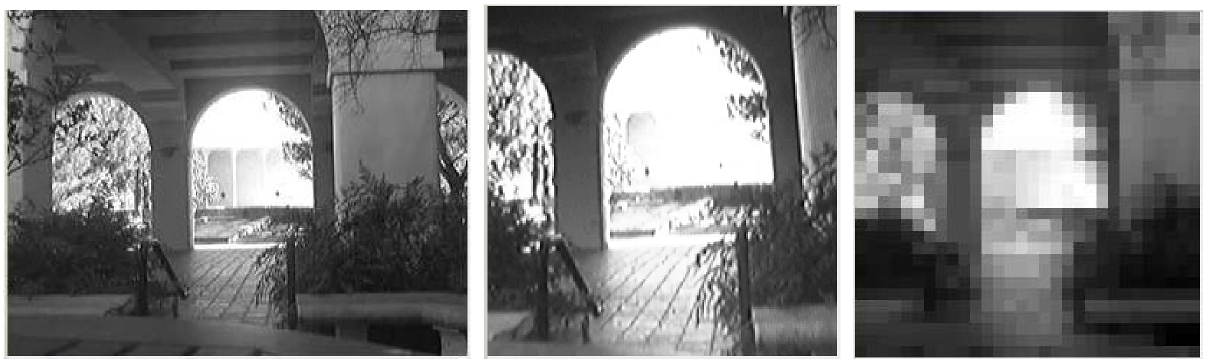


Figure 9. Webcam images used for testing code: 320x240, 200x200, and 30x30 pixels

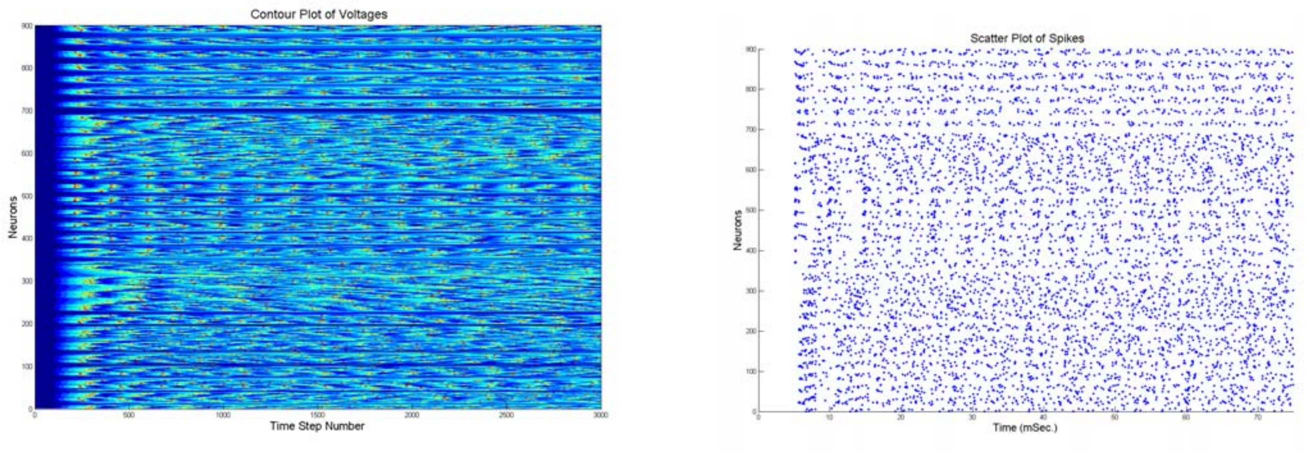
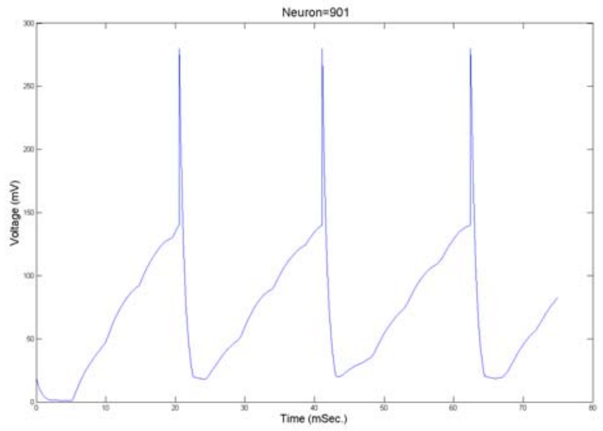


Figure 10. Contour plot of neuron voltages and plot of spike train for 3000 time steps (75 ms of total simulation time).

Figure 11 shows the time history of the neuron in the second layer with and without learning. This is a simple example, but it illustrates the results of Hebbian learning, the neuron in the second layer fires more vigorously (roughly 100 Hz instead of 40 Hz) after learning due to the changes in the synaptic weights. The learning was run 10 times for 3000 time steps each. This illustrates how the system can learn by itself, by simply pointing the camera at a scene.



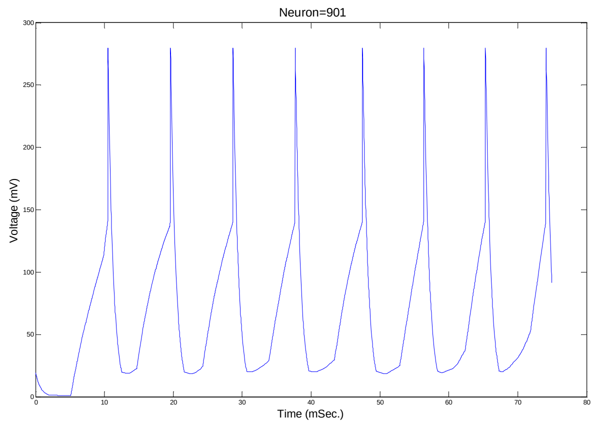


Figure 11. Neuron voltage time history with and without learning. (3000 time steps and 75 ms of simulation time)

Figure 12 shows the spike data for the case where the camera used the 200x200 image, which is roughly the size of the human fovea. This shows when spikes occurred for each of the 40,001 neurons from 5 to 25 ms. The center of the image is quite bright, and those neurons fire quite regularly. This image helps illustrate how difficult it is to visualize the solution when there are a large number of neurons (and this is only 40K neurons). To investigate the details of this solution (beyond just the spike train plot), would require storing and visualizing very large data files.

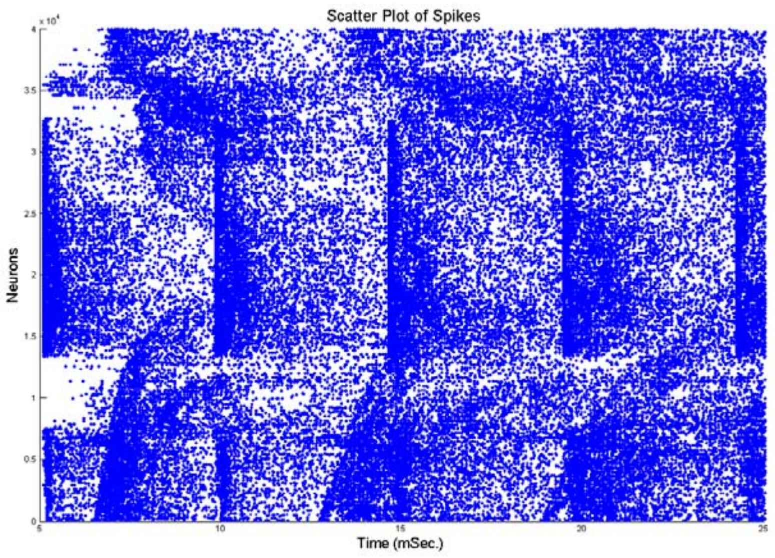
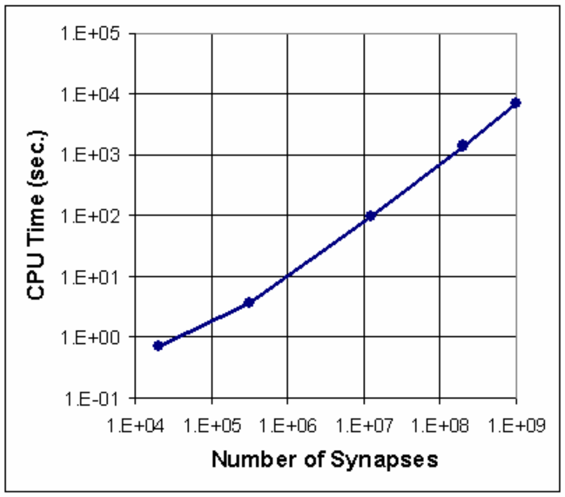


Figure 12. Spike plot for 200x200 pixel image (40,001 neurons)

Figure 13 shows the performance of the code, running on a 1.66 GHz Intel-based laptop with 2GB RAM (a Thinkpad X60) using Java Version 1.6.0_01. Figure 13a shows the CPU time vs. number of synapses. The code was run for 500 time steps (12.5 milliseconds of simulation time). All the networks had three layers. The neurons in the first layer each had only one synapse, but in layers 2 and 3 all the neurons were connected to all the neurons in the previous layer. The number of neurons in these simulations ranged from 300 to 67,500. Other than for very small problems, the performance is linear with the number of synapses. In addition, it shows that due to the low memory requirements the laptop can run up to a billion synapses. Figure 13b shows the performance of just neurons (with no synapses). It too is linear, when more than 10,000 neurons are used. The software was designed to use as little memory and CPU time as possible. Since we expect to use hundreds or thousands of synapses/neuron, the synapse calculations dominate both the memory and CPU requirements. Also, spikes occur relatively rarely, so most of the time steps can be computed quite rapidly with roughly one floating point operation/synapse/step plus some logical operations. In order to minimize memory used, the synapse weights are stored as byte values. When they are needed for calculations, they are converted to floats.



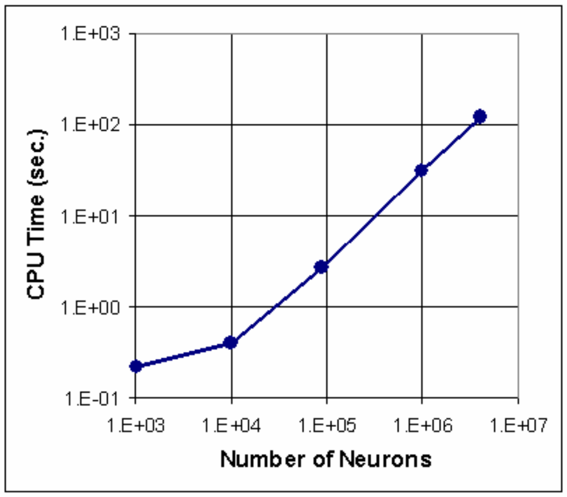


Figure 13a. CPU time dependence on number synapses (number of neurons ranges from

Figure 13b. CPU time dependence on number of neurons (with no synapses).

V. Conclusion

In this paper we have described the algorithms and software for spiking neural network simulations. These neural networks offer the promise of better computer vision systems, as well as the hope of increasing our understanding of biological systems. Some preliminary results were included to demonstrate the algorithms and software. These neural networks can be designed to require minimal memory and processing per synapse, but they do require a large number of very small time steps to march the solutions forward in time. We plan to parallelize these codes and to apply them to more complicated applications in the near future.

VI. Acknowledgments

The first author would like to gratefully acknowledge the California Institute of Technology for supporting him as a Moore Distinguished Scholar, where most of this work was performed. This work was also supported by the Office of Naval Research (Grant No. N00014-05-1-0844).

VII. References

1. Kurzweil, R., *The Age of Spiritual Machines: When Computers Exceed Human Intelligence* 2000: Penguin.
2. Moravec, H., *Robot: Mere Machine to Transcendent Mind*. 1998: Oxford University Press.
3. *Worm Atlas (C. Elegans)*. [cited; Available from: http://www.wormatlas.org/.
4. Churchland, P.S. and T.J. Sejnowski, *The Computational Brain*. 1992: MIT Press.
5. *TOP500 Computer List*. [cited Dec. 31, 2007]; Available from: http://www.top500.org/.
6. Long, L.N. and A. Gupta, *Scalable massively parallel artificial neural networks.* Journal of Aerospace Computing, Information, and Communication, 2008. **to appear**.
7. *Special Issue on the DARPA Urban Challenge.* Journal of Aerospace Computing, Information, and Communication, 2007. **4**(12): p. 1046-1203.
8. Hanford, S.D., O. Janrathitikarn, and L.N. Long. *Control of a Six-Legged Mobile Robot Using the Soar Cognitive Architecture*. in *46th AIAA Aerospace Sciences Meeting, AIAA Paper No. 2008-0878*. 2008. Reno, NV.
9. *Brain Facts: A PRIMER ON THE BRAIN AND NERVOUS SYSTEM*. 2006 [cited Dec. 31, 2007]; 5th:[Available from: http://www.sfn.org/skins/main/pdf/brainfacts/brainfacts.pdf.
10. Wandell, B.A., *Foundations of Vision*. 1995: Sinauer Publishing, MA.
11. Koch, C., *The Quest for Consciousness: A Neurobiological Approach*. 2004: Roberts and Company.
12. Felleman, D.J. and D.C.V. Essen, *Distributed Hierarchical Processing in the Primate Cerebral Cortex.* Cerebral Cortex, 1991. **1**(1): p. 1-47.

13. Hubel, D.H., *Eye, Brain, and Vision*. 1988: W.H.Freeman.
14. Hubel, D.H. and T.N. Wiesel, *Receptive fields and functional architecture of monkey striate cortex.* J. Physiol., 1968. **195**: p. 215-243.
15. Jones, J.P. and L.A. Palmer, *An evaluation of the two-dimensional Gabor filter model of simple receptive fields in cat striate cortex.* Journal of Neurophysiology, 1987. **58**: p. 1233-1258.
16. Trucco, E. and A. Verri, *Introductory Techniques for 3-D Computer Vision*. 1998: Prentice-Hall.
17. Canny, J.F., *A Computational Approach to Edge Detection.* IEEE Trans Pattern Analysis and Machine Intelligence, 1986. **8**(6): p. 679-698.
18. Harris, C. and M.J. Stephens. *A combined corner and edge detector*. in *4th Alvey Vision Conference*. 1988.
19. Ballard, D.H., *Generalizing the Hough transform to detect arbitrary shapes.* Pattern Recognition, 1981. **13**(2).
20. Gabor, D., *Theory of Communication.* Jnl. Inst. Elect. Engr. (London), 1946. **93**(26).
21. Weldon, T.P., W.E. Higgins, and D.F. Dunn, *Gabor filter design for multiple texture segmentation.* Optical Engineering, 1996. **35**(10): p. 2852-2863.
22. Jain, A., N. Ratha, and S. Lakshmanan, *Object detection using gabor filters.* Pattern Recognition, 1997. **30**: p. 295-309.
23. Lucas, B.D. and T. Kanade. *An iterative image registration technique with an application to stereo vision*. in *Proceedings of Imaging understanding workshop*. 1981.
24. Lowe, D.G., *Distinctive image features from scale-invariant keypoints.* International Journal of Computer Vision, 2004. **60**(2): p. 91-110.
25. Nayar, S.K., H. Murase, and S.A. Nene, *Parametric appearance representation In Early Visual Learning*. 1996: Oxford University Press.
26. Riesenhuber, M. and T. Poggio, *Models of Object Recognition.* Nature Neuroscience, 2000. **3**: p. 1199-1204.
27. Vreeken, J., *Spiking neural networks, an introduction*. 2002, Institute for Information and Computing Sciences, Utrecht University.
28. Werbos, P., *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. 1994, New York: John Wiley & Sons.
29. Fukushima, K., *Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition.* Neural Networks, 1988. **1**: p. 119-130.
30. Gupta, A. and L.N. Long. *Character recognition using spiking neural networks*. in *International Joint Conference on Neural Networks*. 2007. Orlando, FL: IEEE.
31. Hodgkin, A.L. and A.F. Huxley, *A quantitative description of ion currents and its applications to conduction and excitation in nerve membranes.* Journal of Physiology, 1952. **117**: p. 500-544.
32. Koch, C., *Biophysics of Computation: Information Processing in Single Neurons*. 1999: Oxford Press.
33. Izhikevich, E.M., *Which model to use for cortical neurons?* IEEE transactions on neural networks, 2004. **15**(5).
34. Ozyoruk, Y. and L.N. Long, *A New Efficient Algorithm for Computational Aeroacoustics on Massively Parallel Computers.* Journal of Computational Physics, 1996. **125**(1): p. 135-149.
35. Traub, R.D. and etal, *A single-column thalamocortical network model exhibiting gamma oscillations, sleep spindles and epileptogenic bursts.* Journal of Neurophysiology, 2005. **93**(4): p. 2194-2232.
36. *GENESIS*. [cited Dec. 31, 2007]; Available from: http://www.genesis-sim.org/.
37. Brette, R. and etal, *Simulation of networks of spiking neurons: A review of tools and strategies.* Journal of Computational Neuroscience, 2006. **23**(3): p. 349-398.
38. *NeoCortical Simulator*. [cited Dec. 31, 2007]; Available from: http://brain.unr.edu/ncsDocs/.
39. *NEST*. [cited Dec. 31, 2007]; Available from: http://www.nest-initiative.uni-freiburg.de/.
40. *SpikeNet*. [cited Dec. 31, 2007]; Available from: http://www.sccn.ucsd.edu/~arno/spikenet/.
41. *NEURON*. [cited Dec. 31, 2007]; Available from: http://www.neuron.yale.edu/neuron/.
42. Fritz, T.E. and L.N. Long, *Object-Oriented Unsteady Vortex Lattice Method for Flapping Flight.* Journal of Aircraft, 2004. **41**(6).
43. O'Connor, P.D., L.N. Long, and J.B. Anderson, *The Direct Simulation of Detonations (Invited Paper)*, in *AIAA Joint Propulsion Conference (AIAA Paper No. 2006-4411)*. 2006, AIAA: Sacramento, CA.
44. Long, L.N., *The Critical Need for Software Engineering Education.* CrossTalk, 2008. **21**(1): p. 6-10.
45. Hebb, D.O., *The Organization of Behavior: A Neuropsychological Theory.* 1949: Erlbaum Pub.
46. Bi, G. and M. Poo, *Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type.* Journal of Neuroscience, 1998. **18**(24): p. 10464-10472.
47. Song, S., K.D. Miller, and L.F. Abbott, *Competitive Hebbian learning through spike-timing-dependent synaptic plasticity.* Nature Neuroscience, 2000. **3**(9): p. 919-926.