Wiley-Series: "Nanotechnology"
Book: "Information Technology"
Editor: Rainer Waser

A Survey Of Bio-Inspired And Other Alternative Architectures

Dan Hammerstrom
Electrical and Computer Engineering
Portland State University

1 Introduction

Since the earliest days of the electronic computer, there has always been a small group of people who have seen the computer as an extension of biology, and have endeavored to build computing models and even hardware that are inspired by, and in some cases, direct copies of biological systems. Though biology spans a wide range of systems, the primary model for these early efforts has been neural circuits. Likewise in this chapter the discussion will be limited to neural computation.

Several examples of this early work include McCulloch-Pitts "Logical Calculus of Nervous System Activity," [2], Steinbuch's "Die Lernmatrix" [3], and Rosenblatt's "Perceptron" [4]. At the same time, an alternate approach to intelligent computing, "Artificial Intelligence" (AI), that relied on higher order symbolic functions, such as structured and rule-based representations of knowledge, began to demonstrate significantly greater success than the neural approach. In 1969 Minsky and Papert [5] of MIT published a book that was critical of the then current "bio-inspired" algorithms, and which succeeded in eventually ending most research funding for that approach. Consequently, significant research funding was directed towards AI and the field flourished. The AI approach, which relied on symbolic reasoning often represented by a first order calculus and sets of rules, began to exhibit real intelligence, at least on toy problems. One reasonably successful application was the "expert" system. And there was even the development of a complete language, ProLog, dedicated to logical rule based inference.

There were a few expert system successes in actually fielded systems, such as Soar [6], whose development was started by Alan Newell's group at Carnegie Mellon University. However, by the 1980s AI in general was beginning to lose its luster after 40 years of funding with ever diminishing returns.

However since the 60s, there have always been groups that continued to study biologically inspired algorithms. And, two of these projects, mostly by being in the right place at the right time, had a huge impact, which re-energized the field and led to an explosion of research and funding. The first was the work [7] of John Hopfield, a physicist at Caltech, who proposed a model of auto-associative memory based on physical principles such as the Ising theory of spin-glass. Although Hopfield nets were limited in capability and size, and others had proposed similar algorithms previously, Hopfield's formulation was clean and elegant. It also succeeded in bringing many physicists, armed with sophisticated mathematical tools into the field. The second event was the "invention" of the back-propagation algorithm by Rumelhart, Hinton, and Williams [8]. Although there too similar work had been done previously [9], what was different about Rumelhart et al., is that they were cognitive scientists and were creating a set of techniques called parallel distributed processing (PDP) models of cognitive phenomena, where back-propagation was a part of a larger whole.

At this point, it is useful to present some basic neuroscience and then a couple of the simpler algorithms that are inspired by this biology. This information will provide a strong foundation for discussing various biologically inspired hardware efforts.

1.1 Basic Neuroscience

This section presents a highly simplified overview of the principles of neural circuits. These circuits consist of large numbers of simple, parallel processing components, the neurons. Neurons tend to be slow, with typical switching times on the order of milliseconds. Consequently, the brain uses significant parallelism rather than speed to perform its complex tasks. Adaptation comes in short and long term versions and can result from a variety of complex interactions.

Although most neurons are exceptions to the canonical neuron shown in Figure 1, the neuron in the figure is sufficiently complex to demonstrate basic principles. Via various ion channels, neurons maintain a constant negative voltage of roughly -70mv relative to the ambient environment. This neuron consists of a *dendritic* tree for taking inputs from other neurons, a body or *soma*, which basically performs charge summation, and an output, the *axon*. Inter-neuron communication is generally via pulses or spikes. Axons form synapses on dendrites and signal the dendrite by releasing small amounts of neurotransmitter, which is taken up by the dendrite.

Axons from other neurons connect via synapses onto the dendritic tree of each neuron. When an axon fires it releases neurotransmitter into the junction between the *presynaptic* axon and the *post-synaptic* dendrite. The neurotransmitter causes the dendrite to depolarize slightly, this charge differential eventually reaches the body or soma of the neuron, depolarizing the neuron.

When a neuron is sufficiently depolarized it passes a threshold which causes it to generate an action potential, or output spike, which moves down the axon to the next synapse. When an output spike is traveling down an axon, it is continuously regenerated allowing for arbitrary fan-out.

While the dendrites are depolarizing the neuron, the resting potential is slowly being restored, creating what is known as a “leaky integrator.” Unless enough action potentials arrive within a certain time window of each other, the depolarization of the soma will not be sufficient to generate an output action potential.

In addition to accumulating signals and creating new signals, neurons also learn. When a spike arrives via an axon at a synapse, it “presynaptically” releases neurotransmitter, which causes some depolarization of the postsynaptic dendrite. Under certain conditions, the effect of a single spike can be modified, typically increased or “facilitated,” where it causes a greater depolarization at the synapse. When the effect an action potential has on the post-synaptic neurons is enhanced, we can say that the synapse is *potentiated* and that learning has occurred. One form of this is called Long Term Potentiation (LTP), since such potentiation has been shown to last for several weeks at a time and may possibly last much longer. LTP is one of the more common learning mechanisms and has been shown to occur in several areas of the brain whenever the inputs to the neuron and the output of the neuron correlate within some time window. Learning correlated inputs and outputs is also called Hebb’s law, named after Donald Hebb who proposed it in 1947. Synapses can also lose their facilitation, one example of this is a similar mechanism called Long Term Depression (LTD), which generally occurs when an output is generated and there is no input at a particular synapses.

Post-synaptic excitation can either be excitatory (an Excitatory Post-Synaptic Potential or EPSP), which leads to accumulating even more charge in the soma, or inhibitory (an Inhibitory Post-Synaptic Potential or IPSP), which tends to drain charge off of the soma, making it harder for a neuron to fire an action potential. Both capabilities are needed to control and balance the activation of groups of neurons. In one model, the first neuron that fires tends to inhibit the others in the group leading to what is called a “winner take all” function.

1.2 A Very Simple Neural Model - The Perceptron

One of the first biologically inspired models was the Perceptron, which, though very simple, was still based on biological neurons. The primary goal of a Perceptron is to do classification. Perceptron operation is very simple, it has a one dimensional synaptic weight vector and takes another, equal dimension, one dimensional vector as input. Normally the input vector is binary and the weight vectors positive or negative integers. During training a “desired” signal is also presented to the Perceptron. If the output matches the training signal, then no action is taken. However, if the output is incorrect and does not match the training signal, then the weights in the weight vector are adjusted accordingly. The Perceptron learning rule, which was one of the first instances of the “delta rule” used in most artificial neural models, incremented or decremented the individual weights depending on whether they were predictive of the output or not. A single layer Perceptron is shown in Figure 2. Basic Perceptron operation is

$$o_j = f(\sum_{i=1}^{n} w_{ij} x_i) = f(W_j^T X)$$

$$O = f(W^T X)$$

Where *f()* is an activation function, it is a step function here (if *sum > 0*, then *f(sum)=1*). But, *f()* can also be a smooth function (more on that later). A “layer” has some number, two or more, of Perceptrons each with its own weight vector and individual output value, leading to a weight matrix and an output vector. In a single “layer” of Perceptrons, each one sees the same input vector.

The Delta Rule, which is used during learning is,

$$\Delta w_{ij} = \alpha(d_j - o_j)x_i$$

Where d_j is the desired output, and o_j is the actual output

The delta rule is fundamental to most adaptive neural network algorithms. Rosenblatt proved that if a data set is linearly separable, the Perceptron will eventually find a plane that separates the set. Figure 3, shows the two dimensional layout of data for, first, a linearly separable set of data and, second, for a non-linearly separable set. Unfortunately, if the data is not linearly separable the Perceptron fails miserably. This was the point of "the book that killed neural networks," *Perceptrons* by Marvin Minsky and Seymour Papert (1968). Perceptrons cannot solve non-linearly separable problems. But also, they do not work in the kind of multiple layer structures that may be able to solve non-linear problems, since the algorithm is such that the output layer cannot tell the middle layer what its desired output should be. We now turn to a description of the multi-layer perceptron.

1.3 A Slightly More Complex Neural Model - The Multiple Layer Perceptron

The invention of the multi-layer Perceptron constituted, to some degree, a "victory" against the "evil empire" of symbolic computing. The Minsky and Papert book focused primarily, through numerous sophisticated examples, on how Perceptrons, as envisioned at that time, could not solve non-linear problems, an excellent example being the XOR problem. Although there are a number of variations, Back-Propagation allowed the use of multiple, non-linear layers, and is sometimes referred to as an MLP (Multi-Layer Perceptron).

The derivation of Back-Propagation is beyond the scope of this chapter, but we can briefly summarize some of the characteristics that allowed Back-Propagation to extend Perceptron like learning to multiple layers. First, instead of a discrete step function for output, we use a continuous activation function. As with the Perceptron there is also a training or "desired" signal. By actually quantifying the error out the output as some function (generally Least Means Square) we create an error surface. We can then find the gradient of the error surface and use that to adjust the weights. By using the chain rule from calculus we can then propagate error back through the various levels.

In summary the steps of the BP algorithm are:

- Present an input vector to the first layer of the network;
- Calculate the output for each layer, moving forward to network output;
- Calculate the error delta at the output (using actual output and the externally supplied target output, *d*);
- Use the computed error deltas at the output to compute the error deltas at the next layer, etc. moving backward through the net; and
- After all error deltas have been computed for every node, use the delta rule to incrementally update all the weights in the network (note, the feedforward input activation values for each connection must be remembered.

Even two level networks can approximate complex, non-linear functions. And this technique generally finds good solutions, which are compact, leading to fast, feed-forward (non-learning) execution time. It has been shown to approximate Bayesian decisions (i.e, it results in a generally good estimate of where Bayes techniques would put the decision surface). However, it can have convergence problems due to many non-local minima. And it is computationally intensive, often taking days to train with complex large feature sets.

1.4 Auto-Association

Another important family of networks are associative networks. One example, giving above, is the Hopfield net. Here we will present a simple associative network developed by G. Palm [10], this is work that was actually done before Hopfield's. One useful variation implements an *auto-associative network* that is an overly simplistic approximation of the circuits in mammalian neocortex. Auto-associative networks have been studied extensively. In its simplest form, an associative memory maps an input vector to an output vector. When an input is supplied to the memory, its output is a "trained" vector with the closest match, assuming some metric, to the given input. Auto-association results when the input and output vectors are in the same space, with the input vector being a corrupted version of one of the training vectors. With *best-match association*, when a noisy or incomplete input vector is presented to the network, the "closest" training vector can usually be recalled reliably. In auto-association the output is fed back to the input, which may require several iterations to stabilize on a trained vector.

Here we assume that the vectors are binary and the distance metric between two vectors is simply the number of bits that are different, i.e., the Hamming distance.

The Palm associative network maps input vectors to output vectors, where the set of input vector to output vector mappings are noted as $\left\{\left(x^{\mu}, y^{\mu}\right), \mu = 1, 2, \cdots, M\right\}$. There are *M* mappings, and both x^{μ} and y^{μ} are binary vectors with size of *m* and *n* respectively. x^{μ} and y^{μ} are sparsely encoded, with $\sum_{i=1}^{m} x_i = l$ ($l << m$) and $\sum_{j=1}^{n} y_j = k$ ($k << n$). *l* and *k* are the numbers of active nodes (non-zero) in the input and output vectors respectively. In training, the "clipped" Hebbian "outer-product," learning rule is generally used, and a binary weight matrix *W* is formed by $W = \bigvee_{\mu=1}^{M} [y^{\mu} \cdot (x^{\mu})^T]$. Such batch computation has the weights computed off-line and then down-loaded into the network. It is also possible to learn the weights adaptively [11].

During recall, a noisy or incomplete input vector $\tilde{x}$ is applied to the network, and the network output is computed by $\tilde{y} = f(W \cdot \tilde{x} - \theta)$, θ is a global threshold, and $f()$ is the Heaviside step function, where an output node will be 1 (active) if its dendritic sum $x_i = \sum_{j=1}^{m} w_{ij} \tilde{x}_j$ is greater than the threshold θ, otherwise it is 0. To set the threshold, the "*k* winners take all (k-WTA) rule" is used, where *k* is the number of active nodes in an output vector. The threshold, θ, is set so that only those nodes that have the *k* maximum dendritic sums are set to "1", the remaining nodes are set to "0". The k-WTA threshold is very close to the minimum error threshold. The k-WTA operation plays the role of competitive lateral inhibition, which is a major component in all cortical circuits. In the BCPNN model of Anders Lansner and his group [11], the nodes are divided into hypercolumns, typically $\sqrt{N}$ nodes in each of the $\sqrt{N}$ columns, with 1-WTA being performed in each column.

An auto-associative network starts with the associative model just presented and feeds the output back to the input, so that the *x* and *y* are in the same vector space and *l* = *k*. This auto-associative model is called an *attractor model* in that its state space creates an energy surface with most minima ("attractor basins") occurring when the state is equal to a training vector. Under certain conditions, given an input vector *x'*, then the output vector *y* that has the largest conditional probability $P(x'|y)$ is the most likely training vector in a Bayesian sense. It is possible to define a more complex version with variable weights, as would be found during dynamic learning, which also allows the incorporation of prior probabilities [12].

1.5 The Development of Biologically Inspired Hardware

With back-propagation and other non-linear techniques in hand, people began solving more complex problems. Concurrent to this there was an explosion in neuroscience that was enabled by high performance computing and sophisticated experimental technologies, coupled with an increasing willingness in the neuroscience community to begin to speculate about the function of the neural circuits being studied. As a result, research into artificial neural networks of all kinds gained considerable momentum in the late 80s, continuing until the mid-90s where research results began to slow down. However, like AI before it and Fuzzy Logic, which happened concurrently, ANNs had trouble scaling to solve the hard problems in intelligent computing. Nevertheless ANNs still constitute an important area of research and ANN technologies play a key role in a number of real world applications [13, 14]. In addition, they are responsible for a number of important breakthroughs.

During the heady years of the late 80s and early 90s, while many researchers were working on theory, algorithms, and applications, other groups began to look at hardware implementation. There quickly evolved three rough schools of thought, though with imprecise dividing lines between them. The first was to build very specialized analog chips, where, for the most part, the algorithms are hard-wired into silicon. Perhaps the best known was the aVLSI (low power analog VLSI) technology developed by Carver Mead and his students at Caltech.

The second was to build more general, highly parallel digital, but still fairly specialized chips. Many of the ANN algorithms were very compute intensive, and it seemed that just speeding up algorithm execution, especially the learning phase, would be a big help in solving harder problems and in the commercialization of ANN technology.

Also, in the late 80s and early 90s, these chips were significantly faster than main-stream desktop technology. However, this second group of chips incorporated less biological realism than the analog chips.

The third option was to use off-the-shelf hardware, DSP and media chips, which ultimately ended up being the winning strategy. This approach was successful because these chips were used in a broader set of applications and had manufacturing volume and software inertia in their favor. They were also successful because of Amdahl's law which will be described later.

The purpose of this chapter is to review examples of these biologically inspired chips in each of the main categories. The rest of this section contains a more detailed discussion of the motivation for these chips, the algorithms they were emulating, and architecture issues. Then each of the general categories presented are discussed in more detail. This chapter will conclude with a look at the future of biologically inspired hardware, especially as we approach the realm of nano- and molecular scale technology.

2 Early Work in Biologically-Inspired Hardware

The hardware discussed in this chapter is based on neural structures similar to those presented above, and, as such, it is designed to solve a particular class of problems that are sometimes referred to as "intelligent computing." These problems generally involve the transformation of data across the boundary between the real world and the digital world, in essence from sensor readings to symbolic representations usable by a computer, this boundary has been called "the digital seashore[1]". Such transformations are found wherever a computer is sampling and/or acting on real world data. Examples include the computer recognition of human speech, computer vision, textual and image content recognition, robot control, Optical Character Recognition (OCR), Automatic Target Recognition, etc. These are difficult problems to solve on a computer, since they require the computer to find *complex structures and relationships* in massive quantities of low precision, ambiguous, and noisy data. These problems are also very important. Our inability to adequately solve them constitutes a significant barrier to computer usage. And we are out of ideas, neither AI, ANNs (Artificial Neural Networks), Fuzzy Logic, nor Bayesian networks[2] have yet enabled robust solutions.

At the risk of oversimplifying a complex family of problems, we will, somewhat arbitrarily, partition the solution to these problems into two domains, the "front end" and the "back end," as can be seen in Figure 4. Front end operations involve more direct access to a signal. They include filtering and feature extraction. Back end operations are more "intelligent." A back end operations include storing abstract views of objects or inter-word relationships. As we move from front end to back end, the computation becomes more interconnect driven, leveraging ever larger amounts of diffuse data at the synapses for the connections. We've learned much about the front end, where the data are input to the system. Here there are developments in traditional as well as neural implementations. This work has led to a useful set of tools and techniques, but they do not solve the whole problem. Consequently, more groups are beginning to look at the back-end – the realm of cerebral cortex – as a source of inspiration for solving the rest of the problem. And as difficult as the front end problems are, the back end problems are even more so. One manifestation of this difficulty is the "perception gap" discussed by Lazzaro and Wawrzynek [15], where the feature representations produced by more biologically inspired front end processing are incompatible with existing back end algorithms.

A number of researchers are beginning to refer to this "backend" as Intelligent Signal Processing (ISP), which augments and enhances existing Digital Signal Processing (DSP) by incorporating contextual and higher level knowledge of the application domain into the data transformation process. Simon Haykin (McMaster University) and Bart Kosko (USC) were editors of a special issue of the *Proceedings of the IEEE* [16] on ISP. In their introduction they state,

> *"ISP uses learning and other 'smart' techniques to extract as much information as possible from signal and noise data."*

[1] Hiroshi Ishii, MIT Media Lab

[2] "A Bayesian network (or a belief network) is a probabilistic graphical model that represents a set of variables and their probabilistic independencies," Wikipedia, http://www.wikipedia.org/.

If you are classifying at Bayes optimal rates and you are still not solving the problem, what do you do next? The solution is to add more knowledge of the process being classified to your classification procedures, which is the goal of ISP. One way to do this is to increase the contextual information (e.g., higher order relationships such as sentence structure and word meaning in a text based application) available to the algorithm. It is these complex, "higher-order" relationships that are so difficult for us to communicate to existing computers and, subsequently, for them to utilize efficiently when processing signal data.

Humans make extensive use of contextual information. We are not particularly good classifiers of raw data where little or no context is provided, but we are masters of leveraging even the smallest amount of context to significantly improve our pattern recognition capabilities.

One of the most common contextual analysis techniques in use today is the Hidden Markov Model (HMM) [17]. An HMM is a discrete Markov model with a finite number of states, that can be likened to a probabilistic finite state machine. Transitions from one state to the next are determined probabilistically. Like a finite state machine a symbol is emitted during each state transition. In an HMM the selection of the symbol emission during each state transition is also probabilistic. If the HMM is being used to model a word, the symbols could be phonemes in a speech system. Since the symbols are not necessarily unique to a particular state, it is difficult to determine the state the HMM is in by just observing the emitted symbols, hence the term "hidden." These probabilities can be determined from data of the real world process the HMM is being used to model. One variation is the work of Morgan and Bourlard [18] who used a large back-propagation network to provide HMM emission probabilities.

In many speech recognition systems, HMMs are used to find simple contextual structure in candidate phoneme streams. Most HMM implementations generate parallel hypotheses and then use a dynamic programming algorithm (such as the Viterbi algorithm) to find a "most likely" match that is the most likely utterance (or most likely path through the model) based on the phonemes captured by preprocessing and capturing features from the speech stream, and the probabilities used in constructing the HMM. HMMs have several limitations, first, they grow very large if the probability space is complex, such as when pairs of symbols are modeled rather than single symbols; yet most human language has very high order structure. Second, the "Markov horizon" which is a fundamental definition of Markov models and makes them tractable analytically, also contributes to the inability to capture higher order knowledge. Many now believe that we have passed the point of diminishing returns for HMMs in speech recognition.

The key point here is that most neuro-inspired silicon, in particular the analog based components, is primarily focused on the front end "DSP" part of the problem, since robust, generic back-end algorithms, and subsequently, hardware, have eluded us. Some have argued that if we did the front end right, the back-end would be easier, and whereas we can always do better in our front-end processing, the room for improvement is smaller there. Without robust back-end capabilities, general solutions will be more limited.

2.1 Flexibility Trade-Offs and Amdhal's Law

In the 80s and early 90s when most of the hardware surveyed in this chapter was created, there was a perception that the algorithms were large and complex and therefore slow to emulate on the computers of that time. Consequently specialized hardware to accelerate these algorithms was required for successful applications. What was not fully appreciated by many was that the performance of general purpose hardware was increasing faster than Moore's law, and that the existing neural algorithms did not scale well to the large sizes that would have fully benefited from special purpose hardware. The other problem was Amdahl's law.

As discussed earlier, these models have extensive concurrency which naturally leads to massively parallel implementations. The basic computation in these models is the multiply-accumulate operation that forms the core of almost all digital signal processing, and which can be performed with minimal, fixed point, precision. Also in the early 90s when much of the work on neural inspired silicon was done, microprocessor technology was actually not fast enough for many applications.

But neural network silicon is highly specialized and there are specific risks involved in its development. One way to conceptualize the trade-offs involved in designing custom hardware is shown in Figure 5. Although cost-performance[3] can be measured, flexibility cannot as easily, so the graph in the figure is more conceptual than quantitative. The general idea is that the more a designer hard-wires an algorithm into silicon, the better the cost-performance but the less flexible the device is.

The line, which is moving slowly to the right according to Moore's law, shows these basic trade-offs, and is, incidentally, not likely to be linear in the real world,. Another assumption is that the algorithms being emulated can be implemented at many different points on that scale, which is not always true either.

An important assumption in this analysis is that most applications require a wide range of computations, although there are exceptions, in most real world systems the "recognition" component is just a part of a larger system. So when considering specialized silicon for such a system, the trade-off shown in Figure 5, must be factored into the analysis. More general purpose chips tend to be useful on a larger part of the problem, but sacrifice cost-performance. Whereas the more specialized chips tend to be useful on a smaller part of the problem, but at a much higher cost-performance. Related to this trade-off then is Amdahl's law [19], which has always been a fundamental limitation to fully leveraging parallel computing,

> *Amdahl's Law: the speed-up due to parallelizng a computation is proportional to that portion of the computation that cannot be parallelized.*

Say, for example, you have a speech recognition application, and 20% of the problem can be cast into a simplified parallel form. If you have a special purpose chip that speeds up that 20% by 1000x, the total system performance increase will be about 25%

$$1/(0.8+(0.2/1000))=1.25$$

Of course, depending on the cost of the 1000x chip, the 25% may still be worth it. But what if you have a comparably priced chip that is slower but is more flexible and can parallelize 80% of the application, though with a more moderate speed increase, say 20x, the total system performance will be over 400%

$$1/(0.2+(0.8/20))=4.17$$

Almost all computationally intensive pattern recognition problems have portions of sequential computation, even if it is just moving data into and out of the system, data reformatting, feature extraction, post-recognition tasks, or computing a final result. Amdahl's law shows us that these sequential components have a significant impact on total system performance. As a result, the biggest problem that many early neural network chips had was that they tended to speed up a small portion of a large problem by moving to the right in Figure 5. For many commercial applications, after all was said and done, the cost of a specialized neural chip did not always justify the resulting modest total system performance increase.

In the mid-90s desktop chips were doubling performance every 18-24 months. And then in the mid-90s Intel and AMD added on-chip SIMD coprocessing in the form of MMX, which, for the Intel chips, has eventually evolved to SSE3 [20]. These developments, for the most part, spelled the death of most commercial neural net chips. However, in spite of limited commercial success most neural network chips were very interesting implementations, often with elegant engineering. The rest of this chapter will look briefly at a representative sample of some of these chips.

The reader should not conclude from this discussion that specialized chips are never economically viable, the continued success of graphics processors and DSPs are examples of specialized high volume chips and some neural networks chips[4] have found very successful niches, but it does illustrate some of the problems involved in architecting a successful niche chip. If one looks at, for example, commercial DSP chips for signal processing and related applications, they provide unique cost-performance, efficient power utilization, and just the right amount of

[3] In this chapter cost-performance is measured by (operations/second) / Cost. The cost of producing a silicon chip is directly related (in a complex, non-linear manner) to the **area** of the chip, larger chips are generally more expensive, which is our use here. Though more recently other factors such as **power consumption** are becoming equally, if not more important.

[4] One example is General Vision, http://www.general-vision.com/.

specialization in their niche to hold their own in volume applications against general purpose processors. In addition, they have enough volume and history to justify a significant software infrastructure.

In light of what we know now about Amdahl's law and about ISP, we can now survey the history and state of the art of neuro-inspired silicon.

2.2 Analog VLSI

There is no question that the most elegant implementation technique developed for neural emulation is the sub-threshold CMOS technology pioneered by Carver Mead and his students at CalTech [21]. Most MOSFETs (MOS Field Effect Transistors) used in digital logic are operated in two modes, off (0) and on (1). For the *off* state the gate voltage is more or less zero and the channel is completely closed. For the *on* state the gate voltage is significantly above the transistor threshold and the channel is saturated. A saturated *on* state works fine for digital and is generally desired to maximize current drive. However, the limited gain in that regime restricts the effectiveness of the device in analog computation. This is due to the fact that the more gain the device has, the easier it is to leverage this gain to create circuits that do useful computation and which also are insensitive to temperature and device variability.

However, if the gate voltage is positive (for the nMOS gate) but below the point where the channel saturates, the FET is still on, though with much lower current. In this mode, sometimes referred to as "weak inversion," there is useful gain and the small currents significantly lower the power requirements, though FETs operating in this mode tend to be slower. Carver Mead's great insight was that when modeling biologically inspired circuits, significant computation could be done using simple FETs operating in the sub-threshold regime, where, like real neurons, performance resulted from parallelism and not the speed of the switching devices. And as Carver and his students have shown, these circuits do a very good job approximating a number of neuroscience functions.

By using analog voltage and currents to represent signals, the considerable expense of converting signal data into digital, computing the various functions in digital and then converting the signal data back to analog, was eliminated. Neurons operate slowly and are not particularly precise, yet combined appropriately they perform complex and remarkably precise computations. The goal of the aVLSI research community has been to create elegant VLSI sub-threshold circuits that approximate biological computation.

One of the first chips developed by Carver and his students was the silicon retina [22] which was an image sensor that performed localized adaptive gain control and temporal / spatial edge detection using simple "local neighborhood" functional extensions to the basic photosensitive cell. There subsequently followed a silicon cochlea and numerous other simulations of biological circuits.

Two examples of these circuits are shown in Figure 6. A transconductance amplifier (voltage input, current output) and an "integrate and fire" neuron are two of the most basic building blocks for this technology. The current state of aVLSI research is very well described by Rodney Douglas [23], of the Neuroinformatics Institute, ETH – Zürich:

> *Fifteen years of Neuromorphic Engineering: progress, problems, and prospects*. *Neuromorphic engineers currently design and fabricate artificial neural systems: from adaptive single chip sensors, through reflexive sensori-motor systems, to behaving mobile robots. Typically, knowledge of biological architecture and principles of operation are used to construct a physical emulation of the target neuronal system in an electronic medium such as CMOS analog very large scale integrated (aVLSI) technology.*
>
> *Initial successes of neuromorphic engineering have included smart sensors for vision and audition; circuits for non-linear adaptive control; non-volatile analog memory; circuits that provide rapid solutions of constraint-satisfaction problems such as coherent motion and stereo-correspondence; and methods for asynchronous event-based communication between analog computational nodes distributed across multiple chips.*
>
> *These working chips and systems have provided insights into the general principles by which large arrays of imprecise processing elements could cooperate to provide robust real-time computation of sophisticated problems. However, progress is retarded by the small size of the development community, a lack of appropriate high-level configuration languages, and a lack of practical concepts of neuronal computation.*

Although still a modest sized community, research continues in this area with the largest group being the one at ETH. However, commercialization of this technology has been limited. The most notable success to date being Synaptics, Inc., which made a number of products that used the basic aVLSI technology, the most successful being the first laptop touch pads.

2.3 Intel's Analog Neural Network Chip And Digital Neural Network Chip

During the "heyday" of neural network silicon, 1986-1996, a major semiconductor vendor, Intel, produced two neural network chips. The first, the ETANN [24] (Intel part number 80170NX), was completely analog, but it was designed as a general purpose chip for non-linear feed-forward ANN operation. There were two grids of analog "inner product" networks, each with 80 inputs and 64 outputs, and a total of 10K (5K for each grid) weights. The chip computed the two inner products simultaneously, taking about 5 microseconds for the entire operation. This resulted in a total performance (feed-forward only, no learning) of over 2 billion connections computed per second, where a connection is a single multiply-accumulate of an input-weight pair. All inputs and outputs were in analog. The weights were analog voltages stored on floating gates – with the chip being developed and manufactured by the flash memory group at Intel. Complementary signals for each input provided positive and negative inputs. An analog multiplier was used to multiply each input by a weight, current summation of multiplier outputs provided the accumulation, with the output being sent through a non-linear amplifier (giving roughly a sigmoid function) to the output pins.

Although not designed specifically to do real-time learning, it was possible to do "chip in the loop" learning where incremental modification of the weights was performed in an approximate stochastic fashion. Learning could also be done off-line and then the weights downloaded to the chip.

The ETANN chip had very impressive computational density. However, the awkward learning and total analog design made it somewhat difficult to use. The multipliers were non-linear which made the computation sensitive to temperature and voltage fluctuations. Ultimately Intel retired the chip and moved to a significantly more powerful and robust all digital chip, the Ni1000.

The Ni1000 [25, 26] implemented a family of algorithms based on radial basis function networks (RBF [26]). This family included a variation of a proprietary algorithm created by Nestor, Inc., a neural network algorithm and software company. Rather than doing incremental gradient descent learning, as one sees with the back-propagation algorithm, the Ni1000 used more of a template approach where each node represented a "basis" vector in the input space. The width of these regions, which was controlled by varying the node threshold, was reduced incrementally when errors were made, allowing the chip to start with crude over-generalizations of an input to output space mapping, and then fine tune the mapping to capture more complex variations as more data are input. An input vector would then be compared to all the basis vectors with the closest basis vector being the winner. The chip performed a number of concurrent basis computations simultaneously, and then, also concurrently, determined the classification of the winning output, both functions were performed by specialized hardware.

The Ni1000 was a two layer architecture. All arithmetic was digital and the network parameters/weights were stored in Flash EEPROM. The first or hidden layer had 256 inputs of 16 bits each with 16 bit weights. The hidden layer had 1024 nodes and the second or output layer 64 nodes (classes). The hidden layer precision was 5-8 bits for input and weight precision. The output layer used a special 16-bit floating point format. One usage model was that of Bayesian classification, where the hidden layer learns an estimate of a probability density function (PDF) and the output layer classifies certain regions of that PDF into up to 64 different classes. At 40 MHz the chip was capable of over 10 billion connection computations per second, evaluating the entire network 40K times per second with roughly 4 watts peak power dissipation.

The Ni1000 used a powerful, compact, specialized architecture, unfortunately space limitations prevent a more detailed description, but the interested reader is encouraged to refer to the references. The Ni1000 was much easier to use than the ETANN and provided very fast dedicated functionality. However, going back to Figure 5, this chip was a specific family of algorithms wired into silicon. Having narrower functionality it was much more at risk from Amdahl's law, since it was speeding up an even smaller part of the total problem. Like CNAPS it too was ultimately run over by the silicon steam roller.

2.4 Cellular Neural Networks

Cellular neural networks (CNN) constitute another family of analog VLSI neural networks. It was proposed by Prof. Leon Chua in 1988 [27] who called it the "Cellular Neural Network." Though now it is called the "Cellular

Non-Linear Network." Like aVLSI, CNN has a dedicated following. The most well known is the Analogic and Neural Computing Laboratory of the Computer and Automation Research Institute of the Hungarian Academy of Sciences under the leadership of Tamas Roska. CNN based chips have been used to implement vision systems, and complex image processing similar to that of the retina has been investigated by a number of groups [28].

Although there are variations, the basic architecture is a 2-dimensional rectangular grid of processing nodes. And though the model allows arbitrary inter-node connectivity, most CNN implementations have only nearest neighbor connections. Each cell computes its state based on the values of its four immediate neighbors, where the neighbor's voltage and the derivative of this voltage are each multiplied by constants and summed. Each node then takes its new value and the process continues for another clock. This computation is generally specified as a kind of filter and is done entirely in the analog domain. But the algorithm steps are programmable. Consequently one of the real strengths of CNN is that the inter-node functions and data transfer is programmable, with the entire array appearing as a digitally programmed array of analog based processing elements. This is an example of an "SIMD" or Single Instruction, Multiple Data architecture, which consists of an array of computation units, where each unit performs the same operation, but each on its own data. CNN has been called "analogic." The downside is that programming can be complex and requires an intimate understanding of the basic analog circuits involved. Also, the limited inter-node connectivity restricts the chip to mostly "front end" kinds of processing, primarily of images. A schematic of the basic CNN cell is shown in Figure 7.

And whereas research and development continue, the technology has had only limited commercial success. As with aVLSI, it is a fascinating and technically challenging, but in real applications, it tends to be used for front-end problems and consequently is subject to Amdahl's law.

2.5 Other Analog/Mixed Signal Work

It is difficult to do justice to the large and rich area of biologically inspired analog design that has developed over the years. Other work includes that of Alan Murray [29], the former neural networks group at AT&T Bell Labs [30], Ralph Ettienne-Cummings [31], Jose Principe [32] and many more that cannot be mentioned due to limited space. And now some workers, such as Kwabena Boahan, are starting to move the processing further into the back end [33] by look at cortical structures for early vision.

Going back to Figure 4, we can see that the first couple of boxes of processing require the kind of massively parallel, locally connected feature extraction that CNN, aVLSI and other analog techniques provide. And for sensors, they can perform enhanced signal processing, and demonstrate better signal to noise ratios than more traditional implementations, providing such capabilities in compact, low power implementations.

Though more work is needed, there is concern that limited connectivity and computational flexibility makes it difficult to apply these technologies to the back-end. Although not a universally held opinion, I feel that these higher level association areas require a different approach to implementation. This general idea will be presented in more detail below, but first, we need to look at another important family of neural network chips, the massively parallel digital processors.

2.6 Digital SIMD Parallel Processing

Concurrent with the development of analog neural chips, there was parallel effort devoted to architecting and building digital neural chips. Although they could have dealt with a larger subset of pattern recognition solutions, like the analog chips they were mostly focused on neural network solutions to simple classification. A common design style that was well matched to the basic ANN algorithms was that of SIMD processor arrays. One chip that embodied that architecture was CNAPS developed by Adaptive Solutions [34, 35].

The world of digital silicon has always flirted with specialized processors. In the early days of microprocessors, silicon limitations restricted the available functionality. As a result many specialized computations were provided by coprocessor chips. Early examples of this were specialized floating point chips, as well as graphics and signal processing. Following Moore's law, the chip vendors found that they could add more and more function and started pulling some of these capabilities into the processor.

Interestingly, graphics and signal processing have managed to maintain some independence and remain as external coprocessors in many systems. Some of the reasons for this were the significant complexity in the tasks performed, the software inertia that had built up around these functions, and the potential for very low power dissipation which is required for embedded signal processing applications, such as cell phones, PDAs, and MP3 players.

In the early 90s, it was clear that there was an opportunity to provide significant speed up of basic neural network algorithms because of their natural parallelism. This was particularly true in situations involving complex, incremental, gradient descent adaptation, as one sees in many learning models. As a result, a number of digital chips were produced that aimed squarely at supporting both learning and non-learning network emulation.

It was clear from Moore's law that performance improvements and enhanced functionality continues for mainline microprocessors. This relentless march of the desktop processors has been referred to as the "silicon steam roller." And as the 90s went on, it became increasingly difficult for the developers of specialized silicon to stay ahead of the steam roller. At Adaptive Solutions, the goal was to avoid the steam roller by steering between having enough flexibility to solve most of the problem, to avoid Amdahl's law, and yet sufficiently specialized function to allow enough performance to make the chip cost-effective – essentially sitting somewhere in the middle of the line in Figure 5. This balancing act became increasingly difficult until eventually the chip did not offer enough cost-performance improvement in its target applications to justify the expense of a specialized coprocessor board.

The CNAPS architecture consisted of a 1 dimensional PN (Processor Node) array in an SIMD parallel architecture [36]. To allow as much performance-price as possible, modest fixed point precision was used to keep the PNs simple. With small PNs the chip could leverage a specialized redundancy technique developed by Adaptive's silicon partner, Inova Corporation. During chip test, each PN could be added to the 1-D processor chain, or bypassed. In addition, each PN had a large power transistor (with a width of 20,000λ) connecting the PN to ground. Laser fuses on the 1D interconnect and the power transistor was used to disconnect and power down of defective PNs. Testing of the individual PNs was done at wafer sort, then an additional lasing stage, before packaging and assembly, would configure the dice, fusing in the good PNs and fusing out and powering down the bad PNs. The first CNAPS chip had an array of 8x10 (80) PNs fabricated, of which only 64 need to work to be a fully functional die. Figure 8 shows the system architecture and internal PN architecture.

Simulation and analysis was used to determine roughly what the optimal PN size (the "unit of redundancy") and the optimal number of PNs were. The die ended up being almost exactly an inch on a side with 14 million transistors fabricated, this led to 12 die per 6 inch wafer, which until recently made it the physically largest processor chip ever made, a die photo can be seen in Figure 9.

The large version of the chip, called the CNAPS-1064, had 64 operational PNs and operated at 25 MHz with 6 watts worst case power consumption. Each PN was a complete 16-bit DSP with its own memory. Neural network algorithms tend to be vector / matrix based and map fairly cleanly to a 1 dimensional grid, so it was easy to have all PNs doing useful work at once. The maximum compute rate then was 1.2 billion multiply-accumulates per second per chip, which was about 1000x faster than the fastest workstation at that time. Part of this speed up was due to the fact that each PN did several things in one clock to realize a single clock multiply-accumulate: input data to the PN, perform a multiply, perform an accumulate, perform a memory fetch, and compute the next memory address. In the late 80s DSPs (Digital Signal Processor chips) were able to do such a single multiply-accumulate in one clock, but it was not until the Pentium Pro that desktop microprocessors reached a point where they performed most of these operations simultaneously.

When developing the CNAPS architecture, there were a number of key decisions that were made, these included the use of limited precision and local memories, architecture support for C, and I/O bandwidth.

At a time when the computing community was moving to floating point computation, and the microprocessor vendors were pulling floating processing onto the processor chips and optimizing its performance. However, CNAPS used limited precision, fixed point arithmetic. The primary reason for this decision was that yield calculations indicated that a floating point PN was too large to take advantage of PN redundancy. This redundancy bought roughly a 2x cost-performance improvement. Since a floating point PN would have been 2-3x larger than a fixed point PN, the use of modest precision fixed point arithmetic meant an almost 6x difference in cost-performance. Likewise, simulation showed that most of the intended algorithms could get by with limited precision fixed point arithmetic. It turned out that this was, in general, a good decision, since there were rarely problems with the limited precision, the major disadvantage being that it made programming more difficult, though DSP programmers have been effectively using fixed point precision for many years.

The second decision was to use local, per PN, memory (4KB of SRAM per PN). Although this significantly constrained the set of applications that could leverage the chip, it was absolutely necessary in achieving the performance goals. And the reality is that it is unlikely that the CNAPS chip would have ever been built had performance been reduced enough to allow the use of off-chip memory. As with DSP applications, almost all memory access was in the form of long arrays that can benefit from some pre-fetching, but not much from caching.

The last two decisions, architecture and I/O bandwidth limitations, were driven by performance-price and design time limitations. One of the objectives of the architecture was that it be possible for two IC engineers to do the, circuits, logic schematics and layout (with some additional layout help) in one year. As a result, the architecture was very simple. This made the design simpler and the PNs smaller, but programming was more difficult. One result of this strategy is that the architecture did not support the C language efficiently. Though there were some fabrication delays, the architecture, board, and software rolled out simultaneously and worked very well. The first system shipped in December 1991 and quickly ramped to a modest volume. One of the biggest selling products was an accelerator card for Adobe Photoshop, which, in spite of Amdahl problems (poor I/O bandwidth), it offered unprecedented performance.

By 1996 desktop processors had increased their performance significantly, and Intel was on the verge of providing the MMX SIMD coprocessor instructions. Although this first version of an SIMD coprocessor was not complete and was not particularly easy to use, the performance of a desktop processor with MMX reduced the advantages of the CNAPS chipset even further in the eyes of their customers, and people stopped buying.

Everyone knew that the silicon steam roller coming, but it was moving much faster (and maybe accelerating as some have suggested) than expected. In addition, Intel quickly enhanced the MMx coprocessor to the now current SSE3, which is a complete and highly functional capability. DSPs were also vulnerable to the microprocessor steam roller, but managed, primarily software inertia and very low power dissipation, to hold their own.

Although there were other digital neural network processors, none of them reached any significant level of success either and basically for the same reasons. Space limits the discussion of all but a few of these others, but there are two, in particular, that deserve mention.

2.7 Other Digital Architectures

Another important digital neural network architecture was the Siemens SYNAPSE-1 processor developed by Ulrich Ramacher and his group at Siemens Research in Munich [37]. The chip was similar to CNAPS in terms of precision, fixed point arithmetic, and basic processor node architecture. Where the SYNAPSE-1 differed is that it used off-chip memory to store the weight matrices.

A SYNAPSE-1 chip contained a 2 by 4 array of MA16 "neural signal processors," each with a 16-bit multiplier and 48-bit accumulator. The chip frequency was 40 MHz, one chip could compute roughly 5 billion connections per second with feedforward (non-learning) execution.

Recall that in architecting CNAPS, one of important decisions was whether to use on-chip per PN memory, or off-chip shared memory for storing the primary weight matrices. For a number of reasons, including the targeted problems space and the availability of a state of the art SRAM process, Adaptive Solutions chose to use on chip memory for CNAPS. However, for performance reasons this decision limited the algorithm and application space to those whose parameters fit into the on-chip memories. Although optimized for matrix-vector operations, CNAPS was designed to efficiently perform a fairly wide range of computations.

The SYNAPSE-1 processor was much more of a matrix-matrix multiplication algorithm mapped into silicon. In particular, they were able to take advantage of a very clever insight, the fact that in any matrix multiplication, individual elements of the matrix are used multiple times. The SYNAPSE-1 broke all matrices into 4x4 chunks. While the elements of one matrix were broadcast to the array, 4x4 chunks of the other matrix would be read from external memory into the array. In a 4x4-matrix by 4x4-matrix multiplication, each element in the matrix is actually used 4 times. This allowed the processor chip to support 4 times as many processor units for a given memory bandwidth than a processor not using this optimization.

Returning to Figure 5, we can see that the SYNAPSE-1 architecture increased performance by specializing the architecture to matrix-matrix multiplications. Fortunately, most neural network computations can be cast in a matrix form, though it did restrict maximum machine performance to algorithms that did matrix-matrix multiplies. However, like the other digital neural network chips, the SYNAPSE-1 eventually lost out to high performance microprocessor and DSP hardware.

2.8 General Vision

A similar chip to the Ni1000 was the ZISC (Zero Instruction Set Computer) developed by Guy Paillet and others at IBM in Paris. The ZISC chip was digital and also was basically a "vector template" approach. It was simpler and cheaper than the Ni1000 but implemented roughly the same algorithms. The ZISC chip survives today as the primary product of General Vision, Petaluma, CA.

In addition to the CNAPS, SYNAPSE-1, ZISC, and Ni1000, there were other digital chips that were developed either specifically or in part to emulate neural networks. HNC developed the SNAP, a floating point SIMD standard cell based architecture [38]. One very nice architecture is SPERT [39] developed by groups at the University of California Berkeley and the International Computer Science Institute (ICSI) in Berkeley. The SPERT was designed to do efficient integer vector arithmetic and to be configured into large parallel arrays. A similar parallel processor array that was created from FPGAs (Field Programmable Gate Arrays) that was suited to neural network emulation was REMAP [40].

3 Current Directions in Neuro-Inspired Hardware

One limitation of traditional ANN algorithms is that they did not scale particularly well to very large configurations. As a result, commercial silicon was generally fast enough to emulate these models, reducing the need for specialized hardware. Consequently, with the exception of on-going work in aVLSI and CNN, general research in neural inspired hardware has languished. However, activity in this area is picking up again. There are two reasons for this. The first is that computational neuroscience is starting to yield scalable algorithms that can scale to large configurations and have the potential for solving large complex problems. And the second is the excitement of using molecular scale electronics, which makes possible comparably scalable hardware. As we will see, at least one of the projected nanoelectronic technologies is a complementary match to biologically inspired algorithms.

There are a number of challenges facing the semiconductor industry: power density, interconnect reverse scaling, device defects & variability, memory bandwidth limitations, performance overkill, density overkill, and increasing design complexity. Performance overkill is where the highest-volume segments of the market are no longer performance/clock frequency driven. Density overkill is where it is difficult for a design team to effectively design and verify all the transistors available to them on a single die. Although none of these is a potential show-stopper, taken together they do create some significant challenges.

Another challenge is the growing reliance on parallelism for performance improvements. In general purpose applications, the primary source of parallelism has been within a single instruction stream, where many instructions can be executed simultaneously, sometimes even out of order. However this instruction level parallelism (ILP) has its limits and becomes exponentially expensive to capture. Microprocessor manufacturers are now developing "multiple core" architectures whose goal is to execute multiple threads efficiently. As multiple core machines become more common place, software and application vendors struggle to create parallel variations of their software.

Due to very small, high resistance wires, many nano-scale circuits will be slow, and power density will be a problem because of high electric fields. Consequently, performance improvements at the nano-scale will also need to come almost exclusively from parallelism and to an even greater extent than traditional architectures.

Looking at these various challenges, which ones does nanotechnology address? It clearly addresses the end of Moore's law and maybe the memory bandwidth problem. However it also aggravates most of the other existing problems, in particular, signal/clock delay, device variability, manufacturing defects, and design complexity.

As we proceed down the path of creating nanoscale electronics, the biggest question though is, how exactly will we use it? Can we assume that computation, algorithms, and applications will continue more or less as they have? Should we? Will the nanoscale processor of the future consist of 100K x86 cores with a handful of application specific coprocessors? The effective use of nanotechnology will require solutions to more than just increased density, we need to consider total system solutions. We cannot create computing structures in the absence of some sense of how they will be used and what applications they will enable. Any paradigm shift in applications and architecture will have a profound impact on the entire design process and the tools required, as well as the requirements placed on the circuits and devices themselves.

As discussed earlier, algorithms inspired by neuroscience have a number of interesting and potentially useful properties, including, fine-grain and massively parallelism. They are constructed from slow, low-power, unreliable components, are tolerant of manufacturing defects, and are robust in the presence of faulty and failing hardware. They adapt rather than programmed. They are asynchronous, compute with low precision, and degrade gracefully in the presence of faults. And very importantly they are self-organizing, in fact, where system design becomes more the provisioning of organizing principles (Prof. Christoph von der Malsburg), rather than the specification of all operational aspects of the models, www.organic-computing.org. Such adaptive, self-organizing structures also promise some degree of design error tolerance. And very importantly, they solve problems dealing with the interaction of an organism/system with the real world. In short, they couldn't be more different than what we do now – and yet they are a remarkable match to nanoelectronics.

There are several very important points to be made about biologically inspired models. The first concerns the computational models and the applications they support. Biologically inspired computing uses a very different set of computational models than have traditionally been used. And subsequently they are aimed at a fairly specialized set of applications (many of which we solve, or not, to various degrees). Consequently, for the most part biological models are not a replacement for existing computation, but rather they are an enhancement to what we do now. Specialized hardware for implementing these models needs to be evaluated accordingly. In the next few sections we will explore some of these models at several different levels.

3.1 Moving to More Sophisticated Neuro-Inspired Hardware

As already mentioned it is the back end where we continue to struggle with algorithms and implementation, and it is the back end where potential strategic inflection points lie. The remainder of this chapter will be focused then on back-end algorithms and hardware.

The ultimate cognitive processor is cerebral cortex. It is remarkably uniform, not only across different parts of cortex, but across almost all mammalian species. Although we are a long ways from understanding the details of how it does what it does, some of the basic computations are beginning to take shape. Nature has, so it appears, produced a general purpose computational device that is a fundamental component of higher level intelligence in mammals.

Some generally accepted notions about cortex are that it represents knowledge in a sparse distributed, hierarchical manner, and that it performs a kind of Bayesian inference over this knowledge base, which it does with remarkable efficiency. This knowledge is added to the cortical data base by a complex process of adaptive learning.

One of the fundamental requirements of intelligent computing, is that we need to capture higher order relationships. The problem with Bayesian inference is that it is an exponentially increasing computation in the number of variables (it has been shown to be NP-Hard, which means that the number of computational steps increases exponentially with the size of the problem) – as order increases, computational overhead increases even more rapidly. Consequently, to use Bayesian inference in real problems, order is reduced to make them computationally tractable.

One common way to do this is to create a Bayesian network, which is a graph structure where the nodes represent variables and the arcs connecting the nodes represent dependencies. If there is reasonable independence between many of the variables then the network itself will be sparsely connected. Bayesian networks "factorize" the inference computation by taking advantage of the independence between different variables. Factorization does reduce the computational load, but at the cost of limiting the knowledge in the network. Also a custom network is required for each problem.

Cortical networks appear to use sparse distributed data representations, where each neuron participates in a number of specific data representations. Distributed representations also diffuse information, topologically localizing it to the areas where it is needed and reducing global connectivity. Computing with distributed representations can be thought of as the hardware equivalent of spread spectrum communication, where pseudo-random sequences of bits are used to spread a signal across time and frequency. In addition to spreading inter-node communication, distributed representations also spread the computation itself. One hypothesis of cortex operation is that distributed representations of information are a kind of extreme factorization, allowing efficient, massively parallel Bayesian inference.

Mountcastle [41, 42] did pioneering work in understanding the structural architecture of the cortex, including proposing the columnar organization. The fundamental unit of computation appears to be the cortical minicolumn, a vertically organized group of about 80-100 neurons which traverses the thickness of the gray matter (~3 mm) and is about 50μ in diameter. Neurons in a column tend to communicate vertically with other neurons on different layers in the same column. These are subsequently organized into larger columns variously called just "columns", "cortical columns", "hypercolumns", or "modules." Braitenberg [43] postulates two general connectivity systems in cortex: "metric" (high density connections to physically local cells, based on actual two-dimensional layout), and "ametric" (low density point to point connections to all large groups of densely connected cells). Connectivity is significantly denser in the metric system, but with limited extent.

One approach to creating cortical like algorithms is to model each column as an auto-associative network such as the Palm model discussed earlier. The columns are then sparsely connected to each other, but the specifics of the inter-column connections are still not certain and different workers have different ideas about those [44]. In addition, neocortex has a definite hierarchical organization where there are as many feedback paths as feed-forward paths.

Among other things, the massive scale is probably one of the more important advantages of biological computation. Consequently, it is likely that useful versions of these algorithms will require networks with a million or more nodes. Back end processing, because of the need to store large amounts of unique synaptic information, will probably have simpler processing than we see at the front end, but on a much larger scale.

Hecht-Nielsen [45] bases the inter-column (which he calls "regions") connections on conditional probabilities, which capture higher-order relationships. He also uses abstraction columns to represent groups of lower level columns. He has demonstrated networks that do a remarkable job of capturing aspects of English, these networks consist of seveal billion connections and require a large computer cluster to execute.

Granger [46, 47] leverages nested distributed representations in a way that adds the temporal dimension, creating hierachical networks that learn sequences of sequences. George and Hawkins [48] use model likelihood information ascending a hierarchy with model confidence information being fed back. Other researchers are also contributing to these ideas, Grossberg [49], Lansner [50, 51], Arbib [52], Roudi and Treves [53], Renart *et al.* [54], Levy *et al.* [55], and Anderson [56]. This is still a dynamic area of research, and at this point there is no clear "winning" approach.

Another key feature of some of these algorithms is that there is an oscillatory sliding threshold that causes the more "confident" columns to converge to their attractors more quickly, the less confident more slowly, and those of low confidence do not converge at all, taking a "NULL" state. This process is remarkably similar to the EM waves that flow through cortex when it is processing data.

Connectivity is still one of the most important problems as we begin to consider scaling to very large models. Axons are very fine and can be packed very densely in a 3D mesh. Interconnect in silicon generally operates in a two-dimensional plane, though with several levels, 9 or more with today's semiconductor technologies. And most importantly, silicon communication is not continuously amplifying as one sees in axonal and some dendritic processes. The following result [57-59] demonstrates one particular problem:

> Theorem: *Assume an unbounded or large rectangular array of silicon neurons where each neuron receives input from its N nearest neighbors – i.e., the fan-out (divergence) and fan-in (convergence) is N. Each such connection consists of a single metal line, and the number of two-dimensional metal layers is much less than N. Then the area required by the metal interconnect is* $O(N^3)$.□

So if we double the fan-in per node from 100 to 200, the silicon area required for the metal interconnect increases by a factor of 8x. This result means that even for modest local connectivity, that portion of silicon area devoted to metal interconnect will dominate. It has been shown that for some models even moderate multiplexing of interconnect would significantly decrease the silicon area requirements without any real loss in performance [60]. Carver Mead's group at Caltech, and others, developed the Address Event Representation (AER), a technique for multiplexing a number of pulse streams on a single metal wire [61, 62]. When analog computation is used, signals can be represented by the time between action-potential-like "spikes." These signal "packets" or "pulses" are transmitted asynchronously the moment they occur, with the originating unit's address, on a single shared bus. This "pseudo-digital" representation allows multiplexing of the bus and the retention of analog and temporal information without expensive conversions.

In studying potential implementations of cortical structures, we developed an efficient connection multiplexing architecture where data transfer occurs via overlapping, hierarchical buses [63]. This structure, *The Broadcast Hierarchy* (TBH) allows simultaneous high-bandwidth local connectivity and long-range connectivity, thereby providing a reasonable match to many biological connectivity patterns.

I next present a relevant proposed hybrid CMOS / nanoelectronic technology, CMOL.

3.2 CMOL

Likharev has proposed CMOL (Cmos / MOLecular hybrid) as a implementation strategy for charge-based[5] nano-electronic devices. Likharev's group has analyzed a number of examples of CMOL configurations, including memory, reconfigurable logic, and neuromorphic CrossNets.[64-66]. In addition, nanogrids are most likely to be the first commercial deployment of nanoelectronic circuits [67].

CMOL consists of a set of nanogrids fabricated on top of traditional CMOS, with the CMOS being used, among other things, for signal restoration and current drive, nanogrid addressing, and to communicate signals into and out of the nanogrids. The nanogrids themselves will generally have more specialized computation such as a memory, which augments the computation being performed by the CMOS.

A nanogrid consists of a set of parallel wires, with another set of parallel wires fabricated on top of and orthogonal to the first set. Likharev shows that such grids need not be laid out in perfect dimensions or alignment, and can be reproduced using nanoimprint templates. Sandwiched between the two grids is a planar material made of a specific molecular structure. Where the horizontal and vertical wires cross a molecular switch is created. Several mechanisms have been identified to effect the desirable electrical properties where two nano-wires cross.

The most important property is of a binary "latching switch" with two metastable internal states [68]. This nanoscale device can be programmed to either an "on" or an "off" state using two sets of voltages. The lower set is used to read out the device to determine its state. The higher set is used to actually change the state of the device. The programming voltages are used to switch the device between high and low resistance states. The lower read-out voltages are used to determine the resistance or "state" of the molecule. Another required characteristic of these devices is rectification, where current is only allowed in one direction.

One of the most important characteristics of CMOL is the unique way in which the grids are laid at an angle with respect to the CMOS grid. Each nanowire is connected to the upper metal layers of the CMOS circuitry through a pin. In order for the CMOS to address each nanowire in the denser nanowire crossbar arrays, when it is fabricated, the nanowire crossbar is turned by an angle α, where α is the tangent of the ratio of the CMOS inter-pin distance to the nanogrid inter-pin distance. This technique allows the grid to be arbitrarily aligned with the CMOS and still have most nanowires addressable by the appropriate selection of CMOS cells. A nano-wire is contacted by two CMOS cells, both of which are required to input a signal or read a signal. This basic connectivity structure is shown in Figure 10.

CMOL is not necessarily biologically inspired, but it is a promising technology for implementing such algorithms as we will see in the next session. CMOL uses charge-accumulation as its basic computational paradigm. This paradigm is also used by neural structures. Other nano-scale devices such as spin technologies do not implement a charge accumulation model, so such structures would have to emulate a charge-accumulation model, probably in digital.

3.3 An Example: CMOL Nano-Cortex

We have performed a high-level analysis of the implementation of a cortical column in CMOL. We assume that column operation is based on the Palm model discussed above. For this analysis we will ignore multiple column communication. Also, for simplicity's sake, we will assume a non-learning model where the weights are computed off line and downloaded into the nanogrid. Table 1 shows some typical values for the parameters used in our cortical column architectural analysis. These values represent typical numbers used by several different simulation models, in particular, Lansner and his group at KTH [69]. There is related work at IBM [70], where a mouse cortex sized model has been simulated on 32K processor IBM BlueGene/L.

We have analyzed four basic designs as shown in Figure 11: (a) all digital CMOS, (b) mixed-signal CMOS, (c) all digital hybrid CMOS/CMOL, and (d) mixed-signal hybrid CMOS/CMOL. For the CMOS designs and the CMOS portion of CMOL, we assumed a 22 nm process as "maximally" scaled CMOS. To approximate the features for this process, we did a simple linear scaling of a 250 nm process. Table 2 shows the results of this analysis, where the

[5] Researchers are investigating a number of molecular technologies based on computational paradigms other than charge, such as spintronics, quantum cellular automata, and DNA computing. However, since neural circuits operate on the principle of charge accumulation, charge based computation seems a better match, though more study of these other technologies is required.

cost-performance for those four systems with the assumption of an 858 mm^2 die size (the maximum lithographic field size expected for a 22 nm process) are presented.

Referring to Table 2, with the mixed signal CMOL, we are able to implement roughly 10M columns, each having 1K nodes, with 1K connections each, for a total of 10 Tera-connections. And we can update this entire network once every millisecond. These are approaching biological densities and speeds, though, of course, with less functionality. This is technology that could be built into a portable platform, with the biggest constraint being the high power requirements. We are now investigating spike based models [71] that should allow us to significantly lower the duty cycle and the power consumed.

Although we did not include real-time learning/adaptation in the circuits analyzed here, deployed systems will need to be capable of real-time adaptation. We expect that additional learning circuitry will reduce density by about 2-3x. Also, we have not addressed the issue of fault tolerance. We have found that the Palm model can tolerate errors, single 1 bits set to 0, in the weight matrix of up to 10%. For this reason, and given the excellent results from Likharev and Strukov [66] on the fault tolerance of CMOL arrays used as memory, we do expect that some additional hardware will be needed to complement algorithmic fault tolerance, but we do not expect it to reduce the density in any significant way.

4 Summary and Conclusions

In this chapter I have provided a brief and somewhat superficial survey of the specialized hardware developed over the last 20 years to support neurobiological models of computation. I then looked briefly at current efforts and speculated on how such hardware, especially when implemented in nanoscale electronics, could offer unprecedented compute density, possibly leading to new capabilities in computational intelligence. Also, biologically inspired models seem to be a better match to nano-scale circuits.

The mix of continued Moore's law scaling, models from computational neuroscience, and molecular scale technology portends a potential paradigm shift in how computing is done. Among other things the future of computing is probably not about discrete logic, but about the encoding, learning, and performing inference over stochastic variables. There could be a wide range of applications for devices like this in robotics, the reduction and compression of widely distributed sensor data, power management, etc.

One of the leading lights of the first computer revolution saw this clearly. At the IEEE Centenary in 1984 ("The Next 100 Years," IEEE Technical Convocation), Dr. Robert Noyce, co-founder of Intel and co-inventor of the Integrated Circuit, said:

> *Until now we have been going the other way; that is, in order to understand the brain we have used the computer as a model for it. Perhaps it is time to reverse this reasoning: to understand where we should go with the computer, we should look to the brain for some clues.*

5 Tables

Table 1 - Typical Values Of Parameters Used For A Cortical Column Analysis

Parameters	Range	Typical Value I	Typical Value II
Hypercolumn node size	128 ~ 128 K	1 K	16 K
Weight matrix size (single-weight-bit)	2^{14} ~ 2^{34} bits	2^{20} bits	2^{28} bits
Weight matrix size (multi-weight-bit)	2^{18} ~ 2^{51} bits	2^{30} bits	2^{42} bits
Multi-weight-bits	7 ~ 17 bits	10 bits	14 bits
# Active nodes in hypercolumn	7 ~ 17	16	16
Inner-product result bits (single-weight-bit)	3 ~ 5 bits	5 bits	5 bits
Inner-product result bits (multi-weight-bit)	11 ~ 21 bits	14 bits	18 bits

Table 2 – Analysis Results

Design		**# Column Processors**	**Power (W)**	**Update Rate (G nodes/sec)**	**Memory %**
CMOS All Digital	**1-bit eDRAM**	**6,600**	**528**	**3,072**	**2.9**
CMOS Mixed-Signal	**1-bit eDRAM**	**19,500**	**487**	**22,187**	**9.0**
CMOL All Digital	**1-bit CMOL Mm**	**4,042,752**	**317**	**4,492**	**40**
CMOL MS	**1-bit CoNets**	**10,093,568**	**165**	**11,216**	**100**

6 Figures

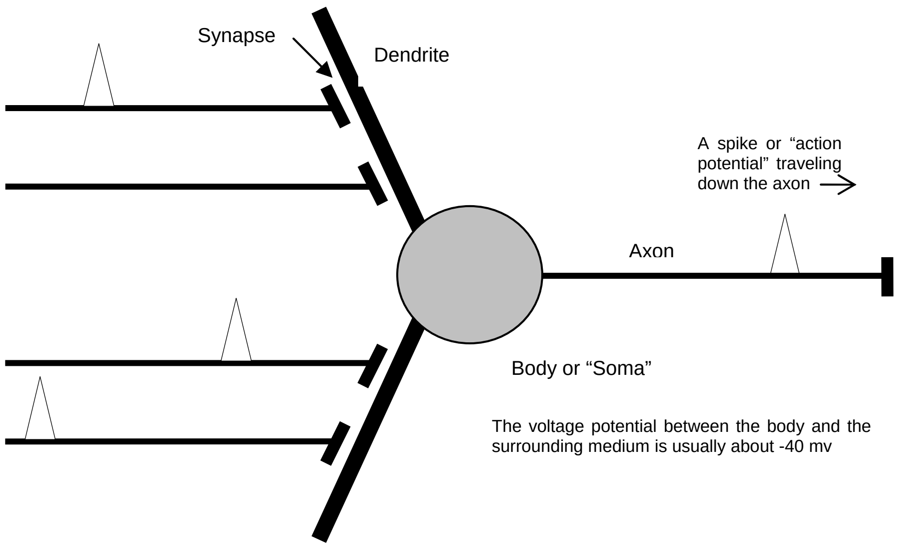

Synapse
Dendrite
A spike or "action potential" traveling down the axon
Axon
Body or "Soma"
The voltage potential between the body and the surrounding medium is usually about -40 mv

Figure 1 **-** *An Abstract Neuron*

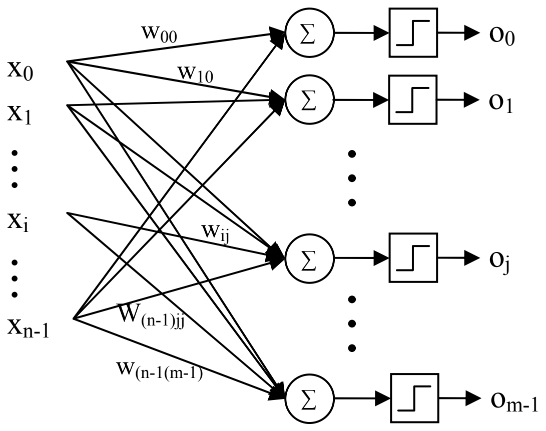


Figure 2- A Single Layer Perception

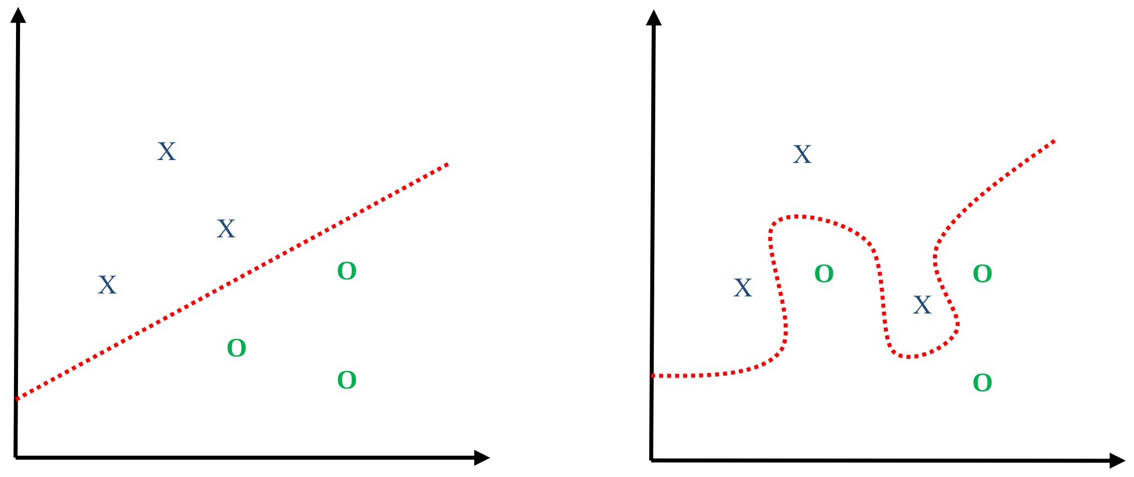


Figure 3 - Linear and Non-Linear Classification

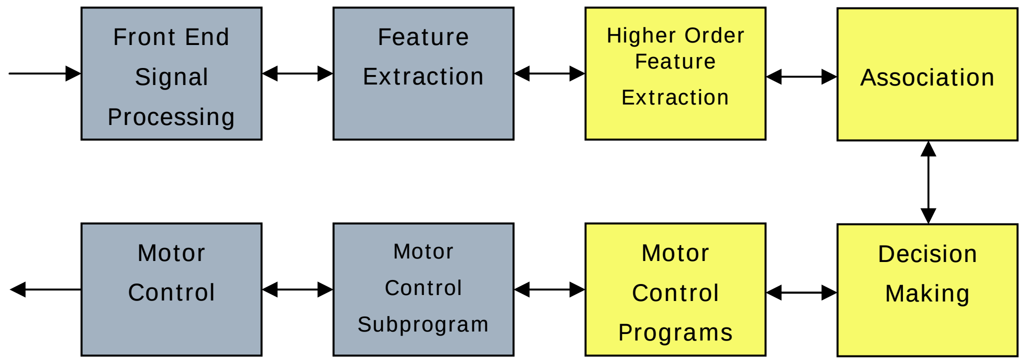


Figure 4 - A Canonical System

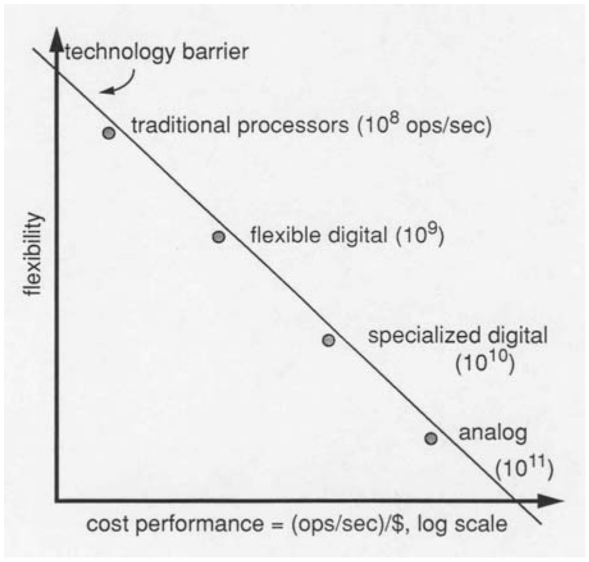
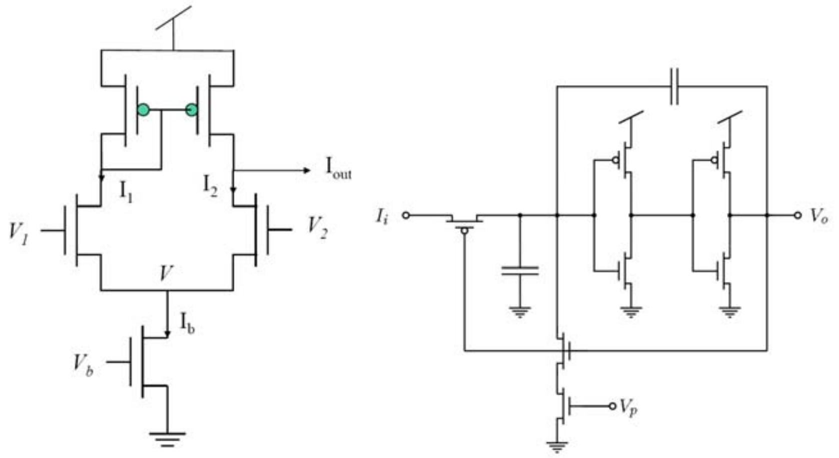


Figure 5 **-** *Flexibility* **-** *Cost/Performance Trade-off*



(a) (b)

Figure 6 **-** *Basic aVLSI Building Blocks:*

(a) A Transconductance Amplifier, (b) An Integrate and Fire Neuron

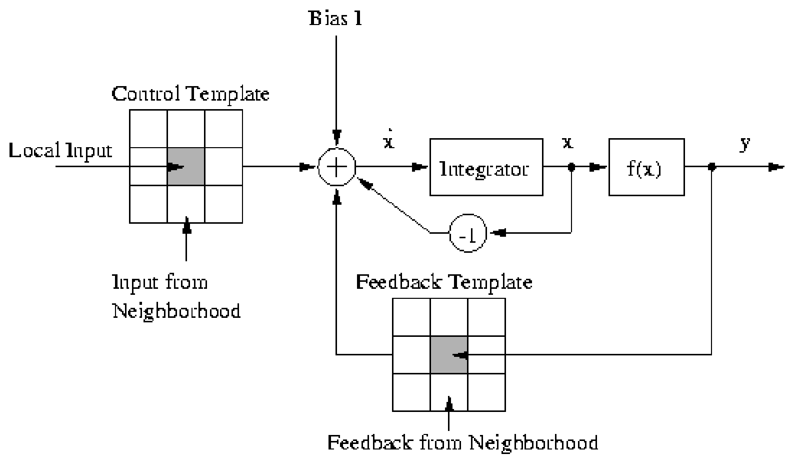
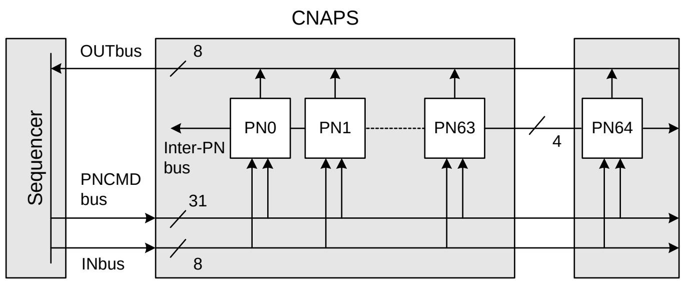
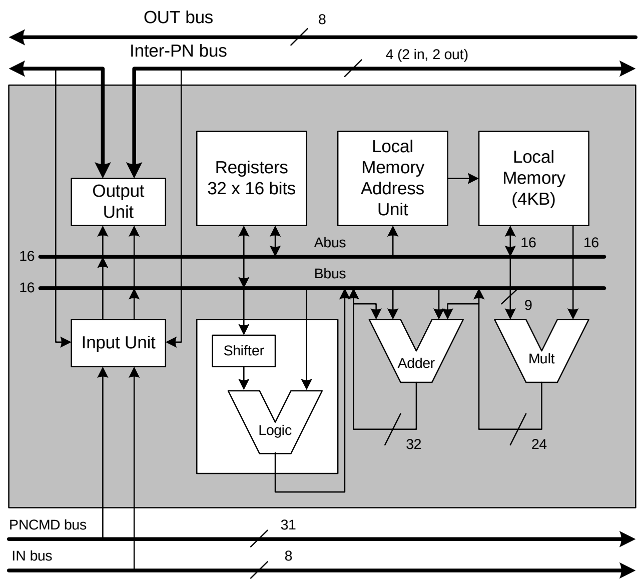


Figure 7 - Basic CNN Operation [72]



(a)



(b)

Figure 8 - CNAPS Architecture (a) System Architecture, (b) PN Architecture

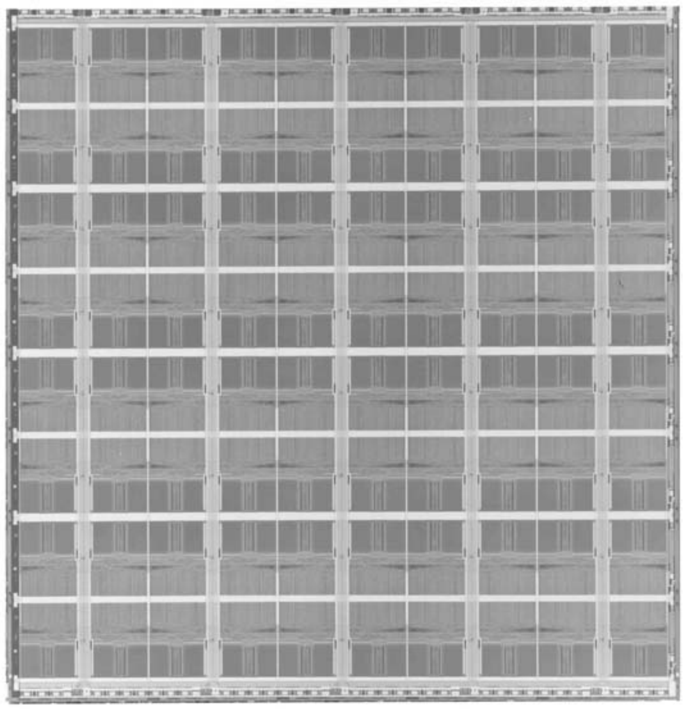


Figure 9 - CNAPS Die Photo

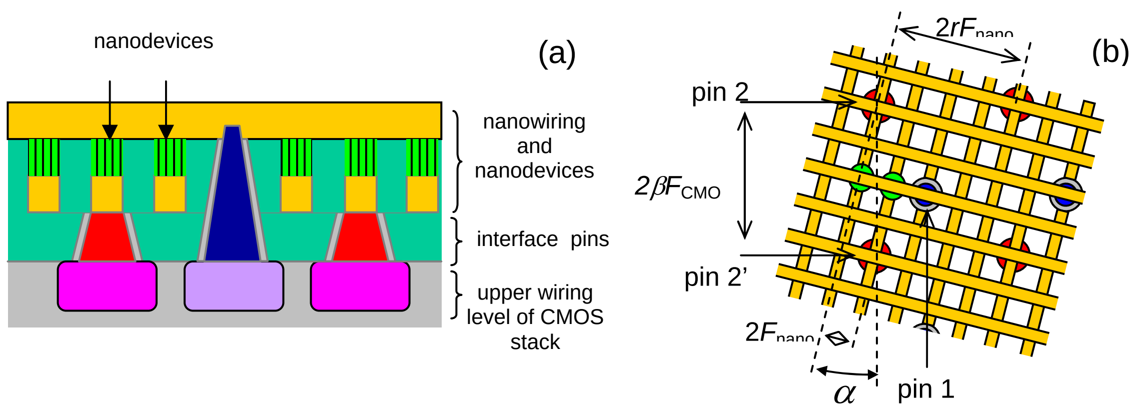


Figure 10 – CMOL [1]

(a) Schematic side view, and (b) top view showing that any nanodevice may be addressed via the appropriate pin pair (e.g, pins 1 and 2 for the leftmost of the two shown devices, and pins 1 and 2' for the rightmost device). Panel (b) shows only two devices; in reality, similar nanodevices are formed at all Nanowire crosspoints. Also not seen on panel (b) are CMOS cells and wiring.

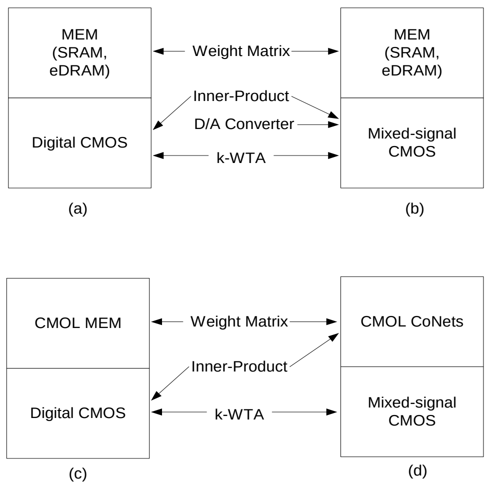


Figure 11 - Architecture Space - Biologically Inspired Models

7 References

1. Likharev, K., *CMOL: Freeing Advanced Lithography from the Alignment Accuracy Burden*, in *EIPBN'07*. 2007: Denver.
2. McCulloch, W.S. and W.H. Pitts, *A logical calculus of the ideas immanent in nervous activity.* Bulletin of Mathematical Biophysics, 1943. **5**: p. 115-133.
3. Steinbuch, K., *Die Lernmatrix.* Kybernetik, 1961. **1**.
4. Rosenblat, F., *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. 1962, New York: Spartan.
5. Minsky, L.M. and A.S. Papert, *Perceptrons: An introduction to computational geometry.* 1988.
6. SOAR. *Web Page*. http://sitemaker.umich.edu/soar [cited.
7. Hopfield, J., *Neural Networks and Physical Systems with Emergent Collective Computational Abilities.* Proceedings of National Academy of Science, 1982. **79**.
8. Rumelhart, D., G. Hinton, and R. Williams, *Learning internal representations by error propagation.* 1986.
9. Werbos, P.J., *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. 1994: Wiley-Interscience.
10. Palm, G., et al., *Neural Associative Memories*, in *Associative Processing and Processors*, A. Krikelis and C.C. Weems, Editors. 1997, IEEE Computer Society: Los Alamitos, CA. p. 284-306.
11. Sandberg, A., et al., *Bayesian attractor networks with incremental learning.* Network: Computation in neural systems, 2002. **13**(2): p. 179-194.
12. Zhu, S., *Associative Memory as a Primary Component in Cognition*, in *CSEE*. 2008, School of Science and Engineering, Oregon Health & Science University: Portland, OR.
13. Hammerstrom, D., *Neural Networks At Work*, in *IEEE Spectrum*. 1993. p. 26-32.
14. Hammerstrom, D., *Working with Neural Networks*, in *IEEE Spectrum*. 1993. p. 46-53.
15. Lazzaro, J. and J. Wawrzynek, *Speech Recognition Experiments with Silicon Auditory Models.* Analog Integrated Circuits and Signal Processing, 1997. **13**: p. 37-51.
16. Haykin, S. and B. Kosko, eds. *Intelligent Signal Processing*. 11 ed. Proceedings of the IEEE. Vol. 86. 1998, IEEE Press.
17. Rabiner, L., *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*. 1989, IEEE.
18. Bourlard, H. and N. Morgan, *Connectionist Speech Recognition - A Hybrid Approach*. 1994, Boston, MA: Kluwer Academic Publishers.
19. Hennessy, J.L. and D.A. Patterson, *Computer Architecture: A Quantitative Approach*. 1991, Palo Alto, CA: Morgan Kaufmann.
20. Intel. *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. 2001 [cited 2001; Available from: http://developer.intel.com/design/pentium4/manuals/245470.htm
21. Mead, C., *Analog VLSI and Neural Systems*. 1989, Reading Massachusetts: Addison-Wesley.
22. Mahowald, M.A., *Computation and Neural Systems*. 1992, California Institute of Technology.
23. Douglas, R. *Fifteen years of Neuromorphic Engineering: progress, problems, and prospects*. in *Brain Inspired Cognitive Systems - BICS2004*. 2006. University of Stirling, Scotland, UK.
24. Holler, M., et al. *An Electrically Trainable Artificial Neural Network (ETANN) with 10240 "Floating Gate" Synapses*. in *International Joint Conference on Neural Networks*. 1989.
25. Nestor, I., *Ni1000 Recognition Accelerator - Data Sheet*. 1996. p. 1-7.
26. Orr, M.J.L., *Introduction to Radial Basis Function Networks*. 1996, Centre for Cognitive Science, University of Edinburgh: Edingburgh.
27. Chua, L.O. and T. Roska, *Cellular Neural Networks & Visual Computing*. 2002: Cambridge University Press.
28. Balya, D., et al., *A CNN Model of a mammalian Retina.* Int. N. Circuit Theory and Applications, 2001.

29. Murray, A.F. *The Future of Analogue Neural VLSI*. in *Proceedings of the Second ICSC International Symposium on Engineering of Intelligent Systems*. 2000.

30. Graf, H.P., L.D. Jackel, and W.E. Hubbard, *VLSI Implementation of a Neural Network Model.* IEEE Computer, 1988. **21**(3): p. 41-49.

31. Culurciello, E., R. Etienne-Cummings, and K. Boahen, *An Address Event Digital Imager.* IEEE J. Solid-State Circuits, 2003. **38**(2).

32. Rao, Y.N., et al., *Stochastic Error Whitening Algorithm for Linear Filter Estimation with Noisy Data.* Neural Networks. **16**: p. 873-880.

33. Choi, T.Y.W., et al., *Neuromorphic Implementation of Orientation Hypercolumns.* IEEE Transactions on Circuits and Systems II - Analog and Digital Signal Processing. **52**(6): p. 1049-1060.

34. Hammerstrom, D. *A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning*. in *International Joint Conference on Neural Networks*. 1990. San Diego.

35. Hammerstrom, D., *A Digital VLSI Architecture for Real-World Applications*, in *An Introduction to Neural and Electronic Networks*, S.F. Zornetzer, et al., Editors. 1995, Academic Press: San Diego, CA. p. 335-358.

36. Hennessy, J.L. and D.A. Patterson, *Computer Architecture: A Quantitative Approach*. 3rd ed. 2002, Palo Alto, CA: Morgan Kaufmann.

37. Ramacher, U., et al. *Multiprocessor and Memory Architecture of the Neurocomputer SYNAPSE-1*. in *World Congress on Neural Networks*. 1993. Portland, OR.

38. Means, R. and L. Lisenbee. *Extensible Linear Floating Point SIMD Neurocomputer Array Processor*. in *Proceedings of the International Joint Conference on Neural Networks*. 1991. Seattle, Washington.

39. Wawrzynek, J., et al., *Spert-II: A Vector Microprocessor System.* IEEE Computer, 1996: p. 79-86.

40. Bengtsson, L., et al., eds. *The REMAP Reconfigurable Architecture: a Retrospective*. FPGA Implementations of Neural Networks, ed. A. Omondi and Rajapakse, Springer-Verlag.

41. Mountcastle, V., *Perceptual Neuroscience* **-** *The Cerebral Cortex*. 1998, Cambridge, MA: Harvard University Press.

42. Mountcastle, V.B., *An Organizing Principle for Cerebral Function: The Unit Model and the Distributed System*, in *The Mindful Brain*, G.M. Edelman and V.B. Mountcastle, Editors. 1978, MIT Press: Cambridge, MA.

43. Braitenberg, V. and A. Schüz, *Cortex: Statistics and Geometry of Neuronal Connectivity*. 1998: Springer-Verlag.

44. Johansson, C., M. Rehn, and A. Lansner, *Attractor Neural Networks with Patchy Connectivity.* Neurocomputing, 2006. **69**: p. 627-633.

45. Hecht-Nielsen, R., *A theory of Thalamocortex*, in *Computational Models for Neuroscience – Human Cortical Information Processing*, R. Hecht-Nielsen and T. McKenna, Editors. 2003, Springer.

46. Granger, R., *Engines of the brain: The computational instruction set of human cognition.* AI Magazine, 2005.

47. Granger, R., et al., *Non-Hebbian properties of LTP enable high-capacity encoding of temporal sequences.* Proceedings of the National Academy of Sciences of the United States of America, 1994. **91**: p. 10104-8.

48. George, D. and J. Hawkins, *Invariant Pattern Recognition using Bayesian Inference on Hierarchical Sequences*. 2004.

49. Grossberg, S., *Adaptive resonance theory*, in *The encyclopedia of cognitive science*. 2003, Macmillan Reference Ltd: London.

50. Lansner, A. and A. Holst, *A Higher Order Bayesian Neural Network with Spiking Units.* Int. J. Neural Systems, 1996. **7**(2): p. 115-128.

51. Johansson, C. and A. Lansner. *Towards Cortex Sized Artificial Nervous Systems*. in *Knowledge-Based Intelligent Information and Engineering Systems KES'04*. 2004. Wellington, New Zealand: WelTec-Springer.

52. Arbib, M., *Towards a Neurally-Inspired Computer Architecture.* Natural Computing 2003. **2**(1): p. 1-46.

53. Roudi, Y. and A. Treves, *An associative network with spatially organized connectivity.* J. Stat. Mech.: Theor. Exp., 2004.

54. Renart, A., N. Parga, and E.T. Rolls, *Associative memory properties of multiple cortical modules.* Network: Comput. Neural Syst., 1999. **10**: p. 237–255.

55. Levy, N., D. Horn, and E. Ruppin, *Associative Memory in a Multimodular Network.* Neural Computation, 1999. **11**: p. 1717–1737.

56. Anderson, J., *Programming Considerations for a Brain-Like Computer*. 2005, Brown University.

57. Bailey, J., *A VLSI Interconnect Strategy for Biologically Inspired Artificial Neural Networks*, in *Department of Computer Science/Engineering*. 1993, Oregon Graduate Institute: Beaverton, OR.

58. Bailey, J. and D. Hammerstrom, *Why VLSI Implementations of Associative VLCNs Require Connection Multiplexing.* 1988 International Conference on Neural Network, 1988: p. 173-180.

59. Hammerstrom, D. *The Connectivity Requirements of Simple Association, or How Many Connections Do You Need?* in *IEEE Conference on Neural Network Information Processing*. 1987: IEEE Press.

60. Means, E. and D. Hammerstrom. *Piriform model execution on a neurocomputer*. in *International Joint Conference on Neural Networks*. 1991. Seattle, WA.

61. Boahen, K.A., *Point-to-Point Connectivity Between Neuromorphic Chips Using Address Events.* IEEE Transactions on Circuits and Systems II - Analog and Digital Signal Processing, 2000. **47**(5): p. 416-434.

62. Lazzaro, J.P. and J. Wawrzynek. *A Multi-Sender Asynchronous Extension to the Address-Event Protocol*. in *16th Conference on Advanced Research in VLSI*.

63. Hammerstrom, D. and J. Bailey, *Neural-Model, Computational Architecture Employing Broadcast Hierarchy and Hypergrid, Point-to-Point Communication*. US Patent Office, 1991: USA.

64. Lee, J.H. and K.K. Likharev, *CMOL CrossNets as Pattern Classifiers*. 2005, Stony Brook University: Stony Brook.

65. Likharev, K.K. and D.V. Strukov, *CMOL: Devices, circuits, and architectures*, in *Introducing Molecular Electronics*, G.C.e. al., Editor. 2004, Springer: Berlin.

66. Strukov, D.B. and K.K. Likharev, *Defect-Tolerant Architectures for Nanoelectronic Crossbar Memories.* Journal of Nanoscience and Nanotechnology, 2006.

67. Snider, G.S. and R.S. Williams, *Nano/CMOS architectures using a field-programmable nanowire interconnect.* Nanotechnology. **18**.

68. Likharev, K. and D. Strukov, *CMOL: Devices, Circuits, and Architectures*, in *Introducing Molecular Electronics*, G. Cuniberti and et al., Editors. 2004, Springer: Berlin.

69. Lansner, A. and others, *Detailed Simulation of Large Scale Neural Networks*, in *Computational Neuroscience: Trends in Research 1997*, J.M. Bower, Editor. 1997, Plenum Press: Boston, MA. p. 931-935.

70. Ananthanarayanan, R. and D.S. Modha. *Anatomy of a Cortical Simulator*. in *SC07 (Super Computing 2007)*. Reno, Nevada.

71. Maass, W., *Computing with Spiking Neurons*, in *Pulsed Neural Networks*, W. Maass and C.M. Bishop, Editors. 1999, MIT Press, A Bradford Book: Cambridge, MA.

72. Hänggi, M. [cited; Available from: http://www.isi.ee.ethz.ch/~haenggi/CNN_web/architecture.html.