On the power of sigmoid neural networks

Joe Kilian
NEC Research Institute
Princeton, NJ 08540
joe@research.nj.nec.edu

Hava T. Siegelmann
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
siegelma@paul.rutgers.edu

Abstract

We investigate the power of recurrent neural networks that apply the standard sigmoid activation function: $\sigma(x) = [2/(1+e^{-x})] - 1$. We show that in the noiseless model, there exists a universal architecture that can be used to compute any recursive function. As a result, basic convergence questions concerning these architectures are shown to be undecidable even for fixed-size networks. This is the first result of its kind for the standard sigmoid activation function; previous techniques only applied to linearized and truncated versions of this function. The significance of our result, besides the proving technique itself, lies in the popularity of the sigmoidal function both in applications of artificial neural networks and in models of biological neural networks. Our techniques can be applied to a much more general class of "sigmoid-like" activation functions, suggesting that Turing universality is a relatively common property of recurrent neural network models.

1 Introduction

We consider the power of recurrent first-order sigmoidal neural networks. In their simplest form, an N-state recurrent neural network is an N-dimensional dynamical system over a bounded subset of the reals (in this paper, over the solid N-cube $[0,1]^N$), and can be expressed as a quadruple (N, W, Θ, f). Here N is the dimension of the network, $W = \{w_{i,j} \in \mathbb{R} | \ 1 \le i,j \le N\}$ and $\Theta = \{\theta_1, \ldots, \theta_N\}$ are called the *weights* (or constants) and $f : \mathbb{R} \to [0,1]$ is called the *activation function*. Each neuron i computes its next state, $x_i(t+1) \in [0,1]$, by the formula

$$x_i(t+1) = f\left(\left(\sum_{j=1}^{N} w_{i,j} x_j(t)\right) - \theta_i\right). \qquad (1)$$

The computational and general dynamical properties of recurrent neural networks depend intimately upon the choice of the activation function. For example, if f is a linear function, then the system is essentially computing repeated matrix multiplications on an initial vector. If f is the Heaviside function given by

$$f(x) = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{otherwise,} \end{cases}$$

then each neuron takes on a value in $\{0,1\}$, and the system becomes finite-state. These qualitatively different behaviors motivate the study of the power of neural network models under different activation functions.

1.1 Previous Work

Pollack [8] proposed a recurrent net model that is Turing universal. This model consists of a finite number of neurons of two different kinds, having linear and Heaviside responses; the activations were combined using multiplications as opposed to just linear combinations. Thus, this model does not fall into the framework given above.

Siegelmann and Sontag first demonstrated the Turing universality of first-order neural nets for a specific activation function ([10], [11]). Their activation function, known as the ***saturated linear function*** is defined by

$$f(x) = \begin{cases} 0 & \text{for } x \le 0 \\ x & \text{for } 0 < x < 1 \\ 1 & \text{for } x \ge 1. \end{cases} \qquad (2)$$

They demonstrated the existence of a single choice of weights (and hence a constant number of nodes) such that by choosing the initial state of the neurons one could simulate the behavior of an arbitrary Turing machine. More recently they have shown a hierarchy of computational power depending on whether the weights are allowed to be integers, rational numbers or arbitrary reals [12]. Koiran [5] generalized their Turing universality result in [10] to more general saturated functions (ones that eventually become constants in both ends), not necessarily saturated-linear ones.

Given these results, it is natural to ask whether one can prove Turing universality results for activation functions used in practice. One activation function widely considered in the literature is the *sigmoid* function, defined by

$$\sigma(x) = \frac{2}{1+e^{-x}} - 1. \tag{3}$$

Much effort has been directed towards the practical implementations of sigmoidal neural networks applications ([1], [2], [3], [9], [13]). However, almost no previous theoretical work was done on such networks, mainly because of technical difficulties encountered with the sigmoidal function. The primary difficulty is in implementing noise-free logical operations. For instance, even if values x and y were guaranteed to have "ideal" $0/1$ values, we still cannot exactly compute the logical AND of x and y in our model. Not only will the answer not take on an "ideal" $0/1$ value, but it will take on slightly different values depending on whether $(x, y) = (0,0)$ or $(x,y) = (0,1)$. In particular, continuous fluctuations in the state of the finite control will send unmanageable amounts of noise throughout the entire system. Indeed, just the fact that it remembers state information will in subtle ways corrupt the data. The previous techniques in proving universality can not be applied here.

We show the existence of a finite dimensional universal neural network for a large class of activation functions that includes the standard sigmoid. We call these functions *valid sigmoids*.

As a corollary of our result, there is no computable limit on the running time of a valid-sigmoidal neural network. Also, if one wishes to emulate a valid-sigmoidal neural network using fixed-precision arithmetic, one cannot fix in advance the number of bits of precision. Thus, our construction may be thought of as a negative result concerning real-life valid-sigmoidal neural networks. One cannot automatically assume that a neural network converges or enters a detectable oscillatory state within any reasonable time bound. Also, one cannot *a priori* ignore the presence of even the slightest noise or roundoff error - since our construction is exquisitely sensitive to both effects.

The rest of the paper is organized as follows. In Section 2, we introduce alarm clock machines and prove that they are Turing universal. We end this section by showing how to substitute the alarm clocks with counters that behave in a restricted manner. In Section 3, we describe how to simulate alarm clock machines by first-order neural networks having a valid-sigmoidal activation fucntion, thus proving their universality.

2 Alarm Clock Machines

An *alarm clock machine* consists of a restricted finite control that has access to a finite number of alarm clocks. Each alarm clock c_i has a variable period p_i. If the clock alarms at time t_i, then the clock will next alarm at time $t_i + p_i$. Until it is woken by one or more alarm clocks, the finite control is required to spend its time in a memoryless "sleep" state that prevents the data from being corrupted faster than it can be repaired. When woken, the finite control is allowed to run for a constant number of steps, in which it may perform operations such as delaying a clock (so that it next alarms at $t_i + 1$ or lengthening the clocks day (setting $p_i = p_i + 1$) before going back to sleep.

Formally, an *alarm clock machine* $\mathcal{A}$ is a triple (F, k, c) where $k, c \geq 1$ and F is a function from $\{0,1\}^{kc}$ to a subset of ACTION, where

$$\text{ACTION} = \{\textit{delay}(i), \textit{lengthen}(i) | 1 \leq i \leq k\} \cup \{\text{halt}\}.$$

Here, k denotes the number of alarms clocks available to F, and F is a function that, based on the history of alarms from the last c time steps, halts and/or performs some simple operations on its clocks.

The input to (F, k, c) consists of $((p_1, t_1), \ldots, (p_k, t_k))$, where p_i denotes the period of clock i, and time t_i denotes the next time it is set to alarm.

The alarm clock machine operates as follows. For notational ease, we (conceptually) keep arrays $a_i(t)$, for $t \in Z$ and $1 \leq i \leq k$, with each entry initially set to 0. At time step T (initially 0), for $1 \leq i \leq k$, if $t_i = T$, then $a_i(T)$ is set to 1 and t_i is set to $t_i + p_i$. This event corresponds to clock i alarming. F looks at $a_i(t)$ for $1 \leq i \leq k$ and $T - c < t \leq T$, and executes 0 or more actions. Action $\textit{delay}(i)$ sets t_i to $t_i + 1$, action $\textit{lengthen}(i)$ sets p_i to $p_i + 1$, and action halt halts the alarm clock machine.

We make two stipulations on a legal execution of an alarm clock machine. First, if its input consists of all 0's, then F outputs the null set of actions (the machine is "asleep" until woken). Second, we require that $|p_i/p_j| < O(1)$ for all $1 \leq i, j \leq k$. That is, there is a positive upper bound on the ratio between any two clock periods. This second restriction allows us to more easily simulate our machines. In fact, in our proof of Turing universality, we guarantee that p_i and p_j differ by at most 1.

Theorem 1 *There exists an alarm clock machine (F, k, c) and a recursive encoding function enc(M) such that for all Turing machines M and binary inputs α, (F, k, c) halts on input enc(M, α) iff M halts on input α. Furthermore, if M halts in T steps, then (F, k, c) will halt in $2^{O(T)}$ steps.* ∎

We prove this result by a series of reductions to counter automata, which are known to be Turing universal.

2.1 Adder Machines

Definition 2.1 An adder machine $\mathcal{D}(k)$ is a machine consisting of a finite control and k adders. The operations on the adders are

- Inc(adder) for adders $i = 1 \ldots k$,
- Compare(Adder-i, Adder-j) is a function with the range $\{\leq, >\}$.

Definition 2.2 An Adder machine is said to be *simply controlled* if its finite control consists of a combinational circuit only, with no loops.

Lemma 2.3 For every adder machine $\mathcal{D}(k)$ with c states in the finite control, there is a simply controlled adder machine $\mathcal{D}'(k + c)$ with no more than c states.

(proof ommitted)

Now, we show the equivalence of adder machines and counter machines, thus proving that adder machines compute all recursive functions.

Definition 2.4 A counter machine $\mathcal{C}(k)$ consists of a finite control and k counters. The counters hold whole numbers; the operations on each counter are: Test for 0, Inc, Dec, and also No change. ([4])

Lemma 2.5 Adder machines and counter machines are linear time equivalent.

(proof ommitted)

Corollary 2.6 The class of functions computed by an adder machine is recursive. $\forall$ recursive function ϕ which is computed by a TM M in time T, $\exists$ an adder machine that computes ϕ in time $O(2^T)$.

Proof. Counter machines with at least four counters are known to simulate TM's in exponential time slowdown ([4], page 171, Lemma 7.4). ∎

2.2 Alarm Clock and Adder Machines

An alarm clock machine $\mathcal{A}$ is a special case of a counter machine, and hence $\mathcal{A} \subseteq \mathcal{D}$. Next, we show the other inclusion.

Lemma 2.7 Given a simply controlled adder machine $\mathcal{D}(k)$ that computes in time T, $\exists$ an alarm clock machine $\mathcal{A}(O(k^2))$ that simulates $\mathcal{D}$ in time $O(T^3)$.

The rest of this section is the proof of lemma 2.7

Given a simply controlled adder machine $\mathcal{D}$ with k adders $1 \ldots k$, we construct an alarm clock machine $\mathcal{A}$ which simulates $\mathcal{D}$. First, we overview the simulation shortly, and then prove its correctness in greater detail.

The alarm clocks $0 \ldots k$ of $\mathcal{A}$ simulate the adders. Alarm clock 0 is used as the "0" value to be compared with the other k alarm clocks. An adder i is simulated by the alarm clock i, by its temporal shift from alarm clock 0. That is, if adder i is set to n, then clock i has the same period as clock 0, but alarms n time units after clock 0 alarms. We always ensure that the period of the clocks is greater than their phase differences, thus avoiding wraparound problems. The correspondence between adders and the alarm clocks $1, \ldots, k$ is as follows:

Adder-i	Alarm clock-i
Inc(A-i)	delay(i)
Compare(A-i, A-j)	Compare shift phase of clocks i and j from 0

One subtlety is how to implement the Compare operation. The alarm clock machine's finite control is only allowed to remember the alarm sequence for the last $O(1)$ time steps. However, after simulating the tth time step of the adder machine any two alarm clocks may be phase shifted by $\Omega(t)$ time units. We need to perform the comparisons and represent this information in a way usable by the finite control. We accomplish this task by having a set of $O(k^2)$ auxiliary counters used to collect this information.

For each pair of clocks (i, j), $i < j$, the auxiliary clock ij determines whether the phase shift of clock i is less than or equal to the phase shift of clock j. The auxiliary reference clock 00 is used to synchronize the auxiliary clocks.

We now describe how the finite control uses the auxiliary clocks to compare the phase shift of the adder clocks. The period of the auxiliary clocks is maintained to be one greater than the period of the adder clocks. Thus, they alarm one time-step later in each successive cycle of the adder clocks. Conceptually the finite control uses these clocks to sweep through the adder clock cycle, and records the information it needs by delaying the auxiliary clocks.

Initially, we assume that all of the auxiliary clocks alarm in synchrony with clock 00, and that their phase shift with respect to clock 0 is less than that of any of the adder clocks (this is easily accomplished by suitably setting the initial conditions). The finite control works as follows:

- If clocks 00, ij and i alarm simultaneously, but not clock j, then the finite control delays clock ij once. If j but not i alarms, it delays clock ij twice.

- If clocks 00 and 0 alarm simultaneously, then the finite control waits for 2 more steps. At this point, the alarm pattern determines whether clock i's phase shift is less than, equal to or greater than that of clock j. The finite control then delays the auxiliary clocks so that they will again be synchronous.

It is easy to verify that each of these operations can be performed by remembering the alarm history of the last 4 time steps.

Once the finite control has the comparison information, it determines if the original adder machine would have halted, and halts accordingly. Otherwise, it determines adders the original machine would have incremented, and delays their corresponding clocks. Finally, in order to ensure that the phase shift for the adder clocks do not wrap around, the finite control lengthens the period of all of the clocks by 1.

To simulate the tth step of the adder machine, the alarm clock machine performs the comparisons in $O(t^2)$ time (the period is $O(t)$) and in $O(1)$ time it performs the requisite delays and lengthens the clock periods. Thus, $O(t^3)$ steps are required to simulate t steps of the adder machine.

2.3 Simulating Clocks with Counters

We now show how to simulate the clocks in the universal alarm clock machine with simple restricted counters. This will make the simulation of alarm clock machines by sigmoidal networks —described in the next section— easier. A similar idea was used in [5]. To simplify matters, we assume that the universal alarm clock machine runs a valid simulation of a simply controlled adder machine, and thus behaves as described above.

We implement each clock i with a *morning* counter M_i and an *evening* counter E_i. When the clock is in its steady state (neither being delayed or lengthened) with period p, the value of each counter has the following periodic behavior:

$$\cdots 0\,0\,0\,1\,2\,3\,4 \cdots (2p-1) \cdots 4\,3\,2\,1\,0\,0\,0 \cdots$$

To achieve this oscillatory effect, we put M_i and E_i p time steps out of phase. If M_i (resp. E_i) is decremented to 0 at time t, then E_i (resp. M_i) (which has been incrementing) starts decrementing at time $t+2$ and M_i (resp. E_i) starts incrementing at time $t+3$.

Thus, in its steady state, the system oscillates with a period of $4p$. We interpret a unit of clock time as four units of counter time, and identify the event that M_i turns from 1 to 0 with the clock alarming. Here is an example for $p = 3$:

$$\begin{aligned} M &= \cdots 3\,2\,1\,0\,0\,0\,1\,2\,3\,4\,5\,4\,3\,2\,1\,0\,0\,0\,1 \cdots \\ E &= \cdots 1\,2\,3\,4\,5\,4\,3\,2\,1\,0\,0\,0\,1\,2\,3\,4\,5\,4\,3 \cdots \end{aligned}$$

(This construction does not handle clocks with period 1. However, such clocks are not necessary for our alarm clock machine to be universal.)

We now show how to implement the *delay* and *lengthen* operations. For these operations, we assume that neither counter is equal to 0 and that it is known which counter is decrementing and which counter is incrementing. By inspection of our "program," one can verify that the finite control will always have this information within $O(1)$ time after it has woken up, and that it must wait only $O(1)$ steps before the nonzero condition is met. For example, when the finite control has received all of its comparison information, it can wait a few steps and ensure that the morning counters of all the comparison clocks and the 0 clock are incrementing, while the morning counters of all the adder clocks are decrementing.

To delay a counter, the finite control for one time step increments the counter it had previously been decrementing (and continues to increment the counter it was incrementing), and then at the next time step resumes its normal operations. Here is an example:

$$\begin{aligned} \text{steady state:}\quad M &= \cdots 3\,4\,5\,6\,5\,\mathbf{4}\,3\,2\,1 \cdots \\ E &= \cdots 2\,1\,0\,0\,0\,\mathbf{1}\,2\,3\,4 \cdots \\ \text{delay operation:}\quad M &= \cdots 3\,2\,3\,4\,5\,6\,5\,\mathbf{4} \cdots \\ E &= \cdots 2\,3\,2\,1\,0\,0\,0\,\mathbf{1} \cdots \end{aligned}$$

To lengthen the day's period, the finite control increments, for one time step, the counter that it was previously decrementing. For example:

$$\begin{aligned} \text{steady state:}\quad M &= \cdots 3\,4\,5\,6\,5\,\mathbf{4}\,3\,2\,1 \cdots \\ E &= \cdots 2\,1\,0\,0\,0\,\mathbf{1}\,2\,3\,4 \cdots \\ \text{lengthening:}\quad M &= \cdots 3\,4\,5\,6\,7\,8\,7\,\mathbf{6}\,5\,4\,3 \cdots \\ E &= \cdots 2\,3\,2\,1\,0\,0\,0\,\mathbf{1}\,2\,3\,4 \cdots \end{aligned}$$

Note that this operation will also alter the phase shift of the counters. However, since it will be performed on all of the clocks in the simulation, the relative phase shifts will be preserved.

3 Sigmoidal Networks are Universal

Definition 3.1 A *valid sigmoid* $\tilde{\sigma}$ is a function which satisfies:

1. Existence of exponential attractors:
$\exists A_1, A_2$ and $\exists B$ so that $\forall x \in$ range r_i $|\tilde{\sigma}(Bx) - A_i| < \frac{1}{2}|x - A_i|$, where $\frac{A_i}{2}, 2BA_i \in r_i$, $i = 1, 2$.

2. ϵ-closeness of the range including A_i and $2BA_i$: $\forall x \in r'_i \ |\sigma(Bx) - A_i| < \epsilon$ for some $\epsilon < .01$ and $A_i, 2BA_i \in r'_i$.
3. Differentiability around A_i: $\tilde{\sigma}'$ and $\tilde{\sigma}''$ exist in ranges $r''_i, i = 1, 2$ where r''_i includes all $y = \tilde{\sigma}(x)$ when $x \in r_i$.
4. Approximation of x by $\tilde{\sigma}(x)$ around $x = 0$. $\tilde{\sigma}(x) = x + O(x^2)$.

A valid sigmoidal network as described in the introduction is a network of N neurons, each of which updates its activation by

$$x_i(t+1) = \tilde{\sigma}(\sum_{i=1}^{N} w_{ij} x_j + \theta_i)$$

for constants w_{ij}, θ_i and any valid sigmoid function $\tilde{\sigma}$. Our main result is as follows.

Theorem 2 *Given an alarm clock machine $\mathcal{A}$ (with no input) that computes the function ϕ in time T, there is a valid sigmoidal network $\mathcal{N}$ that computes ϕ in time $O(T)$. Furthermore, the size of this network is linear in the number of clocks and the size of finite control of $\mathcal{A}$.*

For clarity, we describe the simulation in terms of the standard sigmoid function. It is very easy to generalize the result to any other valid sigmoid function.

We first show that the standard sigmoid is indeed valid.

- For every $b > 1$ and c, $\sigma(bx)$ has three fixed points. One is zero and the two others are denoted A and $-A$, as they differ only in sign. The larger b is, the closer A gets to 1. For example, using 15 decimal digits in the precision:

$b = 5$	$A = 0.98562369130483$
$b = 10$	$A = 0.999909121699349$
$b = 30$	$A = 0.999999999999812$
$b = 100$	$A = 1$

- For $b > 1$ and for every point x, $x \neq 0$, $\sigma(bx)$ is attracted exponentially to either A or $-A$ (depends on its sign). This can be verified by the Taylor expansion around bA, which results in the formula: $\sigma(bx) = \sigma(bA) + \sigma'(bA)b\epsilon + c\sigma''(bA)b^2\epsilon^2$,. For b sufficiently large, $\sigma'(bA)$ and $\sigma''(bA)$ decrease exponentially with b, and (for example)

$$|\sigma(bx) - A| = |\sigma(bx) - \sigma(bA)| < \frac{1}{2}(x - A).$$

In fact, we can achieve d^{-t} convergence for any $d > 0$ by a suitable choice of b.

Let c be a constant. In the case where we have the equation $\sigma(bx + c)$, the fixed points are shifted as a function of c, and the exponential convergence property holds provided that x is not equal to the unstable middle fixed point. The fixed points of the equation $\sigma(bx + c)$ ($b > 1$) are denoted as A_1 (≈ -1) and A_2 (≈ 1).

- For every x, $\sigma(x) = x + O(x^3)$. (Hence, if x is a small number then $\sigma(x) \sim x$.) This is proved by considering the Taylor expansion around 0.

Now, we show the simulation: Given an alarm clock machine $\mathcal{A}$, the network $\mathcal{N}$ that simulates $\mathcal{A}$ consists of three main components: a finite control, a set of counters, and a set of flip-flops.

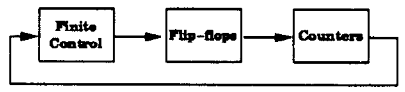

Figure 1: Block diagram of our simulation.

Implementing the Finite Control

It has long been known how to simulate any finite control $FC_{\mathcal{A}}$ of $\mathcal{A}$ by a network of threshold devices ([6],[7]). If the original finite control depends only on the last $O(1)$ time steps, the resulting threshold network can be made to be feed-forward.

We substitute each threshold device $x_i(t+1) = \mathcal{H}(\sum_{i=1}^{N} w_{ij}x_j + \theta_i)$ with a sigmoidal device $x_i(t+1) = \sigma(\alpha_a(\sum_{i=1}^{N} w_{ij}x_j + \theta_i))$, for a large fixed constant α_a. As long as the summation in the above expression is guaranteed to be bounded away from 0, the output values of the neuron using the sigmoid activation function will closely approximate the output of the neurons using the Heaviside activation function. By choosing α_a sufficiently large, we can make this approximation as close as we desire.

Note that the number of states in our "finite control" is in fact infinite, since every neuron can take on an infinite set of values. Since these values fall within a small neighborhood of either 1 or -1, we can conceptually discretize them; however the continuous nature of these values result in accuracy problems.

For each counter i, the finite control has two output lines (implemented as neurons), *Start-Inc*$_i$ and *Start-Dec*$_i$. When *Start-Inc*$_i$ is active (i.e., ≈ 1), it means that counter i should be continually incremented. Similarly, an active *Start-Dec*$_i$ means that counter i should be continually decremented. Most of the time both output lines are in an inactive state (i.e., ≈ 0). In this case counter i is treated according to the last issued command, allowing operations to be performed on the counter when the finite control is inactive. It will never be the case that both signals are simultaneously active.

Bi-directional Flip-flops

Recall that to avoid unrecoverable data corruption, we implement finite controls that converge to a constant "ground state" during the the long periods between interesting events. In order to maintain control of the counters during these quiet period, we introduce special flip-flop devices. These devices will have two stable states, and are guaranteed to exponentially converge to one of them during the quiet periods. While the finite control is active, it can set or reset the value of a flip-flop. Otherwise, the flip-flop maintains its current state.

The update equation of each flip-flop is $\text{ff}_i = \sigma(\alpha_{f1}(\text{ Start-Inc}_i - \text{Start-Dec}_i) + \alpha_{f2}\text{ff}_i + \alpha_{f3})$, where α_{f1}, α_{f2} and α_{f3} are suitably chosen constants.

Counters

Each counter of $\mathcal{A}$ is implemented via three sigmoidal neurons: one, called the *counter neuron*, holds the value of the counter, and the other two assist in executing the Inc/Dec operations. Let B be a constant $B > 2$. A counter with the value $v \in \mathbb{N}$ is implemented in a counter neuron with a value "close" to B^{-v}. That is, a value 0 in a counter is implemented as a constant close to 1 in the network. When the counter increases, the implementing counter neuron decreases by a factor of B.

Thus, at each step, the counter neuron is multiplied with either B or $\frac{1}{B}$. To do this, we use the approximation:

$$\sigma(\sigma(V + cx_i) - \sigma(V)) \approx \sigma'(V)cx_i \quad ,$$

for sufficiently small c and $|x_i| < 1$. Let V be the direction input signal, coming from the ith flip flop. That is, V converges to either A_1 or A_2. A counter neuron updates itself by the equation:

$$\begin{aligned} x_i(t+1) &= \sigma[\alpha_{c1}\sigma(\alpha_{c2}V + \alpha_{c3} + \alpha_{c4}x_i(t)) - \\ &\quad \alpha_{c1}\sigma(\alpha_{c2}V + \alpha_{c3}) + \alpha_{c5}x_i] \\ &\approx \sigma[(\alpha_{c1}\sigma'(\alpha_{c2}V + \alpha_{c3}) + \alpha_{c5})\alpha_{c4}x_i(t)] \end{aligned}$$

By a suitable choice of the constants $\alpha_{c1}, \ldots, \alpha_{c5}$, we have:

$$\begin{aligned} \alpha_{c1}\alpha_{c4}\sigma'(\alpha_{c2}A_1 + \alpha_{c3}) + \alpha_{c5} &= B \\ \alpha_{c1}\alpha_{c4}\sigma'(\alpha_{c2}A_2 + \alpha_{c3}) + \alpha_{c5} &= \tfrac{1}{B} \ . \end{aligned}$$

If the value of x_i is close enough to 0, we can approximate $\sigma[(\alpha_{c1}\sigma'(\alpha_{c2}V + \alpha_{c3}) + \alpha_{c4})x_i] \approx (\alpha_{c1}\sigma'(\alpha_{c2}V + \alpha_{c3}) + \alpha_{c4})x_i$.

The above discussion provides the intuition of why the update equation 4 of counter i computes either Bx_i or $\frac{1}{B}x_i$. When x_i is close to 1, and it is "multiplied by B" then it will in fact be drawn towards a fixed point of the above equation. This acts as a form of error correction.

3.1 Proof of Convergence: Sketch

Ideally, our finite-state neurons would all have $\{0,1\}$ values, our flip-flops would take on precisely two values (A_1, A_2) and a counter neuron would have a value of B^{-v}, where v is the value of the simulated counter. Unfortunately, it is inevitable that the neurons' values will deviate from their ideal values. To obtain our result, we show that these errors are controllable.

The proof of convergence is organized inductively on the serial number of the day. As the network $\mathcal{N}$ consists of three parts: FA, FF, and counters: for each part we assume a "well behaved input" in day d and prove a "well behaved output" for the same day. As at the first day, input to all parts is well behaved, the correctness follows inductively.

Lemma 3.2 In following three claims, $1 \Rightarrow 2$, $2 \Rightarrow 3$, and $3 \Rightarrow 1$.

(1) At each day d, FC sends $O(1)$ signals (intentionally non-zero) to the ffs. Each signal has an error bounded by $\alpha < .01$. The sum of errors in the signals of the FC during the dth day is bounded by the constant $\beta < .1$.

(2) At each day d, $O(1)$ of the signals sent by FF have an error of γ, where γ can be made arbitrarily small. The sum of error of all signals during the dth day are bounded by δ, where δ can be made arbitrarily small. $\{A_1, A_2\}$.

(3) At each day d, a counter with a value y acquires total multiplicative error $\zeta < .01$. That is, the ratio of the actual value with the ideal value will always be between .99 and 1.01.

Proof.
$1 \Rightarrow 2$:
Assume the finite control sends *Start-Inc*$_i$ and *Start-Dec*$_i$ to ff$_i$, and never these two values are both active. The update equation of each flip-flop is

$$\text{Start-Dec}_i\text{ff}_i = \sigma(\alpha_{f1}(\text{ Start-Inc}_i - \text{Start-Dec}_i) + \alpha_{f2}\text{ff}_i + \alpha_{f3}) \ .$$

- When either *Start-Inc*$_i$ or *Start-Dec*$_i$ is active, ff$_i$ is set to the new value. The error γ is bounded by
$$\gamma \le |1 - \sigma(\alpha_{f1}(1-\alpha) - \alpha_{f2} + \alpha_3)| \ .$$
It is easy to see that when $|\alpha_{f1}| - |\alpha_{f2}| - |\alpha_{f3}|$ increases, γ decreases. That is, γ is controllable. For example, if $\alpha_{f1} \ge \alpha^{-1}$, α_{f2}, and $\alpha_{f3} \le 20$ then $\gamma < .01$.
- When both *Start-Inc*$_i$ and *Start-Dec*$_i$ are small, ff$_i$ converges to its closer fixed point. If (Start-Inc$_i$ − Start-Dec$_i$) were exactly 0, then by an analysis similar to that of observation ??, ff$_i$ would be attracted exponentially to its closest fixed point. If α_{f2} is large enough, the fixed

p .ts can be made arbitrarily close to -1 and 1. Furthermore, noise from α_{f1}(Start-Inc$_i$ − Start-Dec$_i$) can be arbitrarily attenuated, since $|\alpha_{f1}\sigma'(\alpha_{f2}\text{ff}_i + \alpha_{f3})|$ can be made vanishingly small by a suitable choice of constants.

$2 \Rightarrow 3$:
The update equation of a counter x_i is given by

$$\begin{aligned} x_i &= \sigma[\alpha_{c1}\sigma(\alpha_{c2}V + \alpha_{c3} + \alpha_{c4}x_i) \\ &\quad -\alpha_{c1}\sigma(\alpha_{c2}V + \alpha_{c3}) + \alpha_{c5}x_i] \,. \end{aligned}$$

We show that by using such an update equation, the counter neuron x_i multiplies itself by either B or $\frac{1}{B}$ with a small controllable error.
Recall that for y small,

$$\sigma(\sigma(V+y) - \sigma(V)) \sim \sigma'(A)y.$$

We can choose constants $\alpha_{c1}, \alpha_{c2}, \alpha_{c3}, \alpha_{c4}, \alpha_{c5}$ such that

$$\begin{aligned} \alpha_{c1}\alpha_{c4}\sigma'(\alpha_{c2}A_1 + \alpha_{c3}) + \alpha_{c5} &= B \\ \alpha_{c1}\alpha_{c4}\sigma'(\alpha_{c2}A_2 + \alpha_{c3}) + \alpha_{c5} &= \frac{1}{B} \,. \end{aligned}$$

The deviation from this ideal behavior is caused by four elements:

- the error caused by approximating the difference equation by the differential.
- the error in $\sigma'(\alpha_{c2}V + \alpha_{c3})$ relative to the desired $\sigma'(\alpha_{c2}A_i + \alpha_{c3})$,
- the error caused by using the approximation $\sigma(x) \approx x$ for x small.

In the first case, the multiplicative error is proportional to $\sigma''(A_i)(\alpha_{c4}x_i)^2$. However, x_i shrinks exponentially (and then grows back in a symmetric manner). Hence, in a given day, these terms form two exponentially decreasing sums. In the second case, we can bound the resulting multiplicative error by a function of $\alpha_{c1}, a_{c_2}, \alpha_{c4}$ and $\sigma''(\alpha_{c2}A_i + \alpha_{c3})$ times the error in V relative to A_i. Finally, note that $\sigma(x) = x + O(x^3) = x(1 + O(x^2))$. Since x_i exponentially vanishes (and reappears), the multiplicative error terms form two exponentially decreasing sums.

By "summing" these multiplicative errors, we get the desired bound. We can then use the identity that

$$(1+\delta_1)(1+\delta_2)\cdots(1+\delta_k) = 1 + O(\delta_1 + \cdots + \delta_k),$$

when $\delta_1 + \cdots + \delta_k$ is sufficiently small.

$3 \Rightarrow 1$: Because the finite control is feed-forward, and since each counter alarms $O(1)$ times a day, the finite control will output (intentionally) nonzero signals only $O(1)$ times a day. We can bound the errors caused by the counters being nonzero as some constant c times the sum of the values of all the counters at every time in the day. By choosing the weights appropriately, we can in fact make c as small as desired. ∎

Acknowledgements

This research was partially supported in part by US Air Force Grant AFOSR-91-0343.

References

[1] Cleeremans A., D. Servan-Schreiber, and J. McClelland, "Finite State Automata and Simple Recurrent Recurrent Networks", *Neural Computation*, vol 1, No. 3, p. 372 (1989).

[2] Elman J.L., "Finding Structure in Time", *Cognitive Science*, vol 14, p. 179 (1990).

[3] Giles C.L., C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun and Y.C. Lee, "Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks," *Neural Computation*, vol 4(3), pp: 393-405 (1992).

[4] Hopcroft, J. and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[5] Koiran, P., *Universal Neural Networks*, Manuscript.

[6] W.S. McCulloch, W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys.* 5(1943): 115-133.

[7] M.L. Minsky, *Computation: Finite and Infinite Machines*, Prentice Hall, Engelwood Cliffs, 1967.

[8] Pollack J. B., *On Connectionist Models of Natural Language Processing*, Ph.D. Dissertation, Computer Science Dept, Univ. of Illinois, Urbana, 1987.

[9] Pollack J.B., "The Induction of Dynamical Recognizers", Tech Report 90-JP-Automata, Dept of Computer and Information Science, Ohio State U. (1990).

[10] Siegelmann, H. T. and E. D. Sontag, "Turing Computability with Neural Networks" *Appl. Math. Lett.* 4:6, November 1991.

[11] Siegelmann, H. T. and E. D. Sontag, "On the Computational Power of Neural Networks" *Proc. 5th ACM Workshop on Computational Learning*

[12] Siegelmann, H. T., and E. D. Sontag, "Neural networks with Real Weights: Analog Computational Complexity" — journal submission.

[13] Williams R.J., and D. Zipser, A Learning Algorithm for Continually Running Fully Recurrent Neural Networks, *Neural Computation*, Vol. 1, No. 2, p.270, (1989).