Super-Turing Computation: A Case Study Analysis

Keith Douglas

Carnegie Mellon University
2003

Dedication

This work is dedicated to my friends (particularly birds and cat).

Your affection has made this work pleasant. Much thanks goes out to my supervisory committee: Wilfried Sieg, Jeremy Avigad and Horacio Arlo-Costa, for making this work possible and manageable. I also thank CMU’s philosophy department for being a remarkable place to work, and in particular, the CAAE for helping out a non-ethicist. A final word of thanks also extends to my parents, who helped financially during the time of writing of this document.

Table of Contents

Dedication 2
Table of Contents 3
Introduction and Terminological Note 4
Part 1: Features of the Turing Machine Model 6
Part 2: Remarks on Super-Turing Computation in General 20
Part 3: Description of Siegelmann Analogue Neural Networks . 23
Part 4: Criticism of the Siegelmann Model 42
Part 5: Responses to Our Critics 52
Part 6: Conclusions and Future Directions 59
Works Cited 60

Figures

Figure 1: Hierarchy 17
Figure 2: Block Diagram of General Siegelmann Network 27
Figure 3: Example Siegelmann Network 30
Figure 4: Lever 52

Introduction

In 1936, Alan Turing introduced his characterization of the notion of computability in terms of the now famous “Turing machine” model. By the 1970s, this model was being scrutinized to see whether it could be improved upon so that the class of functions so “effectively” computable could be enlarged[1]. By the time of writing of this thesis, there have been a few dozen “super-Turing” models put forward and discussed in the literature by computer scientists, philosophers, mathematicians, physicists and others. Our work is a contribution to the philosophical discussion of this field.

We begin by discussing several key features of the Turing machine model that are important for understanding where it might be modified. Sieg (e.g., 1994, 1997, 2000) and Gandy (1980) have discussed some of these properties and formulated general presentations of the class of Turing-equivalent computers. However, some features of Turing’s model are not always emphasized in the literature (for reasons that we shall note) and so it is important to carefully consider them. Our discussion will focus on those features which, when suitably modified, lead to an appropriate super-Turing model, or whose emphasis is needed to understand it. Some of the features are only relevant when understood in the light of modification of others and thus might seem strange to mention in isolation.

Our second section introduces these “super-Turing” models in general terms, states the goals for their introduction, and defines some terminology used in the literature about them.

Following the above, we discuss one interesting putative super-Turing model from the literature, the analogue neural networks of Hava Siegelmann (1999). We explain how it generalizes an existing Turing-equivalent model.

[1] For representative surveys of “super-Turing computation”, see Ord (2002) and Cotogono (2003).

Further, we analyze the plausibility of Siegelmann's model in light of the previous discussion of features. We show that, despite its many virtues, it has various oversights which render it unsuitable as a model of computation.

Subsequently, we respond to some brief objections to our work that have been raised. For instance, we discuss whether Siegelmann's proposal should be regarded as a model of **computation**, and also why it should be regarded as a potential rival to the Turing model.

Finally, we conclude that Siegelmann's model, while it goes a long way in exploring super-Turing computation, does not convincingly develop the idea. We are thus left with the outcome that Turing's own viewpoint is basically still viable as an acceptable understanding of computability.

Terminological Note

An important piece of terminology we use (following Sieg (1994) and Gandy (1980)) is the distinction between comput**or** and comput**er**. A computor is a human who performs a computation; a computer is anything which computes, though generally we shall use it in contrast with computor.

Part 1: Features of the Turing Machine Model

In this section we draw attention to several features of the Turing machine model that are important for understanding putative super-Turing computation. We presuppose that the reader is familiar with both the Turing machine model generally (Turing 1964 [1936]; Kozen 1997; Epstein and Carnielli 2000) and with some of the assumptions and conditions which characterize it (Gandy 1980; Sieg 1994, 1997, 2000). We begin with a sketch of this model for the sake of reference and move on to discussing the features that shall be of importance: ones that are vital for understanding the debate over super-Turing computation.

We begin with a description of the Turing machine model. There are many equivalent machine models which are extensionally equivalent: they calculate the same class of functions. These are the Turing-equivalent models. Below we sketch one version, which is more or less that which Turing himself introduces (1964 [1936]). We focus on this particular version as it is the most familiar to most readers, and the decision is otherwise more or less arbitrary. Our taking of the Turing machine model concretely is parallel to what we do later with the Siegelmann model.

A Turing machine involves 2 primary parts. One is a sensor and control mechanism, the other an unbounded piece of paper, called the “tape” of the machine. The sensor and control mechanism is a finite state device of unspecified character. All that is important is that it is able to write any of a finite repertoire of symbols on the tape, move the tape left or right, and change its internal state based on the currently scanned tape symbol (again, one of a finite collection) and the existing state. For our purposes we assume a **deterministic** Turing machine, in which this state change is uniquely determined by the existing state and the scanned symbol.

A finite input gets represented in a suitable fashion by writing it on the tape in the appropriate alphabet. We then place the machine in the start state and next its state evolves according to

the state transitions of its finite control. This changes the tape contents as needed. We understand the state transitions as representing the procedure of computing the function. By the same token, reading off the results of the tape after the machine has entered a “halt” state is a suitable understanding of its output. This output is always a finite string after a finite time, as we assume the state transitions of the machine take place in non-decreasing time periods. (There is, of course, no requirement that the “halt” state ever be reached.) Since the tape is unbounded in length, the machine has as much scratch space as one wants to use when computing. It is this latter feature that increases the power of the machine over that of the finite control alone. It also allows the output to be of any finite size whatsoever. Thus, to use a Turing machine to calculate the value of a function, place its input on the tape (using the correct protocol), set the machine in motion by putting it in a start state, and then after a finite time, the result of the calculation (if there is one) is written on the tape using the appropriate protocol. The class of functions computable by the Turing machines is referred to as the (Turing-)computable functions, or alternatively, the class of recursive functions.

Finally, it is useful to note that since any particular Turing machine is a finitely describable object, one can feed an appropriate description of the state transitions of a Turing machine and its initial tape contents to another suitably constructed one and have it simulate the behaviour of the first. This can be generalized to a particular class of Turing machines, called the universal Turing machine[2], which can simulate the behaviour of any Turing machine whatsoever.

Before beginning the second section of this part, it is important to realize that many of the features discussed will seem

[2] Convention has it that Turing machines that compute the same function are regarded as “the same machine.” Strictly speaking one should speak of equivalence classes of machines, but that is not important for the present purpose.

unimportant or strange if one is used to usual presentations of the Turing machine. We beg the reader's indulgence: we shall see further in the sequel why each of the features to which we draw attention are important. The characteristics we shall focus on are: finiteness, protocol, symbolic computation vs. computation by measurement, procedural form of computation, communicability, representations, generality, timelessness, hierarchy.

The features we discuss can be motivated as they arise under three different headings: how they extend existing models in a minimalist way, how they allow the proposed model of computation to make some degree of sense as an engineering-like proposal, and finally, how they permit the proposal seem plausible as a model of computation or computing.

The first of the features to discuss, "finiteness", is mentioned by Turing himself (1964 [1936], pp. 116). This condition can be justified under all three headings. First, one has to be aware of which features **of** the Turing machine can be extended to yield putative super-Turing computations. The Turing machine is a model of finite computation in all relevant respects. For this reason it seems likely that in order to develop a super-Turing model **some** aspects of its finiteness will have to be modified[3] . Second, engineering considerations make it obvious that every machine we would seriously attempt to build is finite in most respects. Human beings cannot build something actually infinite in spatial extent, for instance. But there is slightly more controversy about whether we can build machines that are infinite in any other respect. In particular, we emphasize our six notions of finiteness to stress that we do construct machines that are (at least apparently) infinite in some respects or other. The question thus is whether we can use these infinite respects to perform computations. Third, our desire to understand models of **computation** also motivates this condition. Odifreddi (1989) is surely right when he says that computations are finite in some respect or other.

[3] This view is also supported by the proofs of Sieg (2000) and Gandy (1980) large classes of finite models are Turing equivalent.

However, the finiteness condition is a rather vague one as it stands. One must ask: “finite in what respect”? At least the following six ways come to mind:

(a) number of (computational) parts
(b) time of operation
(c) amount of memory
(d) number of states (“of mind”)
(e) power of recognition of symbols (“sensitivity”)
(f) precision of output of symbols

Qualification of (a) is needed: in some mereologies, all concrete things (save perhaps for some pointlike “atoms”) have continuum many parts. It is thus useful to state this condition in terms relevant to computing. (This also avoids worrying about whether those mereologies are correct.) Church in his review (1937) of Turing’s work mentioned this finite aspect of the “Turing machine.” Point (b) is not even discussed by Turing himself or his contemporaries: no doubt the idea of a computor calculating for $\aleph_0$ years would have struck Turing as too far fetched to be worth mentioning. It should, however, be pointed out (in light of some literature) that time of operation should be regarded as the “proper computation” time of a given computer[4]. This allows careful discussion of relativistic tricks, Zeus machines (à la Boolos and Jeffrey 1980), etc. We leave these applications to future discussions. Aspect (c) is explicitly discussed by Turing, and motivates the discussion of the finite but unbounded tape (memory) that forms a key feature of his machine model.

Similarly, point (d) is discussed by Turing. Even if one dispenses (as Turing does) with the psychologistic overtones of “of mind” in the above description, “something” with state is necessary. A system of Post-like systems which are numbered (and hence one can “GOTO” around) does not dispense with the state requirement: at

[4] For a computer equipped with a clock this can be loosely regarded as the number of cycles of operation.

minimum a “program counter” is needed. In this case the finite “program counter” is the relevant subcondition, entailing that the number of instructions be finite. (Alternatively, one can specify that the number of instructions is finite; then the condition that the “program counter” be finite follows.) Note that this “Post-inspired” approach does make the “program counter” part of the symbolic configurations in question. One must also remember that in many **applications** of the Turing analysis the importance of this “bit of state” is nil.

However, any application of the appropriate theory of computing to **concrete** proposals requires recognition of the notion of state. In particular, if we want to develop a putative super-Turing model, the state space of the relevant model is important. In particular we must ask: does it involve a continuous quantity? Are continuous state spaces plausible?

(e) and (f) are sufficiently linked that we can discuss them together. Turing motivates these finite aspects of his model by suggesting that if the computor were to use an infinity of symbols, some would be arbitrarily close together in shape (or however they are to be recognized as distinct) and thus be indistinguishable. It is important to realize (in the light of the distinction between computation by symbolic manipulation and computation by measurement discussed below) that Turing’s assumption that symbolic configurations must be of a certain finite fixed bound is **not** necessarily violated by certain kinds of “infinite” devices.

Each of (a)-(f) above can in principle be modified without modifying the finiteness of the other conditions. (In practice, (e) and (f) stand or fall together, but this is just a matter of making a reasonable choice of “calling convention” or protocol.) As we shall see below, our typical putative super-Turing model tries to obtain its power by relaxing some of these subconditions and keeping others.

Hence, it might be rejoined that the finiteness condition is so basic to any understanding of (effective) computation that any putative model of computation that denied it in any form is radically different enough to be unworthy of consideration. Our answer to this is simple: we wish to give as “fair a shake” to putative super-Turing models as possible. Obviously, the more ways a model fails to be finite, the less plausible it becomes.

After all, **what** stops a “presentation” or “use” of a completed infinite? We admit this latter is unlikely, but why rule it out *a priori*, especially as we shall see, what counts as an infinite “object” is not obvious.

The second feature of the Turing machine model to discuss is its “computational protocol.” This is how a Turing machine is to be interpreted as computing, and thus we can see how “protocol” can be motivated as a feature under all three of the above conditions. Since the Siegelmann network we discuss later makes essential use of its protocol, it is vital that we also specify the protocol of the Turing machine. Moreover, it is also a good engineering consideration: without the protocol the models do not describe a process of any **use**. A model of computation would be completely unrealistic if we were told that it was merely a matter of “reading off” in some unspecified fashion some “infinite” property of some arbitrary system as the computation. Further, by understanding the operations of a machine as it transforms well specified input to well specified output, we can see that the machine may well be viewed as computing something or other.

In Turing’s original presentation, the “calling convention” (one form of protocol suitable for “procedural” or “imperative” computing) involves a scratch space. Turing decides to interleave scratch with output (1964 [1936], pp. 121). We could pick another protocol; for instance, that the left of the initial position of the head is for scratch and input, and the right for output. This choice is not critical. So long as there is an unbounded amount of “storage” for any such protocol, the model captures the same class

of computable functions. Our presentation of the Turing machine model also involves the use of special computational states, the “halt” states (“accept” and “reject”), entering which ends the computation. This halting is also a sensible part of the protocol as it makes “obtaining the output” easier. We use the “accept” and “reject” states when the machine is used to recognize languages.

Our third feature can be put in terms of the distinction made in the 1950s and 1960s between computation by measurement vs. computation by symbolic manipulation. The former is also sometimes referred to as “dynamic” computation. Making a model for use in a wind tunnel, then physically simulating a system of interest, etc. is an example of the former sort of computation. The Turing machine model is an example of a description of the latter. It must be said, however, that upon closer inspection, this apparent distinction is difficult to pin down. In fact, the question of how exactly one should distinguish the two approaches to “computations” is a key issue in the later discussion of the Siegelmann proposal. Intuitively speaking, the Turing machine model does not characterize something that computes by measurement, because in general the constructions performed by the Turing machine are not of the “same form” as what is being described. For instance, the differential equations describing flow of air over an airplane which are solved by a Turing machine are not of the “form” of an airplane and atmosphere in the way that a scale model plane and a wind tunnel would be. From the above we can see that this feature is motivated from engineering as well as computational considerations.

Symbolic computation connects directly to our feature of procedural computation. The inclusion of this feature can be motivated by noting that it reflects a virtue of the Turing model. We include it for this reason, even if it largely has to be given up later on.

The Turing machine works procedurally (or imperatively) as opposed to functionally or “processually”. As is well known, the

specification for a Turing machine includes its “program”, a sequence of tuples which form instructions which it is to follow in order to compute. (All procedural computation is symbolic, but not conversely.)

Communication is our next feature and can be motivated much the same way as the procedural form of computation feature above. However, it is also vital from the engineering perspective - we cannot take a proposal for an artifact seriously unless we are at least given a partial sketch of how it is to function.

Douglas Hofstadter’s (1979, pp. 562) remark concerning communication in the context of the Church-Turing[5] thesis (italics in original) is part of the motivation for our inclusion of this feature:

> “Suppose there is a method which a sentient being follows in order to sort numbers into two classes. Suppose further that this method always yields an answer within a finite amount of time, and that it always gives the same answer for a given number. *Proviso:* Suppose also that this method can be communicated reliably from one sentient being to another by means of language. *Then:* Some terminating FlooP program (i.e. general recursive function) exists which gives exactly the same answers as the sentient being’s method does.”

The “instructions” by which the Turing machine is to compute (i.e. write symbols, move the tape and change state) are communicable in human languages (including, now, programming languages or

[5] We shall not debate the various forms of this thesis in the present work. We shall take it to be a thesis about how to understand “effective computability” - normally by identifying it with Turing-computable functions or any of the extensionally equivalent understandings. See Odifreddi (1989) for a survey of nuances and subtle variations on the thesis. Some of the work concerning super-Turing computation takes itself as finding counterexamples to the thesis.

mathematics)[6]. Clearly, however, as one can describe noneffective procedures, the putative procedure has to meet pretheoretic notions of effectiveness. Nevertheless it is an important consideration when we recall that Turing's approach originates as analysis of human computing (Sieg 1994; Sieg 1997; Sieg 2000; Gandy 1980; Gandy 1995 [1987]). It seems likely that a non-communicable method is an uneffective one, (as Hofstadter would suggest), but this is by no means certain.

Next we shall discuss the question of representation[7] of what is being computed. Turing considers easily understood representations (see later) and demonstrates their merits with a clear discussion of their properties and those of the computor responsible for reading and writing them. The feature of representations is absolutely vital for engineering considerations. Not only must we know how the functioning of the system proceeds, we must also know how properties of the system are used to solve the problem we wish to solve. For example, in civil engineering we might design a pipe system. We have to pick a material for our pipes that has certain desired properties: for example, we might want it to have a certain tensile strength, a certain (lack of) solubility in water, and a certain cost per unit mass. The engineering proposal for using this material must illustrate that the material selected has these virtues. Similarly, the properties of a computer being discussed must be the ones relevant to computation. This is precisely what representation is about and hence is the key engineering feature.

[6] Since the present author has almost no musical talent whatsoever, we shall ignore Hofstadter's (fanciful, even by his own admission) possibility that there is a musical procedure for (e.g.) deciding mathematical statements.

[7] In this work, we use "representation" as it is used in electrical engineering: state of a system used to represent some external object or its properties. For example: the pattern of voltages (say) in a circuit [high][low][low][high] represents the bit string "1001", which in turn can represent (say) the number 9 in the usual fashion. This usage should not be confused with the notion of "representability" and related terminology as used in proof theory.

The inclusion of the feature of representation also can be justified by noting that in order to compute at all one needs to compute **something** and have access to the appropriate parameters and data for the computation in question. Representations allow one to do this.

We shall restrict ourselves in the case of the Turing machine to representations of functions of natural numbers to natural numbers (and of course natural numbers themselves), as other sorts of functions computable (or tasks performable) by the Turing machine can be reduced to such. It is possible, as we now know, to reduce the number of symbols used by the Turing machine to two, commonly written[8] as “0” and “1”. Thus we can represent a number by writing its numeral in an appropriate binary string. A Turing machine with more symbols can make due with less internal state than one with less and conversely, but the computational power of all these variations is the same. It is only required that the number of distinct symbols be finite (as we have seen above) and at least two. A function to be computed by the machine is represented by the class of transformations from input to output via internal state changes. Any given sequence of such transformations is a particular computation of a function on its argument (input). The internal state changes can loosely be regarded as the steps performed in the computation.

Our next consideration of note in the Turing approach is “generality”. The universal Turing machine requires us to “need” only one Turing machine. This virtue we would like to extend to any putative super-Turing model, and so we included it in our list of features. It is also a useful engineering feature: engineers like machines that have the virtues of their predecessors.

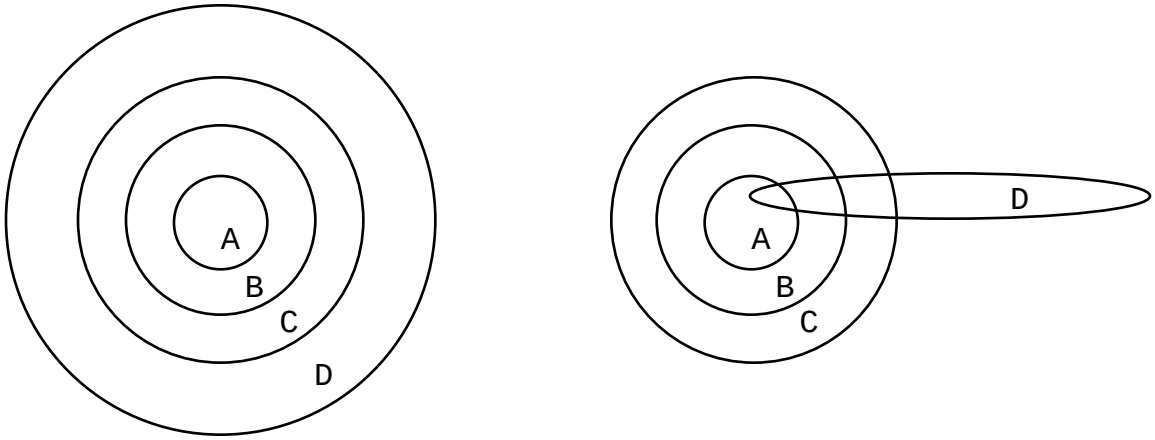
[8] Curiously, this way of putting it is more vital to what follows than it might appear. Our way of describing matters acts as an “intuition pump” that we know how representation of numbers, etc. works in a Turing machine. Or, in any case, our descriptions shows more is known about the Turing-style representations than those of Siegelmann model we analyze in what follows.

Our next consideration is perhaps the most unusual for some readers. This is the “timelessness” of the Turing model. Timelessness is included as a feature “in retrospect.” That is to say, in order to understand how the Siegelmann model extends the Turing one, we need to discuss its use of the time variable. In particular, we need to know about how the time variable interacts with the Siegelmann protocol, so the feature of “timelessness” is also motivated as a feature under the perspective of engineering.

The Turing machine may calculate for any finite period of time whatever. (Although it is not Turing computable to determine when infinite looping occurs, we consider a machine that goes into an infinite loop to have “stopped calculating” after a finite time. All we mean is that the machine cannot, for instance, decide the Fermat conjecture by trying all aleph null instances, as that would require infinite time if done at a constant rate.) We have already met the feature of “time” above in the discussion of “protocol”.

We close this subpart with a discussion of what might be called “hierarchy.” Hierarchy is another virtue we would like to recapture in an extension to the Turing machine model.

Kozen (1997) discusses computability in the fashion of increasingly complex languages being recognizable by given types of automata. He discusses how finite automata can recognize regular languages, how nondeterministic pushdown automata can recognize context free languages, and how Turing machines can recognize type 0 languages, etc. Since all regular languages are context free, and all context free languages are type 0, each model of computability recaptures the power of its predecessors. We suggest that a putative super-Turing model ought to fit nicely into this hierarchy. In Figure 1 (below), A represents the set of regular languages, B the context free languages, C the type 0 languages, and D some putatively super-Turing set of languages. (Alternatively, one can view these sets in terms of their respective automata.)



#1 #2

Figure 1: Hierarchy.

In Figure 1 part #1 reflects the desired hierarchy; #2 represents a putative super-Turing model that does not fit nicely into the hierarchy.

Our final task in this part is to summarize several of our findings in more exact terms. From this approach many similarities and differences between the Turing machine and the Siegelmann model stand out clearly. We can take each model to be discrete dynamical systems describable as a structure. Taking as starting point the work of Kozen (1997), we get rid of unneeded parts of his presentation and modify it slightly for ease of understanding and integration with our purpose. Thus, a Turing machine and its input are describable as a 7-tuple: $M=<Q,\Sigma,\delta,s,t,r,i>$.

Here, Q is a finite fixed set of *states.* Intuitively these correspond to parts of the machine or the computor and his aids. (In the latter case, one can imagine the computor keeping track of his stage in the computation by a further piece of paper or the like.) Generally one can describe these fixed states simply by giving each a natural number.

Σ is the *alphabet* of input, output and scratch, in our case {“0”,”1”}. It is important that this alphabet is fixed in advance

of the construction of the Turing machine in question, that it s finite, and that each is letter bounded above in size by a finite amount.

δ is the *transition function* between states, a function of the form: $Q \times \Sigma \rightarrow Q \times \Sigma \times \{l,r\}$. Intuitively, this function (given a state q and a symbol s), returns a new state q’ and a new symbol s’, as well as tells us to move the tape head (l)eft or (r)ight accordingly. s’ is the symbol to write, q’ the new state of the machine. Very often it is fruitful to state this function in the form of a giant table of tuples of the form <state,symbol,state’,symbol’,direction>. This table that characterizes the transition function can be viewed as a series of instructions. This illustrates how the Turing machine can be viewed as a procedural form of computation. Note that this table must be finite and since it describes a **function** by ensuring that no two tuples begin with the same state and symbol, this part of the description limits us to deterministic Turing machines.

Further, since the state table is in the above form, the instructions (and hence our description as a whole) are in principle human communicable.

Finally, by using an appropriate transition function and special states (see below) we can make a Turing machine a universal Turing machine and thus make it general in the appropriate way. (We shall not construct one here.)

s is an element of Q, called the *start state.*

t is an element of Q, called the *accept state.* This is one of our two halt states. The other, r (also an element of Q), is called the *reject state*. Needless to say, $t \neq r$ if the machine is used to recognize languages.

i is a finite string over Σ^*, the *initial tape contents*. We

assume that the rest of the unbounded tape (to the left and to the right of the string) is blank. Turing's remarks about representation make this a suitable elucidation.

We also require that the machine stay in the reject state or the accept state once entered. This requirement makes interpreting the function calculated by the machine easier to figure out; it corresponds to part of our protocol above. Similarly, we can describe (by suitable restrictions on δ) the other aspects of the protocol (e.g. how to place scratch, etc.).

Finally, it is interesting to note what happens to hierarchy in our **descriptions**. If we allow no scratch space at all (and hence the input tape is read only), the Turing machine is just a finite automaton, incapable of recognizing more than regular languages. If we add a stack (i.e. [approximately] a unidirectional tape with read/write access only to the first element) and nondeterminism (as understood in computer science), then we have a push-down automaton. Notice that the machine model hierarchy, when described in terms of implementation details, is not as uniform as it is when described in terms of functions computable (or languages recognized).

We have discussed some features of the Turing model of computation that are important to have in mind when contrasting it with putative super-Turing models. Next we discuss super-Turing models in general.

Part 2: Remarks on Super-Turing Computation in General

In this section we discuss briefly the purpose behind much of the research into super-Turing computation. In so doing, we describe motivations for this research, and define additional terminology from this field.

Throughout the present work, we use the term "super-Turing computation" as a synonym of "hypercomputation" or "hypercomputing" as used in some parts of the literature. As may be inferred from its name, super-Turing computation is computation of a sort that "goes beyond" Turing's formulation of computability and computation. For the most part, this is expressed in terms of new functions being computable, or new numbers being computable, or new languages being computable, etc. Since these are relatively interchangeable in many cases (such as in the original Turing machine presentation itself) it is understood that one characterization does not rule out applicability to others. For example, if one particular super-Turing model[9] is stated in terms of computable numbers, it is usually relatively straightforward to transform the model so that it applies to computing (or recognizing) languages instead. The fact that under most (if not all) known models of computation these are equivalent should not blind us to at least the conceivability of them not corresponding in some model.

"Going beyond" means for our purposes only computing members of a wider class. We are not interested in matters of computational complexity in this work. "Model" is used in the sense it is used in the philosophy of science (for our purposes the use in Bunge 1999 is sufficient). In other words, a model is taken to be a (usually) mathematized theory of a concrete system, process or thing[10]. A super-Turing model of computation is thus a theoretical model (like the original Turing model) of the process of "computing". A "Turing-equivalent" model is one that is equivalent with respect to computational power to the Turing machine model.

[9] We shall explain this notion as we shall use it in due course.

Siegelmann’s work seems to have a tension between being a purely mathematical theory and being a model **of** something in the sense we have just described. In what follows, since the Turing model is of something (human computability or computing) “concrete”, we also take Siegelmann as analyzing something putatively “concrete.” It is not (for the reasons we discuss) immediately obvious to what the Siegelmann model refers to beyond this sketch. Furthermore, we take Siegelmann’s model as literally as possible. Since she discusses signals and related matters, we shall use these to help understand intended implementations and thus her model.

This leaves us with "computing" and "computation" as undefined terms. We do not intend to define these, for any definition would likely beg the question for or against super-Turing models. Nevertheless, we answer some questions about what is and is not a computation or computing device in part 4 of the present work. (“Computability” is just the name we attribute to what is capable of being computed. “Human computability”, for example, is what is computable by humans.) We also presuppose that the computations of the super-Turing model are at least intended to be "effective" in some sense or other. Again, we shall not argue over this matter. Instead, as will be discussed later in greater detail, we shall assume that "effectiveness" is *prima facie* guaranteed by a super-Turing model which is an extension of an existing Turing-equivalent model. This will not always seem plausible, but it will at least give us a way to begin.

The various super-Turing models make use of differing assumptions about where to modify the Turing machine model. Hence, they also differ in motivation for their introduction as models of super-Turing computation. We focus on the assumptions used in our particular super-Turing model case study and must stress that there are other features of the Turing machine model that are subject to debate in other super-Turing models. As we are limiting ourselves to discussions of only one model in this work, we shall not discuss these other features except implicitly to note how the Siegelmann model we discuss is more (or less) plausible compared

to other models. For example, some super-Turing models make use of infinite inputs or outputs. The Siegelmann model does not; we comment on how this is part of an attempt to keep as many of the features of the Turing model as possible and to modify it minimally in order to putatively recognize more languages (and compute more functions, etc.).

Finally, we use "putative" in various places to emphasize the contentious nature of many super-Turing models. This is necessary because the field of super-Turing computation is controversial. Our goal is to give one model the benefit of the doubt.

We have met some terminology and the motivation behind super-Turing models in general. Next we shall meet a specific example of such, the analogue neural networks of Hava Siegelmann.

Part 3: Description of Siegelmann Analogue Neural Networks

In this section we discuss a representative promising super-Turing model of computation, the analogue neural networks elucidated by Hava Siegelmann (1999). Since this approach to computation is not as well known as the Turing machine model, we shall discuss it in substantial detail. In passing we shall refer to our features from part 1 to prepare us for the criticism based on these features in part 4.

Siegelmann introduces her version of neural networks on page 19 of her monograph *Neural Networks and Analog Computation*: *Beyond the Turing Limit*. A neural network is considered to be a finite collection of N processors. These processors are elementary in the sense that they are not further decomposable, computationally speaking. Each processor has a local state $x_i(t)$. "t" here represents a discrete time index. We assume that at time 0 all the weights[11] are initialized, the processors (nodes) are connected in an appropriate fashion, and then the input is fed in. The network is then presented with a (representation of a) vector u_j of

[11] A **weight** of a connection between nodes in a neural network represents some quantity associated with the embodiment of the nodes or their connection (e.g. a resistance, potential, etc.). As noted above, the precision of the weights is of critical importance for the computational capacities of the network. Interpreting (as we are) neural networks as a model of computation, these are a part of the structure of the network that allows one to calculate, representing parameters of the calculation. The weights, together with the structure of the network and the activation function, determine which function(s) each network computes. Because of the immediately preceding considerations, it is difficult to say whether or not the operations of the Siegelmann network are human-communicable. If we allow the conceit that the nodes should be regarded as unanalyzed computational units, then we will grant that Siegelmann's networks perform human communicable operations. After all, she gives both a "network style" representation of a particular network and one that resembles a procedure in a procedural programming language. We include this point primarily to draw attention to the very strong proviso that is required in order to state the previous conclusion. That is, we are assuming that Siegelmann's model has left the operations of the nodes unanalyzed. We shall discuss this further in our criticism.

dimension M at each time step. Each component of this vector is an element of {0,1}. (See the protocol below.)

Thus at each step the network behaviour can be described as follows:

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij}\, x_j(t) + \sum_{j=1}^{M} b_{ij}\, u_j(t) + c_i\right).$$

Here we call the a_{ij}, b_{ij} and c_i the weights of the network. These have an initial value set by the net's designer. Depending from which domain these weights assume values, the network has different computational power. We shall focus on the case where the weights and activation values[12] are permitted to assume arbitrary real numbered values, as this (Siegelmann claims) is the case in which the network can perform super-Turing computations. This infinite precision is the first area where the Siegelmann model makes assumptions concerning infinite properties. The σ function in the above equation is called the activation function. Activation functions are of critical importance in what follows as they are the second of the assumptions the model makes concerning the infinite. Siegelmann discuses several of these. One important one is the truncated linear function:

σ(x) = 0 if x < 0
= x if 0 <= x <= 1
= 1 if x > 1

(See pp. 20-21 of her monograph for other activation functions.)

[12] **Activation values** are the time indexed values associated with a given node: each of these is a function of the weights of the nodes connected to a given node. In Siegelmann's networks these are linear combinations. The **threshold functions** then determine what a given node should transmit to the nodes to which it is connected, based on the given activation value at each time step.

Note carefully that this choice of function corresponds to a hypothesis that the Siegelmann networks are infinitely sensitive. Also note that neither hypothesis concerning the infinite requires infinite symbolic configurations of the sort that Turing rules out in the case of the Turing machine. Finiteness of types (e) and (f) above are not violated. This is so because the Siegelmann style networks do **not** compute by symbolic manipulation.

After selecting an activation function, we then pick l of the processors, and call those the outputs of the network. These transmit the value of the representation of the activation function of the l processors along an output line, as we describe below.

Siegelmann describes a protocol by which the dynamics of such a network can be used to compute functions and recognize languages. This involves restricting the input lines to two, called “data” and “validation”. The data line (D) carries a binary signal into the network. It goes high to indicate “1”, low to indicate “0”. The validation (V) line indicates that the data line is active, going high if input is present and low otherwise. The input to a network is coded as a signal along these two lines in the expected way. (As usual, the signal can represent numbers, characters, etc.) For example, if we wanted input of the number 42_{ten}, we would at time zero raise V high. Then at times 0...5 we send bits 1,0,1,0,1,0 respectively along D and then at time 6 drop V to low and hold it low forever afterwards.

By the same token, there are two data and validation lines, G and H, which function analogously to the input lines, but are used for output.

Thus, one can exactify classification in the expected way: a word is classified by a network (in time r with given appropriate weights) if starting from the initial state, one presents the word to the input lines, the output validation line H is high (has value 1) at time r and 0 at all times before. We read off the

classification from G. If G is high at time r, then the word is recognized, rejected if G is low.

From there, it is not too difficult to construct an appropriate elucidation of acceptance of a language. In particular, a language is accepted by a network if every word in the language is accepted by the network (per above) and its complement rejected. It is this use of the networks that Siegelmann suggests is the most important. She says (pp. 24) she will only discuss computations of functions directly when the Siegelmann network coincides in power with the Turing model.

Finally, we can show how networks compute functions. A (partial) function $\varphi(x)$ is computable by a given network if for every argument x presented to the network according to the protocol above:

(1) when $\varphi(x)$ is undefined, H stays low
(outputs all zeros). G may fluctuate between its two states but since H never goes high, this is irrelevant from the perspective of “using” the network.

(2) when $\varphi(x)$ is defined, there is an r, the response time such that:
G outputs the successive digits of $\varphi(x)$ from times r to r + |x| - 1, where || is the length of the bit string coding x.
and
H remains high from times r to r + |x| - 1, and is low at all other times.

See below for a block diagram of a Siegelmann-style network.

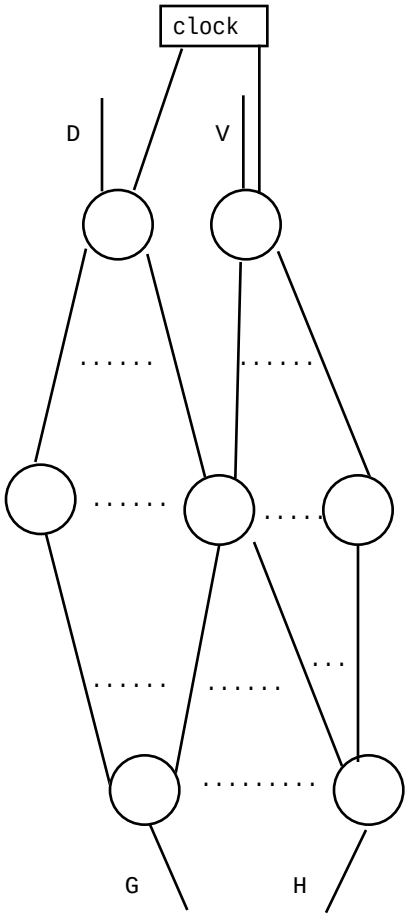


Figure 2: Block Diagram of General Siegelmann Network

Now that we have described in basic outline how a Siegelmann style network is to function, we discuss a family of such networks that supposedly do super-Turing computations.

Siegelmann introduces a system of three networks (pp. 67) which are designed to calculate the usual output with range in {0,1} from a network of wires and logic gates and thus simulate families of circuits. These three networks are described as the input network (which takes the appropriate input to the simulated circuits and stores it accordingly), the retrieval network (which takes an appropriate input from the input network and simulates it on the appropriate circuit), and a synchronization (and output) network that coordinates the first two networks and performs the appropriate global output. It is the retrieval network that is discussed in most detail, as it is the only part that performs

super-Turing computations and is also the most difficult to construct. The motivation (explained on pp. 16 of Siegelmann's book) for this particular system is that non-uniform circuit families are equivalent to non-recursive languages.

Page 65 of the monograph performs the construction of this network for the reader. The notation x_i+ = f(some other nodes or properties thereof) means that at node x_i at time t+1 it calculates[13] the function f() given the values of the other nodes at time t. u represents the input of the network, presented at time t = 0. It consists of n 1s, where n is the length of the coded circuit family. In other words, n is the number of significant figures of C, the encoded circuit family to be simulated, in an appropriate base 9 representation. This representation is chosen here in order to facilitate recognition of values. Rather than using a directly continuous operation, the base 9 representation uses a Cantor set[14] style.

[13] How this calculation is performed is not clear. Siegelmann seems to suggest that this would occur by an aspect of the network measuring some appropriate property of the relevant part of the rest of the network system.

[14] Cantor set encodings are used by Siegelmann in several places of her work to help avoid the problems with recognizing infinite precision numbers. Rather than having to distinguish between two close values by reading all the bits representing a value, the Cantor set encoding enforces "gaps" between" valid encodings. She says (pp. 34), concerning encoding of a stack as a binary number:

> "For example, in order to describe the first bit of the stacks 011•••1 and 100•••0, one must read the whole number."

This is especially important with infinite sequences of bits, as the usual encoding is not one-one. So instead we use base 9 (as in the example of Siegelmann's) as follows. We fix the digits used to represent parts of circuits to the set S = {0,2,4,6,8}. Then we use only the real numbers q of the form: $q = \sum_{i}^{\infty} \frac{a_i}{9^i}$ where each of the a_i is an element of S. Then the coding gives us the "gaps" property, since the numbers used now are sufficiently far apart; i.e. we have avoided numbers like $0.1111111..._{ten}$.

By picking certain strings of these base nine digits, Siegelmann can have the network she constructs do the appropriate simulation. She assumes that the gates occur at d+1 levels, where the input nodes to the circuits are at level 0, the single output at level d. Each gate has inputs only from the preceding levels and the value it computes is an input to the following level. (Different circuits may have different values of d.) A family of circuits then is a set of circuits such that there is a circuit which computes on inputs of length n for every natural number n. On page 62, she shows us a good way to encode these families of circuits, starting with how to encode a single circuit. Each level i begins with the digit 6. Levels get encoded sequentially, bottom to top. Each level has its gates encoded successively, 0 to indicate the start of a gate, a two digit sequence from {42, 44, 22} to indicate {AND, OR, NOT}, respectively. Then a sequence of digits over {2,4} encode which gates feed into the current level's gate. The jth position of this sequence is 4 if and only if the jth gate of the previous level feeds in the current gate, 2 otherwise. From there we can then encode entire families of circuits by placing their encodings sequentially, delimited by the digit 8 (placing each circuit encoding in reverse order to simplify proofs - this is not important for the simulation).

Once that is done, this completes the requirements for the constructed network, which we reproduce here:

$$x_i^+ = \sigma(9x_{10} - i) \qquad [0 \leq i \leq 8]$$

$$x_9^+ = \sigma(2u)$$

$$x_{10}^+ = \sigma(Cx_9 + x_0 - x_1 + x_2 - x_3 + x_4 - x_5 + x_6 - x_7 + x_8)$$

$$x_{11}^+ = \sigma((1/9)x_{12} + (2/9)(x_1 + x_3 + x_5 + x_7) - 2x_{13})$$

$$x_{12}^+ = \sigma(x_{11})$$

$$x_{13}^+ = \sigma(u + x_{14} + x_{15})$$

$x_{14}^{+} = \sigma(2x_{13} + x_7 - 2)$

$x_{15}^{+} = \sigma(x_{13} - x_7)$

$x_{16}^{+} = \sigma(x_{12} + x_7 - 1)$

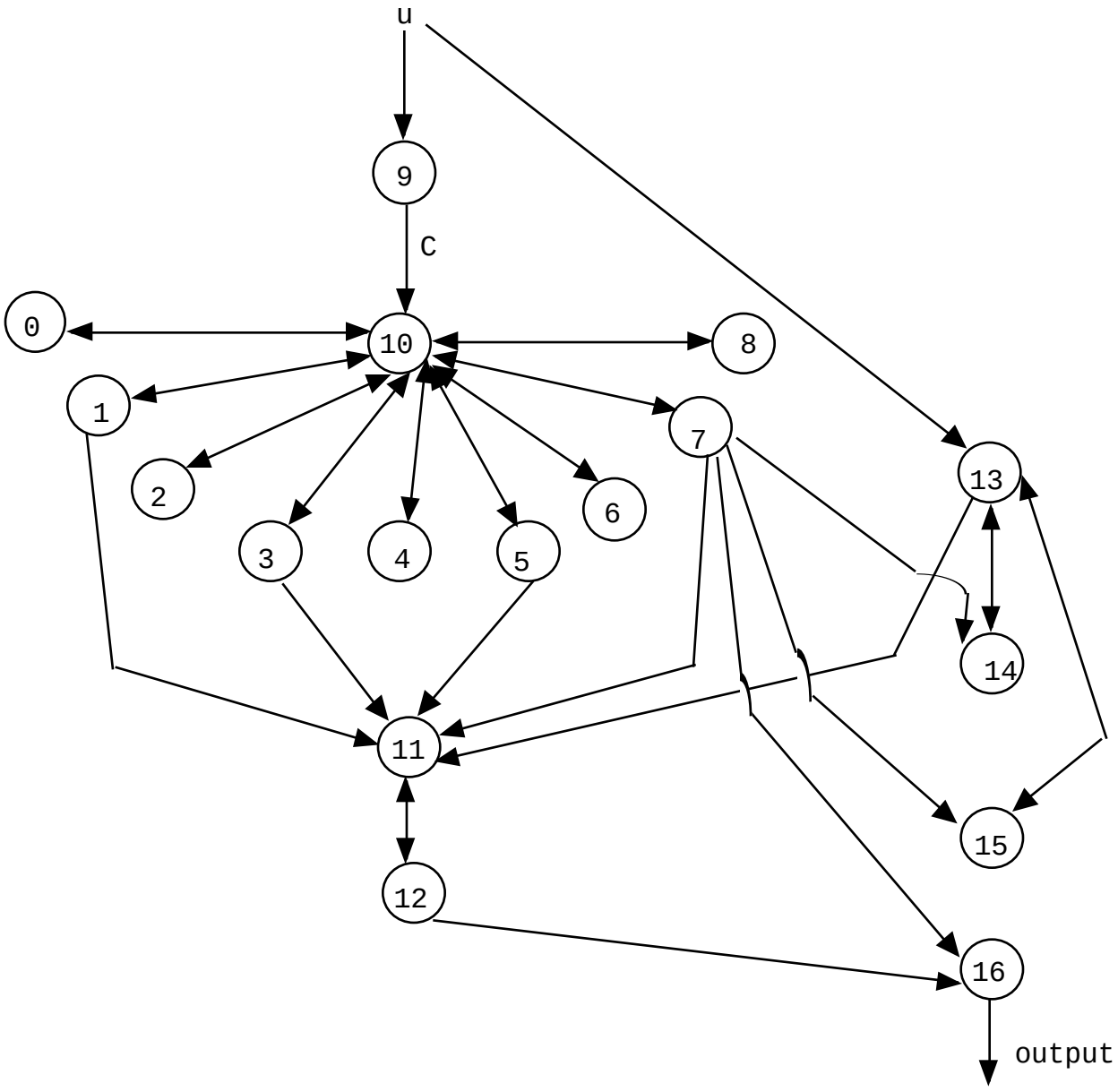


Figure 3: Example Siegelmann Network.

An important feature to note about this network is that it is recurrent. Note also we (and Siegelmann) have not depicted the

protocol directly, as the above part is only meant as a piece of a larger network. Siegelmann also emphasizes that all the weights, save the one marked C above, can be understood as being describable as rational values rather than arbitrary real numbered values. Since this weight can be an arbitrary real numbered value, it can do its trick by simply having the non-Turing computable arrangement of gates stored in it as a weight. (This makes it not too surprising that the Siegelmann networks can do **one** thing super-Turing: whether they can compute other super-Turing functions is not discussed.) How this works to perform super-Turing computation is simple: interpret the value of C as a non-Turing computable real number: all languages are recognizable in exponentially sized circuits, where size is understood as the total number of gates (Siegelmann 1999, pp. 16). Note that each circuit family to be simulated requires a different weight assignment C, and hence a different network. Hence, each language (at least on this approach) also requires a different network.

She does not give the proof of the above result herself, but instead appeals to a volume by Balcázar, Díaz and Cabarró (1995; hereafter B-D-C). However, this volume does not support her claim as fully stated.

B-D-C's text is primarily about computational complexity, and thus is outside our current subject. In particular, they only make use of a super-Turing model (Turing's own abstract "oracles" from his 1938 paper) in order to speed up computation. Siegelmann by contrast interprets their result (i.e. proving a complexity bound for simulating certain circuit families) as concerning super-Turing computation. This would only be true if the construction of a given network of circuits corresponded to a super-Turing computable function. B-D-C do not discuss these possibilities even in passing; the introduction to their work makes it clear that they are not considering matters of computability *per se*. Siegelmann's appropriation of their results is thus in error. In particular, we cannot know if Siegelmann's claim that her networks can be set up to recognize all possible languages over $\{0,1\}^*$ is

justified from **this** example. By contrast, a general proof concerning **all** of the Siegelmann networks is provided (earlier in the chapter) which demonstrates the equivalence of these networks with polynominal advice Turing machines. Here she makes essential use of her result that linear precision suffices - this avoids using infinite numbers of bits 'all at once' (pp. 60):

> "For this, we use the observation 'linear precision suffices,' which guarantees that if a network N computes in time T(n) then its 'T(n) truncated version' (to be formalized later) computes the same on any input of length n. T-truncated network [sic] can be specified with an advice of O(T) bits only; this completes the simulation."

(Siegelmann's proof of the linear precision bound can be found on pp. 68-70 of her work. Note: since the bound proved is of a certain complexity class only, the linear precision bound is of necessity inexact unless calculated for a given network. This would make using it rather difficult without a more detailed calculation. We assume in the rest of this thesis that the precision needed can be calculated exactly.)

By contrast, to simulate an advice Turing machine on the network, we Cantor-encode the advice for inputs of length n into a weight. It is thus a relatively straightforward matter to construct a network which retrieves (as in the above example) the advice from a weight and then passes along the input string and a weight to a Turing-machine simulation, and hence computes a function accessible only to an advice Turing machine[15] with appropriate truncation.

Returning to our specific case, it would appear that to make a non-Turing computable pattern of gates it one would need[16] an

[15] This is subject to a rather large proviso that we debate later on concerning infinite time. Nevertheless we have presented Siegelmann's argument as it stands, regardless of its plausibility.

[16] It has been suggested in the literature that one can make do with simply an irrational Turing-computable number (e.g., π), but this has never been proved to this author's knowledge. We ignore this possibility in what follows.

infinite circuit collection in a rather unusual pattern. Later Siegelmann suggests that exponential advice is required. We shall return to this in our criticism.

It is fruitful to close by elucidating the Siegelmann network description as a 5-tuple $N=<Q,\Sigma,\delta,i,r>$, paralleling our discussion in part 1 of the Turing machine.

Q is a continuous state variable. For simplicity we are pretending all the nodes of the Siegelmann network share one state space. We can do that by reducing all the activation values to a single real number for any given network. We do this by interleaving each of the digits from each activation value: $0.a_1a_2a_3a_4...$ and $0.b_1b_2b_3b_4...$ becomes $0.a_1b_1a_2b_2a_3b_3a_4b_4...$ in the case of a two node network. A similar procedure can be done for any Siegelmann network, as they all have a finite number of nodes. It is of no benefit to use a high-dimensional state space to elucidate the behaviour of the Siegelmann network for our applications, as the trajectories through such are no more illuminating than the ones through the one dimensional case. Further, by using a one dimensional state space all Siegelmann networks are represented as being similar in structure. This idealization of the description makes comparing them all as a class to the Turing machine more straightforward. It is also important to realize we are simplifying matters (see above) by considering that the “linearly truncated” versions of the network actually make use of dynamics that just do not move into the appropriate part of the state space

prior to the increased precision being needed[17]. In that sense one can see the state space used as actually evolving. Of course we would like to know by what mechanism this occurs as it is a vital part of understanding Siegelmann's model. More on this in our criticism, as Siegelmann does not attempt to answer this question.

The non-denumerable state-space raises some questions about Siegelmann's model we should answer. Doing so also shows some of the virtues and characteristics of the Siegelmann model. First: one might think Siegelmann's example network shows that rational values alone suffice as weights. Since these can be represented as pairs of (represented) integers, there should be no problem here. But there **is a** parameter that is explicitly **not** represented as a rational value.

As in usual representations of real numbers on computers (regardless of how this is to be accomplished), there will be ways to "break" the representation. A common one in ordinary computers is the breaking of the associativity of arithmetic. Since Siegelmann's networks require real valued weights, unless infinite precision real numbers are available, there will be a pattern of

[17] For example, suppose a network calculates 0.17 (exactly) as an intermediate value. The evolving network would represent this (somehow!) as 0.17 at one time, 0.170 later, 0.1700 later still, etc. as more and more precision is needed. Note that with these networks precision is always finite and hence the networks always have a finite state space. By contrast, a non-evolving network has to represent 0.17000000... to infinite precision from the time the value is calculated (or is "put" in a weight). This requires a non-denumerable state space as **each** of the digit after the (here) decimal point can in principle be different. This in spite of the fact that the computing of 0.17 exactly by some means may well involve a Turing-computable function. Note also that a typical Turing machine approach to computable real numbers involves representing them in some sort of functional way. (For example, we might represent π as a function in some programming language that sums an appropriate series.) This approach is not open to the Siegelmann network directly without modification of protocol because it apparently has to be able to perform operations on them directly. (This depends critically on how the representations of arguments and functions etc. is accomplished, of course.)

parameters (inclusive of inputs) where the representation breaks. For instance, real numbers in existing computers produce mathematically incorrect results under certain circumstances when magnitudes being added are very different. Consider the following example (adapted from Hennessy and Patterson 1998; as usual sub scripts denote number bases.). Let $x = -1.5_{ten} \times 10^{38}$; $y = 1.5_{ten} \times 10^{38}$; $z = 1.0$, all in IEEE single precision format. Then:

$$
\begin{aligned}
x + (y + z) &= -1.5_{ten} \times 10^{38} + (1.5_{ten} \times 10^{38} + 1.0) \\
&= -1.5_{ten} \times 10^{38} + 1.5_{ten} \times 10^{38} \\
&= 0 \\
\text{but:} & \\
(x + y) + z &= (-1.5_{ten} \times 10^{38} + 1.5_{ten} \times 10^{38}) + 1.0 \\
&= 0 + 1.0 \\
&= 1.0
\end{aligned}
$$

This sort of problem occurs however the numbers in question are represented in any usual format, and is not exclusive to the IEEE representation. By contrast Siegelmann's model, (were it implementable), would overcome this problem. It may be viewed that we are being uncharitable to the Turing machine model. Someone might claim that the sort of issue we are discussing here should be viewed as one for consideration of numerical methods and approximation theories and not for computability per se. We rejoin that our critic has not given an acceptable answer unless the defender of the Turing machine can show how it would overcome this issue. If Siegelmann's networks **were** of infinite precision as claimed, in a way this would render them super-Turing. We can call this a "weak" form of super-Turing computation. Siegelmann has given an otherwise Turing-equivalent machine model a feature it is not normally associated with - infinite sensitivity. (The plausibility of this is examined later.)

Second: can the Siegelmann network thus **actually** calculate any

general super-Turing **function**[18]? Here matters are not so clear. Since the protocol requires that any output be finite, the Siegelmann network can only compute the functions whose range of values can be expressed in a finite number of bits per value. It is thus not possible for it to output the values of certain functions which are calculable internally. Nevertheless, if infinite precision general real numbers are available as weights to a given network it is able to calculate certain restricted super-Turing functions. Note, however, that the current discussion does not yield a conclusion about how internal representations of the networks should be described. We are given some attributes of this representation, but not enough to develop the description further. More on this in the next point.

Σ is a finite input and output alphabet, usually {“0”,”1”}. Unlike in the Turing machine, we are not told what the internal alphabet is. It is not immediately clear that there even is one: if the Siegelmann network computes by measurement, then the internal representations of values are the magnitudes of some property or other, not values of some ‘alphabet’ at all. This raises further questions of “representations” as pertains to the Siegelmann network which are of vital importance. First of all, it is important to realize that (unlike the Turing model, which has finite representations throughout), the Siegelmann model has finite input and output, but infinite internal representations. How these representations function, i.e. how weights and other

[18] Note the difficulty even in computing a constant function. Since the weights of each node in a Siegelmann network are of infinite precision, outputting their value directly is impossible by the protocol described. This arises because such a constant is still an infinite precision number, and so outputting requires an infinite amount of time, followed by a signal to indicate that the output is finished. At best this would require a supertask. A suitable re-encoding would have to be found, and that is not suggested anywhere by Siegelmann. Moreover, such would have to handle rational values as well as surds, transcendental values, and (if noncomputable weights are allowed), even non-Turing computable numbers, like Chaitin’s constant. Of course, giving a finite representation of the latter sort of value cannot in general be done.

numerical quantities get “embodied” in the network, is never specified by Siegelmann[19]. Internal representations of numbers in her example network (somehow[20]) make use of Cantor sets, but it is not clear whether this approach is meant in general. This itself raises issues. If we recall her example of the stack encoding (part 3, above), it seems that her proposal presupposes that the stacks are being “read” in binary then transformed to decimal and then compared. For example: $0111_{two} = 7_{ten}$ is compared to $1000_{two} = 8_{ten}$.) See also the above discussion of the protocol as the questions of representation strongly interact with it. Second, while the network is given multiplication (after a fashion) and addition as primitive operations, their representation is not given either: i.e. which measurements are to be taken as performing these operations. The other characteristics of Σ are as in the Turing machine.

δ is the transition function of the Siegelmann network. Normally, one would describe it as a set of activation functions of all of the nodes of the network. But, as we have seen, we can place the activation values into one state space, so only one transition function is needed of the form $\delta:RxNx\{0,1\}^2 \rightarrow Rx\{0,1\}^2$, where R is the set of real numbers corresponding to the 1 dimensional activation activation function. N is the set of natural numbers, used as our time index. The set of pairs over {0,1} is the set of possible inputs and validation line signals at the given time.

δ then “returns” a new activation value represented as a number from the set of real numbers (the new value in the activation space) and two bits corresponding to the data ready signal and the output line.

[19] One should compare this to the Turing machine proposal where the fact that it calculates by symbolic manipulation makes the functioning of representations relatively straightforward, as we have sketched above.

[20] Since a set is an abstract object, strictly speaking one would **describe** the representation using them.

Not all transitions in the activation space are permitted, only those which respect the restriction (or its truncated version)

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{ij} x_j(t) + \sum_{j=1}^{M} b_{ij} u_j(t) + c_i\right)$$

we have discussed. This function should not be understood as corresponding to a **symbolic calculation** in the usual Turing machine sense. Instead the state function of the Siegelmann network refers to a property of the system that evolves in such a way as to be describable in such a fashion. This is how we understand computation by measurement.

Finally, note that since the state space of the Siegelmann network is nondenumerable, it is impossible in general to specify δ in terms of a table (like one often does in the Turing machine case). In fact, even the equations of the form mentioned above are elliptical in their description: in general, it is not possible to describe a Siegelmann network in completely finite terms, rendering its communicability suspect. It is, however (as we shall comment on below) possible, to describe the operations of the network somewhat procedurally if we grant the conceit that we know the exact value of the weights.

i is a finite string over Σ^*, the input contents. Note that this, strictly speaking, requires the protocol to delimit the input from the rest of the signal into the network (which is an infinite string). This input is a binary signal familiar to engineers, and so the finite string description is suitable. Like the class of Turing machines with its universal member, there is a Siegelmann style neural network that is similarly general (pp. 56-57). We can feed in a coded description of a Turing machine via the input to the Siegelmann network in much the same way as with the Turing machine itself. But it is unclear whether there is a general network that computes what all the Siegelmann style networks do. If we take the “weak” reading of super-Turing computation

introduced above, then there is of course such a network - the same universal network that computes all the Turing computable functions. In the case of the “strong” reading, matters are not so clear: Siegelmann’s monograph does not attempt to answer this question.

r is the time of calculation, characterizable as a (finite) natural number. The protocol needs this in order to manage the output properly, as strictly speaking, the Siegelmann network outputs forever. This parameter delimits the relevant portion. (This is exactly parallel to the input contents case.) The protocol requires the Siegelmann networks to continue operating forever, but after a finite time to assert that their output is ready. In this way they cannot perform the iterative output of a Turing machine except by fixing a protocol change by which successive approximations appear on the output sequentially. But these (without the change in protocol) would appear as one long output - the H flag would be high throughout. Nevertheless, since the protocol also requires output (if any) to occur after a finite time, the timelessness assumption of the Turing machine is exactly paralleled in this respect. It is also unclear who designs the output signal. Does the designer of the network have to know how many time slices into the network operation the signal is to be raised? Or, by contrast, does it work like the halt state of the Turing machine? In this case the “programmer” selects which state(s) will cease operation of the Turing machine or Siegelmann network, but does not have to worry about **when** these are to occur, merely which conditions are to provoke them. Siegelmann does not tell us. The choice between these possibilities is of vital importance in our debate on the merits of the Siegelmann proposal, as we shall see in part four.

Finally, hierarchy also requires that the Siegelmann description collapses to appropriate weaker models given appropriate restrictions. Having already illustrated how the Turing machine collapses, we only need to discuss how the Siegelmann model collapses to the Turing machine. This collapse occurs by reducing

the precision of the state variable to some finite fixed size. When this reduction is done, the Siegelmann networks (as intimated earlier) are no more powerful than Turing machines. Siegelmann proves this result on pages 33 to 57 of her monograph. It is also true that if the networks use Turing computable real weights the Siegelmann network computes no more than the Turing computable functions.

Note how our features show up as restrictions on these descriptions as well. (Strictly speaking the features are restrictions and properties of what the descriptions refer to, but of course the descriptions ought to match the referents in question.)

We can now use the above description to make a direct comparison between the two models. In particular, the state spaces are the most interesting point of comparison. Not only is the state space of the (universal) Turing machine finite and the Siegelmann network infinite, the latter is nondenumerably infinite. Describing the two models in this fashion shows how likely it is that the Turing machine is a less powerful model. Nevertheless, the dynamical systems approach does not allow us to assess the plausibility of the model **as** a model of computation without discussion of what the state spaces are supposed to be state spaces **of**. We know some of the details of the Siegelmann network's state space, but do not know how to understand (for instance) the subspace corresponding to the properties of a single pair of nodes.

One could, however, make some assumptions about the physical character of these nodes and their weights, leading to the selection of an appropriate state space. This again raises the question of idealizations. Suppose that we decide that that the electrical resistance of a material is to be the weight associated with a link between nodes. Classical electrodynamics elucidates resistance as an infinite precision real number, but there are two important facts to keep in mind. First of all, it is not clear

that the physical property **actually** varies continually. Until 1900 or so, physicists thought energy levels of atoms varied continually. Planck is credited with the insight that this is not so. Resistance “depends” on lower level properties, so it would not be at all surprising if it is not actually a continuous property. But, we do not need to even entertain speculative physics to raise the second of our points, one we have raised elsewhere. It is not merely that we have to think of how the property is to be used to “hold” values, but also how this property interacts dynamically with others, as computation is not a static occurrence, but a process. Here is where our previous worries about sensitivity and such matters come into play. Thus this lower level description would also need a discussion of activation functions and how they “work”, in order to clarify how they describe evolution of the node-level subspaces.

We have introduced our case study of a putative super-Turing model, and seen a specific case where it supposedly performs super-Turing computation. We next examine where this Siegelmann network is substantially different from the Turing model to see if the assumptions it makes render it plausible as a model of super-Turing computation.

Part 4: Criticism of the Siegelmann Model

Our criticism of the plausibility of the Siegelmann model shall again consider: finiteness, protocol, symbolic computation vs. computation by measurement, procedural form of computation, communicability, representations, generality, timelessness and hierarchy.

Finiteness (or lack thereof) cannot be a direct reason by itself to reject Siegelmann's analysis. Having seen that there are ways a certain sort of infinite object might be used in well specified ways to play a role in super-Turing computations, we now put this assumption through careful scrutiny. We examine both ways in which Siegelmann's networks are not finite: precision and sensitivity.

First we discuss precision. As noted, Siegelmann's networks require infinite precision "registers". This precision either needs to be present from the moment of a network's construction (how we are not told), or to somehow get built in as the network runs. The latter is what Siegelmann has called an "evolving" or "learning" system, and again a mechanism for this is not described. This is a grievous oversight, for it does not allow us to satisfactorily explore its plausibility.

Siegelmann is between a rock and a hard place on this issue. That is, if the network is finite at each stage, it is (more) believable; but it seems then to require a supertask to be able to do something super-Turing. On the other hand, if the networks are not so finite, it must somehow get the infinite precision (non-Turing computable) weight from **somewhere**, and furthermore it must be infinitely sensitive from the moment of construction. (See also our response to Davis in Part 5.)

Second, sensitivity: We have seen that Siegelmann's networks require infinite sensitivity in order to make use of infinitely large "registers" (interpreting them in the usual way as containing a number between 0 and 1). Siegelmann uses Cantor set style encoding (in order to minimize the difficulties in

recognition between two close register values; see part 3). However, this trades one problem for another. How do the Cantor sets in turn get represented? Clearly they cannot be encoded in binary or the problem they are intended to solve recurs; one cannot have appropriate Cantor set representations in the same fashion in base two. However, if they are to remain in base nine, the usual problem (familiar to electrical engineers) of unstable states applies. (Many electronic components only have two stable states.) While we do not intend to suggest that the Siegelmann proposal **must** be interpreted as one in electronics or electrical engineering, this is the easiest case to consider. After all, as seen in section three, Siegelmann seems to occasionally make use of this sort of terminology.

Protocol in the Siegelmann network is satisfactory in the way in which it performs I/O. However, the networks make essential use of the time variable, which renders it rather unlike the Turing model. This is simply a point of disanalogy, not a substantial disagreement. (See the discussion of the “timelessness” condition below for more.) However, in some applications (e.g. the circuit family simulation example we have discussed) the encoding of values in Cantor set style presents two problems. First, the interaction between those encodings and other encodings (e.g. of the input in a binary fashion) is an issue. Once number systems are being converted (particularly between an odd numbered base and an even one!) round off errors occur and a mechanism is needed to handle this. The “infinite” registers are needed as well, at the least to avoid the around off errors. Siegelmann does not discuss this matter. Second, and more critical, is her claim that the Cantor set style encoding of the structure of the circuit family allows us to save the need to (pp. 63):

> “distinguish between two very close numbers; thus the analog neurons can retrieve circuits efficiently using finite-precision operations only.”

We grant that the Cantor set encoding does what she claims **if** one can make the distinctions between different elements of the set {0,2,4,8} using finite precision. Here the problem of unstable

states mentioned above occurs and Siegelmann does not seem to be aware of the connection between this problem and representations. We do not stress this point too much as a somewhat analogous point can be made about the Turing machine. However, it is vital for Siegelmann's network that these values be taken as numbers in base 9. That is, in order to compute what she claims the network computes, Siegelmann needs to explain more how her representations are to work.

Also, her representations used in I/O make calculation of many, if not all, constant functions impossible by restricting output to finite bit strings. Without a discussion of a suitable protocol to convert (say) internal representations into pairs, or the like, most constants require infinite numbers of bits to specify. This oversight probably stems from Siegelmann's emphasis on languages (which only need one bit of output) rather than general functions.

We have no direct objection to calculations by measurement, as even to this day they are used (with the aide of ordinary computers!) in aeronautical engineering, for example. However, that said, Siegelmann's networks (sort of!) require a measurement of a kind that can never be accomplished - one with infinite precision. Since even a tiny error reduces the computational characteristics of the Siegelmann network to a sub-Turing[21] class, this implausibility, coupled with the unlikelihood of an infinitely sensitive sensor (see above in our discussion of the plausibility of finiteness) is an idealization of the sort we

[21] In fact, for the sort of error she discusses on pp. 136-137, those of "catastrophic nets", the networks fail to compute anything more than constant functions! We consider (in general terms) other forms of error - ones related to more "graceful" failure. (A catastrophic net is one containing only "forgetting neurons". A "forgetting neuron" is a node with a certain probability p of failure which results in the node receiving state = 0 and regular updates with probability 1 - p. The value p can be different for each node and makes no difference to the result Siegelmann mentions. She also mentions that correlating the reliabilities of the nodes (with suitable assumptions) makes no difference to the (lack of) computational power of the networks that use them.)

cannot countenance. If, however, we are willing to live with infinitely precise construction **with an error**, we have another possibility. Namely, it seems plausible that one could form a construction along the lines of the two sides of an 1-1-$\sqrt{2}$ isosceles triangle, and assert that the two sides were unit length, to within error e_1, and thus we produce a "real valued magnitude $\sqrt{2}$ to within an error e_2 (where e_2 can be calculated from e_1 in any of the theories of error one chooses). The problem here would be the generality condition (see below) that this violates. A $\sqrt{2}$ machine does not immediately permit construction of a $\sqrt{3}$ machine, not to mention any arbitrary real number (in either a strong or a weak sense). For example: how would one construct a $(\sqrt{\pi} + 3e - \ln 91)^{-\sqrt{3}}$ "machine"? (Nor do these "machines" lead to a strongly noncomputable real number in any straightforward way.) Our possible proposal is too ill-specified to consider further. It has been included simply for the sake of completeness.

Procedural computing is our next point of discussion. As noted, there is a clear sense in which Siegelmann's networks are sufficiently like the Turing model in this respect. We thus find no point of disagreement here and to the extent that they are different, we find the differences irrelevant.

The above is said with the caveats about our next concern, communicability, where the comparison is not as cut and dried. Siegelmann has greatly underspecified the operations of her network, and so its communicability is acceptable **only if** we restrict ourselves to the level in which she discusses the operations of her network. However, it is unclear that we should do so for the reasons discussed below under "representations". Let us examine the procedural version of the example network discussed above (part 3) and introduced on page 64 of Siegelmann's monograph. We reproduce it with line numbers (and minor typographical changes, e.g. we use := as in Pascal to denote variable assignment) below in order to facilitate discussion:

```
1) Procedure Retrieval (C,n)
2) Variables counter, y, z
3) Begin
4)   Counter := 0, y := 0, z := C
5)   While Counter < n
6)   Parbegin
7)        z := Gamma(z);
8)        if Xi(z) = 8 then increment Counter
9)   Parend
10)  While Xi(z) < 8
11)  Parbegin
12)       z := Gamma(z)
13)       y := (1/9)(y + Xi(z))
14)  Parend
15)  Return (y)
16) End
```

Xi() and Gamma() can be taken as “select left” and “shift left” operations respectively, as they have the consequence of performing these sorts of operation (familiar in binary to assembly language programmers) on numbers in base 9. We assume that we have access to the continuous versions of these operations Siegelmann gives (pp. 64-65), and focus on other issues of communicability raised by the procedure itself. On line 4, Retrieval() makes use of an unbounded “register”. As noted previously, Siegelmann makes no attempt to explain how this part works. Purely procedurally speaking, we are left with an operation that is not as decomposed as we would like and thus is unsatisfactory from the perspective of communication. Similarly, line 13 also makes use of an “infinite register”. It is true that these registers at any given run are only finite (albeit unbounded) in size. However, in order for this network to be used to simulate **any** circuit family, and hence recapture the generality condition, the register would have to be larger than any possible circuit, that is, actually infinite in size. It must be said, however, that this objection is not as damming as the sensitivity objections we have discussed.

However, there is an aspect of this procedure that is less debatable but is still sufficiently different from the Turing approach to render its communicability somewhat problematic. These are the **Parbegin** ... **Parend** blocks of lines 6-9 and 11-14. Siegelmann uses these to indicate that the operations within are to be understood as occurring simultaneously. Ignoring the relativistic problem that this raises (as it is a common assumption in electrical engineering), the question arises of whether “emulation by time sharing” would count as a suitable interpretation. We grant that it would, but note that it would require substantial additional set up and that it is unclear whether such belongs to the **procedure** or not. One would have to imagine some sort of “locking” protocol for concurrent programming and so on.

We move on to the question of representations. As noted above, there are several areas in which questions concerning representations interact strongly with other issues and we do not need to rework them here. However, a general question about representations remains to be asked. Since, as we have seen, we are not given any details about representation in Siegelmann’s model, we cannot evaluate her model fully and thus consider it to be underdeveloped for that reason. And yet this point is absolutely vital for evaluating the proposal’s plausibility as a whole. The criticality of this consideration can also be seen by a comparison to the Turing model: Turing argues for the plausibility of his representations. He contends that the subject of his analysis (human computation) can motivate restrictions on the representations, for instance, that they are bounded in size and hence, at least *prima facie*, cannot represent infinite objects like arbitrary infinite precision real numbers. As we have seen, however, there are (apparently) ways in which finite objects also have infinite properties, so it is not quite as easy as simply saying “finite parts therefore finite numbers represented (or “used”) so Turing computable functions only”. We just cannot be sure that this is necessarily so until Siegelmann offers more

detail about her proposal.

The next feature, generality, is one which we have already met in passing. Since there are general (or universal) networks, at least at the Turing computable level, Siegelmann's proposal is satisfactory on that account. It is unclear from her monograph, however, whether or not there is a **general** network that computes all the Siegelmann network functions. Siegelmann comes close to suggesting that there is not; she writes (emphasis added):

> "In particular, if exponential computation time is allowed, one can **specify a network for each binary language**, including non-computable ones."

However, the quoted remark is not conclusive. One needs to ask in this case "when" infinite precision would be needed. For instance, Chalmers (1996) has pointed out that at least one model of super-Turing analogue computation requires infinite precision "all at once." Siegelmann would thus have to show that to produce this general network one needs only finite precision at the beginning of operation. (Recall that most Siegelmann style networks do not begin with infinite precision in a certain respect.) Further she must show how the general network gains the appropriate precision and in a finite time, otherwise her proposal also involves a supertask. Note this even more complicated in the case of our "strong" understanding of super-Turing computability.

Ord and Kieu (2003) have claimed that there cannot be a general Siegelmann network as the input to a given network is always finite, and that to specify a network one needs infinite precision for every weight. Hence, they claim, one could not possibly feed in "weights to be simulated." This follows if one assumes that any network whatsoever is capable of being simulated. If one restricts oneself to networks that are not excessively complicated, then it might be possible to "start" with some infinite properties and "distribute them" appropriately in the simulation. Since, as Siegelmann points out, there is a universal Turing machine equivalent neural network this "excessively complicated" qualification we have given may be taken to be the structure

(number of nodes and links, etc.) of this network. Of course this does not show that one could do the “spreading around” we have suggested. The latter is implausible, since it would likely require more operations beyond the addition and multiplication that the Siegelmann networks perform. These considerations are a possible area of future research.

There are two possible responses that Siegelmann could make to the above objection. The first is to consider whether we could somehow encode the precision bound required into the “program” to be run on the network, and somehow have the network “construct” the appropriate “receptors”. (This procedure would be much as we decide when running Bochs, say, to emulate X megabytes of memory for the virtual machine.[22]) This proposal fails because one can always run the “emulation” software in itself. How would one do that on this proposal?[23] The second response is to simply deny the importance of the generality condition we have proposed. We construct specific-task neural networks and do not worry about generality. Giving up the concept of “programmability” is a major loss which would not be acceptable to many computer scientists and engineers. Since the Siegelmann networks are at least Turing-general, it seems plausible that one could at minimum have machines combine one specific sort of super-Turing machine and a universal Turing machine equivalent one. Generality is thus not totally lost.

“Timelessness” is our next area to debate. Unlike the Turing machine, the Siegelmann network requires the explicit use of a time variable. It gets used in several ways, some innocuous. However, there are several ways in which the time consideration of the Siegelmann network creates difficulty. First, we do not in general know the exact running time of a Siegelmann network. (We

[22] Bochs is an 80x86 system emulator for various platforms. In order to set up the settings of the emulator, the user must decide how much 80x86 memory she wishes to emulate.

[23] This objection is similar in flavour to the discussion of “well founded games” in Cameron (1999).

can work out its complexity class, e.g. that it belongs in class O(log n) for some parameter n, but the constants left aside by the bound in this case are critical.) It is true that the network does flag its output in analogous way to the Turing machine halt state with its data ready line. However, unlike the Turing machine which enters the “halt” state by the “programmer’s” decision, the Siegelmann model is not so programmed. We are not told how to “raise the flag” to indicate the value of the computation which is being outputted nor how the network recognizes that its output is ready. Once again, the Siegelmann network is underspecified.

Further, an infinite regress looms here. Since it is not Turing-computable when even a Turing machine will halt (if at all), determining the response time of a Siegelmann network is also not Turing-computable. If this parameter is to be determined at construction of the network, one needs to first create a super-Turing network to calculate the response time of the network one originally intended to create. But how does one determine the response time of the response-time calculating network? The alternative to the preceding considerations seems to be that the network itself is supposed to validate its own solution, that is, it must periodically check internally that the solution is ready. Again, it is not generally clear how exactly this is to work.

Also notice that the “learning networks” create another problem with regards to time. If the precision of an infinitely precise real number is not available at the beginning of the run of a network, and the precision increases uniformly in time, it will take an infinite amount of time for the network to become infinitely precise. As previously noted in passing, this entails immediately that the networks are actually Turing-equivalent for any finite period of time. Here let us note further that this puts Siegelmann’s model in an unfortunate dilemma much as the precision consideration proper above provokes. Once again, either the network is infinitely precise in finite time, in which case the Siegelmann network is implausible from the sensitivity considerations we have canvassed and from related concerns, or it

is only infinitely sensitive in infinite time, in which case using it to perform super-Turing computations would require a supertask.

The consideration of “hierarchy” is our penultimate point of criticism of the Siegelmann networks. As noted, the Siegelmann networks easily fit into the hierarchy if we allow exponential running time and exponential sized weights (see above on protocol). In other words, the class of functions computed by all the Siegelmann networks does strictly include the class of all functions computable by Turing machines. If there is no general Siegelmann network, however, this is cause for some worry regarding the hierarchy: we would want the generality to be preserved and extended in its next stage. There being no general Siegelmann network would make the universal Turing machine rather special: there is no universal finite automaton or universal push-down automaton either. This would be an interesting result if it could be shown, and one that would lend credence to the Turing machine model as a good model of computation for “practical applications.” (Of course, it could be that there is some other model with all the virtues of the Siegelmann model that is general for its class and everything below it. Needless to say, such would have to be further up the hierarchy.)

To summarize, we have discussed several areas where the Siegelmann network model of computation is underspecified or ill-motivated. There were problems with infinite sensitivity and infinite “registers” as well as issues with the protocol such as how the computation time parameter works and how to avoid an infinite regress.

We have concluded that the Siegelmann network is not a suitable model of computation for the critical reasons canvassed. Next we answer some responses to critics of our work.

Part 5: Responses to Our Critics

This section responds to possible criticisms of our work. It summarizes relevant points where this is helpful to answer the criticisms.

A first family of questions we must answer: “Why do we suppose that Siegelmann’s model is a model of **computation** at all? Doesn’t Siegelmann’s approach make everything with quantifiable properties compute? Doesn’t her approach commit one to “pancomputationalism”?” Our answer is perhaps the one that she would give, though she in fact does not provide it in her monograph. Our answer also presupposes that one considers a rational valued neural network a model of computation. Granted that assumption, the Siegelmann model can be looked at as an attempt to modify (minimally) an existing, Turing equivalent model of computation to produce one that is super-Turing. We thus must ask: what are these modifications? are these modifications plausible?

Further, not only does the network at some stage possess a property with an uncomputable magnitude (either strong or weak), but it is so constituted that the property in question is (one might say) **computationally efficacious**. By this we mean it plays a role in the entire process of the network, which we (suitably interpreting it via the I/O protocol) take as computing some function or other. By contrast, a lever like AB:

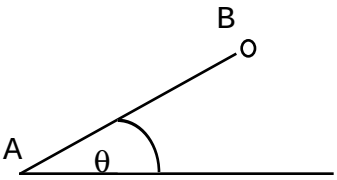


Figure 4: Lever.

does not “compute” an uncomputable function (in the Turing sense), even if θ happens to be Chaitin’s constant. It might be construed that this characterization is question begging. In other words, someone might rejoin: “How do we know that the neural networks of Siegelmann are computing and the lever is not: after all, one

could say it is computing the constant function x(t) = Chaitin's constant?" Our answer appeals to the use of each machine. The lever is not **used** to compute, whereas the networks are. In particular, there probably can be no "programmable lever" where there is a programmable (hence 'universal' in a certain class) network[24]. There is also no protocol for reading off the result in question. We are thus claiming that at least **some** computers are computers by convention. This approach is similar to the Turing set-up. We interpret **as** a computation what Turing's computor is doing mindlessly. The "protocol" in both cases gives a semantics for humans to interpret.

The second group of questions we answer presupposes our answer to the first. We respond to the question "Doesn't Siegelmann's model of computation change the subject? Turing analyzed human computability: why should we expect that another sort of understanding of computation coincides with the Turing one?" We respond by noting some of the similarities between the Siegelmann network and the Turing machine.

It is important to note that an important feature of the Turing analysis is that it centers around a computor capable of operating on "external" representations (configurations of symbols) of items to compute with; the latter which must be writeable and readable in some well specified fashion. Similarly, the Siegelmann network also makes use of a analogous (crucially: digital) external representation and simply uses a different sort of "state machine" to operate on it. In that sense there is also similarity between the two sorts of machines (recall part 1 and part 3 of this thesis). We argue, in other words, since both the Turing machine and the Siegelmann network can be seen as a state machine operating on an external representation of similar character, it does not matter what the internal mechanism is to be, at least from the perspective of "changing the subject" (That the internal

[24] Further, this stipulation does not rule out natural computers - if we permit ourselves to speak of functions of parts of organisms at all. But that is another story for another time.

mechanism of the Siegelmann network is underspecified is another matter, and the key issue we find problematic.)

We move on now to debate a new criticism.

Davis has written a paper (forthcoming) in which he discusses several super-Turing models, including Siegelmann's. We respond to Davis here as his apparent refutation of the model is much shorter than ours. In other words, we answer the question "Why did you spend 40 pages discussing something that can be refuted in a few paragraphs?" We would like to stress that Davis' article is **in conclusion** much in agreement with the present work and the criticisms below speak more to how he arrives at these same conclusions.

Davis asserts that in order to obtain the super-Turing power of the Siegelmann networks one has to provide them initially with non-Turing computable weights. In that sense the powers of the Siegelmann network are unsurprising (and question-begging: how does one obtain such a weight in the first place?). While we do agree with his conclusions, Davis does not (of course) mention the "weak" and "strong" versions of super-Turing computation that we have mentioned in our discussion. He also overlooks that the infinite precision of the Siegelmann networks is only required after infinite time. Siegelmann has proved (pp. 68-69) that only linear (in the time of calculation) precision is needed in her networks. Our problem with Siegelmann's approach is thus that the mechanism by which the precision of the registers increase is not described. This we feel is more charitable to Siegelmann's model than Davis grants. That said, Davis can rejoin that Siegelmann's model presupposes that the weights are actually infinitely precise to begin with and only get truncated "after being used"; thus we need to assume that the full precision weights are available at construction. However, all that is required by the model is that at time t + 1, the precision is increased from time t. The source of these "extra data" is one of our problems with Siegelmann's model.

In the context of discussing discovering a non-Turing computable physics which could provide the Siegelmann network (and other super-Turing computation proposals we need not discuss) with the needed non-Turing computable parameters, Davis' paper also quotes Scott concerning how we would recognize a "nonrecursive black box". We feel this quotation is also slightly mistaken: it proves too much. We agree that no finite amount of interaction with a black box could show that it performs non-Turing computable processes. However, no finite amount of observation could tell you that a black box contains a Turing machine. Any finite experimentation with input and output is consistent with the black box being a (perhaps very large) finite state automaton. This is not to say Scott and Davis are mistaken concerning the difficulty of determining that one has a super-Turing machine of some kind, but instead that it is important not to overstate this difficulty. Davis does, however, correctly stress at this point of the article our previous point about the nomological impossibility which makes Siegelmann's super-Turing proposal implausible. That is, he emphasizes how hard it would be to tell that one had a non-recursive "transparent box" (i.e. a black box with much of its workings well known).

We would also like to point out that our criticisms of Siegelmann apply even if science were to give us a means to obtain a non-Turing computable number. By contrast, Davis focuses more on the implausibility of obtaining such a number, albeit alluding on page page 10 of his paper briefly and ellipitically to the sensitivity consideration we discuss.

Finally, we also feel that Davis gives insufficient attention to Siegelmann's (albeit unsuccessful) attempts to integrate her model with existing models of computation in order to render it more plausible. Our task has been to see how these attempts at integration go. If we were simply to take Davis' refutation of what he calls the "myth of hypercomputation", we would not know exactly why Siegelmann's model would be even worth considering in

the first place. By providing a more detailed discussion we hope to have provided fruitful avenues for debate.

Another criticism with which we must deal concerns the status of Siegelmann's idealizations, after all, the Turing machine model itself makes idealizations concerning computing agents and their resources. The critic will ask: why should we not grant relevant idealizations to the Siegelmann network? Our response to this objection begins by noting that we **are** granting idealizations to the Siegelmann network: we are in fact giving it many of the same idealizations as the Turing machine. In particular, we are allowing it to calculate for arbitrary (but finite) periods of time without breaking or running down; we are allowing it to output any finite number of symbols and to take any finite number as input. We allow further that the network can be constructed out of any finite number of nodes (which parallels allowing the Turing machine any finite number of computationally relevant states[25]), etc. What we object to is the idealization of infinite precision. As we have said, it does not hold, even approximately. An analogy from another area of investigation will hopefully make this clearer. Consider the case of a two body system in Newtonian mechanics. We can work out the gravitational force (and hence acceleration) on each body to a given degree of accuracy. Assuming we know the masses, initial velocities, and mutual distance, etc. sufficiently precisely beyond a certain value, we know the qualitative behaviour of the system: we know that it will be gravitationally bound or that the bodies will escape, etc. This prediction will not be overturned by increasing the precision of our measurements of the distance or masses[26], assuming a certain threshold precision is obtained. By contrast, the Siegelmann

[25] N.B.: We are not saying that the Siegelmann networks are finite in **state.** That would be obviously false. But they have a finite number of computationally relevant **parts**, at least as described by Siegelmann herself.

[26] On the other hand, the assumption that the two bodies are completely isolated or only undergo gravitational forces (etc.) may turn out to be an inappropriate idealization.

networks would change the class of their behaviour (what class of functions they compute) were they actually constructed. In this sense the idealization is unrealistic (nomologically impossible[27], according to current knowledge of physics of noise, etc.: see part 4), and **always** so, suggesting that it is inappropriate as it is (unlike the Turing machine) never even approximately constructible. That said, we have of course no objection to someone who wishes to study the mathematical properties of how the networks are described. This **seems** to be Siegelmann's own primary intention, at least some of the time.

Contrarily, another critic may wonder why we are even considering the idealizations underlying the Siegelmann network. He would ask: "Aren't they so far fetched as to be not worth the time to debate?" Our answer to this relates to our answer to the previous critic. We agree with this critic that the Siegelmann network idealizations are implausible; it is difficult to imagine how humans could make use of infinite properties. But implausibility is by itself not a sufficient reason to reject them[28]. Once again, an analogy helps make our point. Routine, teaching-type, experiments in mechanics make use of the notion of a frictionless surface. In these cases, the idealization can be checked against reality. Similarly, with the Siegelmann network we can see (conceptually or via "thought experiments" of sorts, as Siegelmann

[27] It has sometimes been claimed that super-Turing computation is self-contradictory and thus logically impossible, so the arguments against it from nomological impossibility such (as ours) are hence weaker than necessary. We know of no successful proof to that effect; Ord and Kieu (2003) have shown the fallacious nature of many such arguments. Davis' article (discussed above) seems to agree with us when he suggests that one has to draw a distinction between the abstract theory of computation and the features of physical computers. We wish to avoid debating how to interpret the former, and since super-Turing proposals like Siegelmann's are (in part) an attempt at understanding the latter, we feel Davis' suggestion has merit in that respect.

[28] After all, it is perfectly conceivable that someone might find the Turing assumptions too implausible to be worth debating as well. Rejecting this approach there should lead us to reject the approach with the Siegelmann network.

has done) how these idealizations fail to match reality. If it were the case that these idealizations resulted in different sorts of “malfunction” or error, (say) ones that allowed us to do super-Turing computations with (say) X% reliability (X > 0), then perhaps the Siegelmann network would be a useful model. But since it appears that the networks would have 0% reliability (i.e. would never work to perform super-Turing computations), the network idealizations are not suitable. That is to say concerns over being “far fetched” are necessary but not sufficient for evaluating the merits of a scientific or technological proposal. Another way to understand our answer is as a reminder of precisely **how** far fetched the Siegelmann assumptions are.

We have responded to several criticisms about our work. Next, we summarize our findings and suggest a few areas of future research.

Part 6: Conclusions and Future Directions

Our comparison of the Turing and Siegelmann models and the critical examination of the features of the latter leads us to conclude that Siegelmann's approach, while promising in certain respects, is insufficiently developed to adequately and completely assess its merits. Nevertheless, to the extent that her model is developed we do not feel it serves satisfactorily as a model of super-Turing computation.

There are four possible extensions to the present work we would like to mention before closing. First, more direct analysis of the neural network operations would be possible if Siegelmann were to develop her model further. For example, she could elucidate better how sensitivity is to function or how the infinite "registers" are to be understood. Second, the framework of this thesis could be used to analyze other notions of putatively super-Turing computation. This would bring some unity to the debates on this subject in the literature. Third, the distinction between computing by symbol manipulation vs. computing by simulation or dynamics could be spelled out in further detail in order to provide a greater understanding of the work of Siegelmann (and perhaps others). Fourth, a more extensive exploration of the inter-simulation powers of the Siegelmann networks, prompted by the investigations of Ord (mentioned in part 3), would prove useful in studying their generality.

This ends the text of this thesis. We hope that the reader enjoyed the material presented and found that it helped to introduce and clarify some issues in the debates over super-Turing computation.

Works Cited

Balcázar, José Luis; Díaz, Josep and Gabarró, Joaquim. 1995. *Structural Complexity I* (2e). Berlin: Springer.

Boolos, George and Jeffrey, Richard. 1980. *Computability and Logic*. (2e) New York: Cambridge University Press.

Bunge, Mario. 1999. *Philosophy of Science* (2 vols.) New Brunswick: Transaction.

Cameron, Peter. 1999. *Sets, Logic and Categories.* Berlin: Springer Verlag.

Chalmers, David. 1996. *The Conscious Mind*. New York: Oxford University Press.

Church, Alonzo. 1937. "Review of Turing, 1936." Journal of Symbolic Logic. Vol. 2, pp. 42-3.

Cotogono, Paolo. 2003. "Hypercomputation and the Physical Church-Turing Thesis." Brit. J. Phil. Sci. 54, pp. 181-223.

Davis, Martin. 1964. *The Undecidable*. Hewlett: Raven Press.

Davis, Martin. Forthcoming. "The Myth of Hypercomputation." To appear in a Festschrift for Alan Turing. Heidelberg: Springer-Verlag.

Epstein, Richard and Carnielli, Walter. 2000. *Computability: Computable Functions, Logic, and the Foundations of Mathematics.* (2e) Belmont: Wadsworth/Thomson Learning.

Gandy, Robin. 1980. "Church's Thesis and Principles for Mechanisms." In Barwise, John, Keisler, H. Jerome., Kunen, Kenneth. (eds) 1980. *The Kleene Symposium*. Amsterdam: North-Holland.

Gandy, Robin. 1993. "On the Impossibility of Using Analogue Machines to Calculate Non-computable Functions." Unpublished Manuscript.

Gandy, Robin. 1995 (1987). "The Confluence of Ideas in 1936." in Herken, Rolf. ed., 1995. *The Universal Turing Machine: A Half Century Survey*. Wien: Springer-Verlag.

Hennessey, John and Patterson, David. 1998. *Computer Organization and Design: The Hardware/Software Interface* (2e). San Francisco: Morgan Kaufman.

Hofstadter, Douglas. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books.

Kozen, Dexter. 1997. *Automata and Computability*. New York: Springer.

Odifreddi, Piergiorgio. 1989. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*. Elsevier Science: Amsterdam.

Ord, Toby. 2002. *Hypercomputation: computing more than the Turing machine.* Available at: http://www.hypercomputation.net/cgi-bin/download/download.cgi?id=5

Ord, Toby and Kieu, Tien. 2003. “The Diagonal Method and Hypercomputation” Available at: http://www.arxiv.org/format/math.NT/0302183

Sieg, Wilfried. 1994. "Mechanical Procedures and Mathematical Experience". In A. George, ed. 1994. *Mathematics and Mind.* Oxford: Oxford University Press.

Sieg, Wilfried. 1997. “Step By Recursive Step: Church’s Analysis of Effective Calculability.” Bulletin of Symbol Logic, Vol. 3, No. 2. pp. 154-180.

Sieg, Wilfried. 2000. *Calculation by Man and Machine: Mathematical Presentation*. Technical Report No. CMU-PHIL-105.

Siegelmann, Hava. 1999. *Neural Networks and Analog Computation*: *Beyond the Turing Limit*. Boston: Birkhäuser.

Turing, Alan. 1936. "On Computable Numbers, With an Application to the Entscheidungsproblem." Reprinted in Davis 1964.

Turing, Alan. 1938. “Systems of Logic Based on Ordinals.” Reprinted in Davis 1964.