Available online at www.sciencedirect.com
ScienceDirect
Neural Networks
Neural Networks xx (xxxx) xxx–xxx
www.elsevier.com/locate/neunet

2008 Special Issue

# Some neural networks compute, others don't

Gualtiero Piccinini*

*Department of Philosophy, University of Missouri – St. Louis, St. Louis, MO 63121-4400, USA*

Received 6 August 2007; received in revised form 28 November 2007; accepted 11 December 2007

**Abstract**

I address whether neural networks perform computations in the sense of computability theory and computer science. I explicate and defend the following theses. (1) Many neural networks *compute*—they perform computations. (2) Some neural networks compute *in a classical way*. Ordinary digital computers, which are very large networks of logic gates, belong in this class of neural networks. (3) Other neural networks compute *in a non-classical way*. (4) Yet other neural networks *do not perform computations*. Brains may well fall into this last class.


*Keywords:* Connectionism; Neural network; Computation; Mechanism; Cognition; Brain

## 1. Introduction

In this paper, I draw and apply two distinctions: (i) between classical and non-classical computation and (ii) between connectionist computation and other connectionist processes.

Connectionist systems are sets of connected signal-processing units. Typically, they have units that receive inputs from the environment (input units), units that yield outputs to the environment (output units), and units that communicate only with other units in the system (hidden units). Each unit receives input signals and delivers output signals as a function of its input and current state. As a result of their units' activities and organization, connectionist systems turn the input received by their input units into the output produced by their output units.

A connectionist system may be either a concrete physical system or an abstract mathematical system. An *abstract* connectionist system may be used to model another system to some degree of approximation. The modeled system may be either a *concrete* connectionist system or something else; e.g., an industrial process.

Psychologists and neuroscientists use abstract connectionist systems to model cognitive and neural systems. They often propose their theories as alternatives to classical, or "symbolic", computational theories of cognition. According to classical theories, the brain is analogous to a digital computer (Fodor & Pylyshyn, 1988; Gallistel & Gibbon, 2002; Newell & Simon, 1976; Pinker, 1997; Rey, 1997). According to connectionist theories, the brain is a (collection of) connectionist system(s).

But given the standard way connectionist systems are defined, classical and connectionist theories are not necessarily in conflict. Nothing in the definition of 'connectionist system' prevents the brain, a concrete connectionist system par excellence, from being a (classical) digital computer.

'Connectionist system' is more or less synonymous with 'neural network'. Brains, of course, are neural networks. More precisely, there is overwhelming evidence that nervous systems carry out their information processing, cognitive, and control functions primarily in virtue of the activities of the neural networks they contain. Thus, it should be uncontroversial that brains are concrete connectionist systems and cognition is explained by connectionist processes. Both connectionists and classicists should agree on this much.

To bring out the contrast between the two theories, we need a more qualified statement: according to *paradigmatic* connectionist theories, the brain is a *non-classical* (collection of) connectionist system(s). This statement is informative only insofar as there is a nontrivial distinction between classical and non-classical systems. This is not the same as the distinction between systems that compute and systems that do not. We should be able to ask whether any, some, or all non-classical

* Tel.: +1 314 516 6160; fax: +1 314 516 4610.
*E-mail address:* piccininig@umsl.edu.

0893-6080/$ - see front matter © 2008 Published by Elsevier Ltd
doi:10.1016/j.neunet.2007.12.010

connectionist systems are computational. A clear answer to this question is needed to resolve a dispute about the nature of cognition.

Many mainstream connectionist theorists agree with classicists that brains perform computations and neural computations explain cognition (Feldman & Ballard, 1982; Hopfield, 1982; Marr & Poggio, 1976); Rumelhart & McClelland, 1986; Q1 (Bechtel & Abrahamsen, 2002; Churchland, 1989; Churchland & Sejnowski, 1992; Cummins & Schwarz, 1991; Eliasmith, 2003; Koch, 1999; O'Brien & Opie, 2006; Roth, 2005; Schwartz, 1988; Shagrir, 2006; Smolensky & Legendre, 2006). The claim that distinguishes such connectionist computationalists from classicists is that according to connectionist computationalism, non-classical connectionist (computing) systems are a better model of the brain than classical computing systems. In reply, some classicists argue that (non-classical) connectionist systems do not perform computations at all (Fodor, 1975; Gallistel & Gibbon, 2002; Pylyshyn, 1984). According to such classicists, only classical systems perform genuine computations. This does not bother a different group of connectionist theorists, who reject or downplay the claim that brains compute (Edelman, 1992; Freeman, 2001; Globus, 1992; Horgan & Tienson, 1996; Perkel, 1990).

Who is right? Do brains compute? Do (non-classical) connectionist systems compute? Which kind of system – classical or non-classical, computational or non-computational – is the best model for the brain?

Making progress on these debates – between classicists and connectionists as well as between computationalists and anti-computationalists – requires independently motivated distinctions between, on the one hand, systems that compute vs. systems that do not, and on the other hand, classical vs. non-classical systems. By applying such distinctions to connectionist systems, we can find out which connectionist systems, if any, do or do not perform computations, and which, if any, are classical or non-classical. Yet it has proven difficult to draw such distinctions in a satisfactory way.

The same problem may be framed in terms of theories of cognition. Is cognition explained by *non-classical, connectionist computations*? The answer depends on both what the brain does and where we draw two lines: (i) the line between connectionist computation and other kinds of connectionist processing and (ii) the line between classical computation and non-classical computation. Drawing these lines in a satisfactory way is a contribution to several projects: a satisfactory account of computation, a correct understanding of the relationship between classical and connectionist theories of cognition, and an improved understanding of cognition and the brain.

Unfortunately, different participants in these debates use 'computation' in different ways. I will base my discussion on computation as the subject matter of computability theory Q2 (aka recursion theory) and computer science. It is computation in that sense that is characterized by certain powerful mathematical theorems, comes with a special explanatory style (computational explanation), and inspired computational theories of cognition.

## 2. Do connectionist systems compute?

Accounts of the nature of computation have been hindered by the widespread view that computing requires executing programs. Several authors embrace such a view (Fodor, 1975; Pylyshyn, 1984).[1] Some authors endorse something even stronger: "To compute function $g$ is to execute a program that gives $o$ as its output on input $i$ just in case $g(i) = o$. Computing reduces to program execution" (Cummins, 1989, p. 91) (Roth, 2005). The weaker view – namely, that program execution is a necessary condition for genuine computing – is strong enough for our purposes. Such a view is plausible when we restrict our attention to at least *some* classical systems. The same view gives rise to paradoxical results when we consider connectionist systems.

The view that computation requires program execution leads to a dilemma: either connectionist systems execute programs or they do not compute. Different people have embraced different horns of this dilemma.

A computationalist who is opposed to (paradigmatic) connectionist theories might wish to deny that connectionist systems – or at least, paradigmatic examples of connectionist systems – perform computations. Here is something close to an outright denial: "so long as we view cognition as computing in any sense, we must view it as *computing over symbols*. No connectionist device, however complex, will do" (Pylyshyn, 1984, p. 74, italics original). A denial that connectionist systems compute is also behind the view that connectionism is not a truly computationalist framework, but rather, say, an associationist framework, as if the two were mutually exclusive (Gallistel & Gibbon, 2002).[2]

In light of the thesis that computing requires executing programs, rejecting the idea that connectionist systems perform computations may sound like a reasonable position. Unfortunately, this position does not fit with the observation that the input–output mappings produced by many paradigmatic connectionist systems may be characterized by the same formalisms employed by computability theorists to characterize classical computing systems.

It is difficult to deny that many paradigmatic examples of connectionist systems perform computations in the same sense in which Turing machines and digital computers do. The first neural network theorist to call his theory 'connectionist' appears to be Rosenblatt (1958).[3] Rosenblatt's *Perceptron*

[1] Another view that has hindered our understanding of computation is that computation requires representation (Cummins, 1983; Fodor, 1975; O'Brien & Opie, 2006; Pylyshyn, 1984; Shagrir, 2006). I have argued against it elsewhere (Piccinini, 2004a, 2007a, 2008), so I will set it aside.

[2] A related red herring, coming from an anti-computationalist perspective, is the claim that certain systems (cognitive systems, *some* connectionist systems) do not compute because they are *dynamical* (Port & van Gelder, 1995; van Gelder, 1995, 1997, 1998). As we shall see, Van Gelder is right that *some* connectionist systems do not compute. But not because they are dynamical—computing systems are dynamical too! The relevant question is, how should we draw the line between those dynamical systems that compute and those that do not? That is one topic of this paper.

[3] Rosenblatt was using 'connectionist' in its older sense, in which it applies, roughly, to the explanation of behavior in terms of connections (and changes



Please cite this article in press as: Piccinini, G. Some neural networks compute, others don't. Neural Networks (2008), doi:10.1016/j.neunet.2007.12.010

networks and subsequent extensions and refinements thereof can be studied by the same formalisms and techniques that are employed to study other paradigmatic computing systems; their computing power can be defined and evaluated by the same measures (e.g., Minsky and Papert (1969)).

Nowadays, the term 'connectionist system' is used to encompass more than Perceptrons. It usually encompasses all neural networks, as in my definition above. Rosenblatt himself was building on previous neural network models. He openly acknowledged that "the neuron model employed [by Rosenblatt] is a direct descendant of that originally proposed by McCulloch and Pitts" (Rosenblatt, 1962, p. 5).

It just so happens that, roughly speaking, a McCulloch and Pitts neuron is functionally equivalent to a logic gate. A collection of (connected, synchronous, ordered) logic gates with input and output lines and without recurrent connections is a Boolean circuit. Boolean circuits are the components of ordinary (classical) digital computers. Because of this, there is a straightforward sense in which digital computers are connectionist systems.

(Of course, digital computers are computationally more powerful than Boolean circuits, because they have recurrent connections between their components. In terms of their computation power, digital computers are finite state automata. Under the further idealization of taking their memory to be unbounded, digital computers are equivalent to universal Turing machines.)

That computers are just one (quite special) kind of connectionist system is underappreciated by those who debate the merits of connectionist versus classical computational theories of cognition (cf. Macdonald and Macdonald (1995)). In that debate, connectionist systems are often pitched without further qualifications against classical systems such as digital computers.[4] But unless Boolean circuits and collections thereof are excluded from consideration – which there is no principled reason for doing – denying that connectionist systems perform computations is tantamount to denying that computers compute.

Digital computers execute programs, and executing programs is an important aspect of the way they compute. Insofar as digital computers qualify as connectionist systems, at least some connectionist systems execute programs. But as we have seen, many non-classical connectionist systems, such as systems in Rosenblatt's tradition, perform computations too. Whether they compute by executing programs will be discussed in the next section.

In summary, the view that computation requires program execution generates a dilemma: either connectionist systems execute programs or they do not compute. We have now ruled out the second horn of the dilemma. To put the point bluntly, the computational properties of many connectionist systems are studied by one branch of computability and computational complexity theory among others.

## 3. Do connectionist systems execute programs?

Having ruled out one horn of our dilemma, we should consider the other. Perhaps (non-classical) connectionist systems execute programs, after all. Except for one detail: where are such programs? Did we somehow miss the memory components in which connectionist systems store their programs? Of course not. The view that connectionist systems execute programs requires weakening the notion of program execution to the point that, contrary to what you might have supposed, executing programs does not require writing, storing, or physically manipulating a concrete program in any way—at least in the sense in which programs are manipulated by ordinary computers.

Here is a recent example: "programs ... are just specifications of functional dependencies, and ... a system executes a program if the system preserves such dependencies in the course of its state changes" (Roth, 2005, p. 465).

This is not just an ad hoc proposal. Something like this weak notion of program execution predates the rise of neo-connectionism in the mid-1980s, which brought to the attention of philosophers the question of whether and in what sense connectionist systems perform computations. A similar notion was often employed during discussions of classical computing systems: "programs are not causes but abstract objects or play-by-play accounts" (Cummins, 1983, p. 34) see also Cummins (1977, 1989) and Cummins and Schwarz (1991).

This proposal may be put as follows: all that program execution requires is acting in accordance with a program. Acting in accordance with a program, in turn, may be explicated in at least two ways.

In the ordinary sense, 'acting in accordance with a program' means performing the operations specified by a program in the specified order. For instance, a program for multiplication might require the computation of partial products followed by their addition. A system that acts in accordance with such a program must generate intermediate results corresponding to partial products and then generate a final result corresponding to their sum.

This notion of acting in accordance with a program does not solve the present problem. Typical connectionist systems do not generate successive outputs corresponding to separate computational operations defined over their inputs. Rather, they turn their inputs into their outputs in one step, as it were. Thus, they do not act in accordance with a program in the ordinary sense.

More recently, Roth (2005) has proposed a novel construal of what I am calling 'acting in accordance with a program'. Under his construal, acting in accordance with a program does not require the temporal ordering of separate operations. Instead, it requires that the weights of a connectionist system

thereof) either between neurons or between stimuli and responses (Hebb, 1949; Thorndike, 1932, p. xix). Most neural network theorists are also connectionists in the older sense, though they do not have to be.

[4] A partial exception is Fodor and Pylyshyn (1988), who point out that connectionist systems can "implement" classical systems. Fodor and Pylyshyn focus most of their discussion on what they call "cognitive architecture". By contrast, I focus directly on what a system computes (if anything) from the standpoint of computability theory and how it does so. This leads me to a more inclusive notion of classical system than the one employed by Fodor and Pylyshyn (see below).



Please cite this article in press as: Piccinini, G. Some neural networks compute, others don't. Neural Networks (2008), doi:10.1016/j.neunet.2007.12.010

be defined in a way that (i) satisfies the logical relations between the program's operations and (ii) results in a system input/output equivalent to the program.

Consider again a connectionist system that performs multiplications. Under Roth's proposal, the system acts in accordance with a partial products program if and only if the system's connection weights are derived from a partial products program, even though the system produces its output in one step. By contrast, if the connection weights are derived from a different multiplication program – such as a program for multiplying by successive additions – then the system does not act in accordance with a partial products program (but rather, a successive additions program).

Roth's notion of acting in accordance with a program appears to provide an ingenious solution to the present conundrum. If we accept that executing a program amounts to acting in accordance with one, we can now use Roth's notion of acting in accordance with a program to conclude that at least those connectionist systems that act in accordance with a program in Roth's sense do execute programs. This suggestion is not as helpful as it seems.

First, how can we derive connection weights from programs? Roth refers to a technical report by Smolensky, Legendre, and Miyata, which was later expanded into a book (Smolensky & Legendre, 2006). Smolensky and colleagues describe a technique for defining the weights of certain connectionist systems from certain computer programs. The resulting connectionist systems are input/output equivalent to the programs.

This technique applies only to a special class of connectionist systems. What about the others? Roth's notion of program execution does not appear to apply to them. If we wish to say that they compute, as per Section 3, we need another account of connectionist computation.

It also remains to be seen whether, after the connection weights are defined with Smolensky et al. 's technique, the best thing to say is that the connectionist system executes the program (as Roth puts it). It seems more appropriate to say that the connectionist system computes the function defined by the program without executing the program. The latter, sensibly enough, is Smolensky and Legendre's view.[5]

Finally, suppose you want to say that a connectionist system executes a program based on Roth's proposal. Which programming language is it written in? In ordinary computers, this question has a well-defined answer. But it appears that the connectionist systems described by Roth act in accordance with any program, written in any programming language, which specifies the relevant operations in the relevant order. This makes it difficult to pick one of these programs as the one the system is executing. Or does it execute them all?

We need not press these questions further. For even if we grant Roth his notion of acting in accordance with a program, we should resist his conclusion that the connectionist systems he describes *execute* programs in the relevant sense. If acting in accordance with a program is sufficient for executing programs, then at least the systems defined using Smolensky et al. 's technique may well execute programs. This might work as a new meaning of 'program execution'. But in the sense of 'program execution' employed in computer science, acting in accordance with a program is hardly sufficient for program execution.

In computer science, a program is a list of instructions implemented by a concrete string of digits; 'executing a program' means *responding* to each instruction by performing the relevant operation on the relevant data. As I am using the term, digits are discrete physical entities or stable states that can be stored in memory components and transmitted from component to component.

The modern notion of program execution originates as a way to characterize a special property of modern computers (and some other machines, as we shall see in the next section). Computers do much more than act in accordance with a program. They can act in accordance with *any number* of programs. The reason for that is that they can store physical instantiations of the programs, and it is those physical instantiations that, together with input data, drive computer processes.

Programs in this sense are much more than "specifications of functional dependencies". They are strings of digits that computers can respond to and manipulate in the same way that they respond to and manipulate data. Programs can be written, tested, debugged, downloaded, installed, and, of course, executed. It is the execution of programs in *this* sense that explains the behavior of ordinary computers. This is also the notion of program execution that contributed to inspire the analogy between minds and computers, on which many computational theories of cognition are based.

If we want to honestly assess whether connectionist systems execute programs and use this assessment to compare different computational theories of cognition, we should use the standard notion of program execution that is used in computer science. And in this sense of 'program execution', paradigmatic connectionist systems do not execute programs.

## 4. A mechanistic account of computation

Our hands are tied. We have found that connectionist systems perform computations even though they do not execute programs. We must conclude that computation does *not* require program execution. This is not a bad result. Several independent considerations point to the same conclusion.

First, nothing in the notion of computation studied by computer scientists and computability theorists entails that computation must be performed by executing a program. The original mathematical notion of computation is that of a process that accords with the steps of an algorithm or effective procedure—there is no further requirement that the algorithm be implemented by a program or that the program be executed.

[5] "The input–output mapping of such a function [i.e., the function computed by one of Smolensky et al. 's networks] *could* be specified by a sequential symbolic program, but such a program would not describe how the networks *actually computes* the function" (Smolensky & Legendre, 2006, p. 72, italics original).



Please cite this article in press as: Piccinini, G. Some neural networks compute, others don't. Neural Networks (2008), doi:10.1016/j.neunet.2007.12.010

Second, the notion of program-controlled machines, of which program-controlled computers are a species, did not even originate in the field of computing. It originated in the textile industry. The first technology for programming machines – punched cards and the mechanisms for executing them – was developed in the 18th Century to control the weaving of patterns in mechanical looms. In 1801, Joseph Marie Jacquard developed an improved version of this technology for his Jacquard Loom. Only later did Charles Babbage borrow Jacquard's idea to control his Analytical Engine. The Analytical Engine, which Babbage never managed to construct, was the first (hypothetical) program-controlled computer.

Third, in computer science there is a useful distinction between computing systems that execute programs and computing systems that do not. Computation, including mechanical computation, is much older than program-controlled computers. Ordinary calculators and many other computing devices do not execute programs in the sense in which full-blown computers do. Quite aside from paradigmatic connectionist systems, there are plenty of systems that compute without executing programs.

Finally, the standard explanation of how computers execute programs appeals to components that compute without executing programs. Computers execute programs in virtue of possessing the relevant kind of processors. Such processors contain a control and a datapath. When an instruction reaches the processor, the control and the datapath perform two different functions. The control receives one part of the instruction—the part that encodes a command (e.g., sum, multiply, etc.). Then, the control determines which operation must be performed on the data and sends an appropriate signal to the datapath. The datapath receives the command from the control plus the remaining part of each instruction—the part that encodes the data. After that, the datapath performs the relevant operation on the data and sends the result out.

The control and the datapath are computing components—in the language of computability theory, they are finite state automata. But neither of them alone executes any program—it is only their organized effort, in cooperation with memory components, which allows the computer to execute programs. Thus, in ordinary digital computers, program execution itself requires components that compute without executing programs.

For all these reasons, we need an account of computation that accommodates both computing systems that execute programs and computing systems that do not. In recent years, I have articulated and defended such an account (Piccinini, 2007a, 2008, in press a, in press b).

In my terminology, a mechanism is a system of organized components, each of which has functions to perform (cf. Craver (2007) and Wimsatt (2002)). When appropriate components and their functions are appropriately organized and functioning properly, their combined activities constitute the capacities of the mechanism. Conversely, when we look for an explanation of the capacities of a mechanism, we decompose the mechanism into its components and look for their functions and organization. The result is a mechanistic explanation of the mechanism's capacities.

This notion of mechanism is familiar to biologists and engineers. For example, biologists explain physiological capacities (digestion, respiration, etc.) in terms of the functions performed by systems of organized components (the digestive system, the respiratory system, etc.).

Computing systems are a special class of mechanisms. They are distinguished from other mechanisms by the peculiar capacity they have. Their capacity is to generate output strings of digits from input strings of digits and (possibly) internal states in accordance with a general rule that applies to all relevant strings and depends on the input strings and (possibly) internal states for its application. To make sense of this, I need to explicate 'digit' and 'string'.

Abstract (classical) computations are sequences of strings of letters from a finite alphabet. Within any such sequence of strings, each string derives from its predecessor in virtue of some instruction from a list. Instructions are defined over the letters (plus, possibly, internal states of an abstract mechanism). The list of instructions is often called 'program'. For each program, a rule (e.g., *addition* or *multiplication*) defines the relation holding between the outputs generated in accordance with the program and their respective inputs. The rule defines the computations that satisfy the program. (If there is no rule that can be defined independently of the program, the program itself can be the rule.)

Abstract letters can be realized by concrete entities, which I call 'digits'. A digit in this sense need not represent a number. It may represent something other than a number or even nothing at all.

A digit is a component or stable state of a component of a mechanism. It can enter the mechanism, be transmitted to and processed by the mechanism's components, and exit the mechanism. Most importantly, computing mechanisms are functionally organized so as to reliably distinguish digit tokens of different types and manipulate them according to their type. In order to do that, in any computing mechanism there may be only finitely many digit types (typically there are two: '0' and '1').

The simplest computing components are logic gates, which manipulate one or two digits during any functionally relevant time interval. They take one or two digits as input and yield one digit as output. Their output stands in a definite logical relation to their inputs; such a logical relation is the rule that defines the function they compute.

Digits may be ordered into strings, which are nothing but sequences of digits. Each digit in a string has a unique predecessor and successor, unless it is the first in the string (lacking a predecessor) or last (lacking a successor). What constitutes the ordering of digits into strings varies from mechanism to mechanism: it may be the spatial arrangement of components bearing the digits, the temporal relation between digits, a combination of both, or some more complex functional relation.

In order to manipulate strings of digits, logic gates may be composed (organized) into circuits. In order to manipulate strings according to their ordering, such circuits must possess specific properties. The circuits must be configured so as to



Please cite this article in press as: Piccinini, G. Some neural networks compute, others don't. Neural Networks (2008), doi:10.1016/j.neunet.2007.12.010

perform the relevant operations on the relevant digits within a string. Each pair of digits in a string may require a different treatment from the other pairs of digits, so the circuits' elements must be organized to treat each pair in the specific way it requires. Furthermore, the circuits' elements are synchronized by a periodic signal (clock), so that all digits within a string are processed at the right time and the activities of different components do not interfere with each other.

When a mechanism can reliably distinguish between digits of different types and manipulate them according to their type and their position within a string of digits, strings of digits can be labeled by strings of letters. Thus, the same rules that define abstract computations can be used to characterize the processes of such a mechanism.

Concrete computation, then, is the process performed by a mechanism that manipulates strings of digits in accordance with rules defined over the digits (plus, possibly, internal states). All paradigmatic computing systems, such as Turing machines, finite state automata, and digital computers, perform computations in this sense.

In the case of classical computing mechanisms, the process of string manipulation can be divided into steps, which occur during functionally well-defined time intervals. During each time interval, the mechanism produces a string of digits corresponding to a string of letters within an abstract classical computation. Thus, the mechanism's processes are in accordance with the program that defines a classical computation.

Classical computing mechanisms are computationally decomposable in the following sense. As a whole, a classical computing mechanism performs computations in accordance with a general rule. For instance, it computes the square root of its input. The performance of such a computation may be explained in terms of the mechanisms' components and the computations they perform. For instance, a square root computation is explained by the combined action of a memory component storing a square root program and a processor executing the program.

This explanatory strategy can be iterated. The processor's capacity to execute programs is explained by the computations performed by the control and datapath that constitute it and the way they are wired together (as mentioned above). The computations performed by the control and datapath are explained by the computations performed by the circuits that constitute them and the way those circuits are organized. Finally, the computations performed by circuits are explained by the operations performed by the logic gates that constitute the circuits and the way those gates are connected. Since the operations performed by logic gates are computationally primitive, computational decomposition stops with them. The capacities of logic gates can still be mechanistically explained in terms of the organized functions performed by their components (resistors, capacitors, or whatever). But such mechanistic explanations no longer appeal to *computations* performed by their components.

To recapitulate where we have gotten so far, the idea that executing programs is necessary for computing leads to the dilemma that either connectionist systems execute programs or they do not compute. On the first horn, we must disconnect the notion of program execution from the special type of mechanism that gave rise to the idea of (computational) program execution in the first place. On the second horn, we must say that systems that compute by the lights of computability theory do not, in fact, compute. Fortunately, we need not impale ourselves on either horn. It is preferable to embrace a broader, mechanistic view of computation, according to which computation does not require program execution. This is a good result, with plenty of independent motivation. After abandoning the mistaken notion that computation requires program execution, we can gain a better understanding of connectionist computation.

## 5. Connectionist computation

Many non-classical connectionist systems perform computations in the same sense as classical ones—they manipulate strings of digits in accordance with an appropriate rule.

When the properties of paradigmatic connectionist systems are characterized computationally, the inputs to their input units and the outputs from their output units are relevant only when they are in one of a finite number of (equivalence classes of) states. Hence, these connectionist systems' inputs and outputs – though not necessarily the inputs and outputs of their hidden units – are digits in the present sense.

Even when the values of a connectionist system's inputs or outputs can vary continuously, in many cases there are discontinuities in the way the different values are classified. Typically, only values in certain neighborhoods (e.g., two neighborhoods labeled '0' and '1') are counted as determining what the whole system's input or output is; there may be gaps between neighborhoods; and all values within each neighborhood are counted as functionally equivalent. Hence, the systems' inputs and outputs still constitute digits.

Furthermore, depending on the kind of system, either the spatial ordering of the input units or the temporal sequence of the input units' inputs constitute a string of digits, and either the spatial ordering of the output units or the temporal sequence of the output units' outputs constitute a string of digits. Such strings are the entities in terms of which the computation power of the system is defined.

Finally, the way the system's outputs are (or are supposed to be) functionally related to its inputs (plus, perhaps, its internal states) constitute a rule defined over the digits. Such a rule defines the computation performed (or approximated) by the system.

The first authors who defined a class of neural networks and ascribed computations to them were Warren McCulloch and Walter Pitts (McCulloch & Pitts, 1943). The above account applies straightforwardly to McCulloch–Pitts networks. Each unit receives and returns only two values, typically labeled '0' and '1'. These are the digits. Each unit returns an output that stands in a definite logical relation to its inputs (e.g., AND, OR). The units have a discrete dynamics and are synchronous, so that each unit processes its inputs and returns its output during



Please cite this article in press as: Piccinini, G. Some neural networks compute, others don't. Neural Networks (2008), doi:10.1016/j.neunet.2007.12.010

the same time intervals. Input and output units are arranged in well-defined layers; units in each layer can be ordered from first to last. So during any time interval, the inputs going into the input units and the outputs coming from the output units constitute well-defined strings of digits. Finally, the structure of the network does not change over time.

Rosenblatt's Perceptrons (of any number of layers) are nothing but McCulloch–Pitts networks in which the last condition is dropped. Instead of having a fixed structure, Perceptrons can be trained by varying their connection weights during a special period of time, called the 'training period'. The same is true of *Adalines*, another early and influential type of neural network (Widrow & Hoff, 1960).

Being trainable makes no difference to whether a system is computational. After a Perceptron or Adaline is trained, the system maps its input string of digits onto its output string of digits according to a fixed rule. An elementary example of such a rule is EXCLUSIVE OR (either $x$ or $y$, but not both). EXCLUSIVE OR is notorious in connectionist circles because its computation requires Perceptrons or Adalines with hidden units; until the 1970s, some authors doubted that multi-layer networks could be trained reliably (Minsky & Papert, 1969). The discovery of training methods for such networks (Werbos, 1974) proved such doubts to be unfounded.

Thus, Perceptrons and Adalines are nothing but trainable McCulloch–Pitts networks (cf. Cowan (1990)). After they are trained, they compute the same function computed by the corresponding McCulloch–Pitts network. (Of course, trainable connectionist systems may be trained more than once; after each training period, they may compute a different function.) Other classes of connectionist systems can be defined by relaxing more of McCulloch and Pitts's assumptions.

We may allow units to take any real-valued quantity as input, internal state, or output (instead of only finitely many values, such as '0' and '1'), and we may let the input–output function of the processing units be nonlinear. To be physically realistic, the range of values that units can take and process must be restricted to a finite interval. And in many paradigmatic applications, the inputs to and outputs from the network are defined so that all values within certain intervals are taken to be functionally equivalent. For instance, the real-valued quantities may be restricted to the interval [0, 1]; all values near 0 may be labeled '0'; all values near 1 may be labeled '1'. Thus, all values within the right neighborhoods count as digits of the same type, which may be ordered into strings according to the ordering of the network's input and output units. When such a convention is in place, the power of such networks can be defined in terms of computability theory.

We may also let the units be asynchronous, let the network's dynamics be continuous in time, or both. When this is done, there may no longer be any way to individuate input and output strings of digits, because there may no longer be a well-defined way of classifying inputs or outputs into discrete equivalence classes (digits) and ordering different digits into strings. Unless, that is, the network receives its inputs all at once and there are conventions for grouping outputs into equivalence classes and ordering them into strings. A standard way of doing so is to consider network dynamics that stabilize on a stable state, from which networks produce constant outputs. Under such circumstances, inputs and outputs still constitute strings of digits. Unsurprisingly, these conditions are in place whenever the power of such networks is analyzed in terms of classical computability theory. I will discuss networks that fail to satisfy these conditions in the next section.

Here are some basic results about the computation power of those connectionist systems whose inputs and outputs can be characterized as strings of digits. Feedforward networks with finitely many processing units are computationally equivalent to Boolean circuits with finitely many logic gates. Recurrent networks with finitely many units are computationally equivalent to finite state automata. Networks with an unbounded number of units (or equivalently, with unbounded memories) are computationally equivalent to Turing machines (see Šíma and Orponen (2003, for a recent review)). There are also notional networks whose computation power exceeds that of Turing machines (Siegelmann, 1999), but there is no evidence that they can be physically implemented.

One moral of the above discussion is that connectionist computation can be either classical or non-classical. For instance, McCulloch–Pitts nets are perfectly classical. They are so classical that digital computers are essentially made out of them. Now it is time to say more explicitly what is peculiar about non-classical connectionist computation.

Unlike classical computing systems, (paradigmatic) connectionist systems may be trained. Training is the adjustment of the connections between the units to fit a desired input–output rule.[6] Thus, the dynamics of connectionist systems may change over time independently of which inputs they get. In other words, a connectionist system may evolve so as to yield different outputs in response to the same inputs at different times. Understanding the causal mechanism behind the system's input–output rule and its evolution over time requires understanding the dynamical relations between the system's units and the way they can change.

Besides trainability, there is a deeper difference between classical and non-classical systems. Unlike the former, the latter do not act in accordance with a step-by-step procedure, or program. Classical computation proceeds by performing one operation at a time, where an operation is a change of one string of digits into another.[7] Since strings are discrete entities, changing one string into another is a discrete operation.

[6] This is not to say that classical systems cannot learn. But (classical) machine learning, unlike the training of a connectionist system, does not change the connections between the logic gates in the computer hardware. There are also computer processors designed to adapt their circuitry to their task, though not by a training process like those used for connectionist systems.

[7] Because of this, classical computers are often characterized as 'serial'. Most classical computers are serial in the sense that they have only one central processor, which performs one primitive operation at a time. But in standard computers, each primitive operation performed by the processor requires the simultaneous action of many logic gates. In this sense, ordinary computer processors are parallel. Since this is also the sense in which paradigmatic connectionist systems are parallel, it turns out that (most) classical and connectionist systems are parallel in the same sense. There are other senses of 'parallelism' that apply solely to digital computers (cf. Piccinini (in press b)).



Please cite this article in press as: Piccinini, G. Some neural networks compute, others don't. Neural Networks (2008), doi:10.1016/j.neunet.2007.12.010

Thus, a classical computational dynamics – involving many transformations of one string into another – is by definition discrete. By contrast, many connectionist systems proceed by letting their units affect each other's activation values according to dynamical relations that vary in continuous time. A connectionist system's dynamics is the analogue of an ordinary computer's digital logic. Whereas digital logic determines discrete dynamics, the typical dynamic of a connectionist system is continuous.

Because of this, many connectionist computing systems are not computationally decomposable like classical ones. That is, such connectionist systems have computational capacities, but these capacities cannot be explained in terms of simpler computations performed by their components together with the way their components are organized. The reason is that in the case of a continuous dynamics, there is no well-defined way of breaking down the process by which a system generates its outputs from its inputs into intermediate computational steps, equivalent to the transformation of one input string into an output string.

It does not follow, however, that there is no mechanistic explanation of connectionist processes. The activities of connectionist systems are still constituted by the activities of their components and the way the components are organized. It's just that to explain how typical connectionist systems work, we need mathematical tools different from digital logic and computability theory—tools like cluster analysis, nonlinear dynamics, and statistical mechanics.

In brief, there are two dimensions along which classical and non-classical computing systems differ. First, they may or may not be trainable. Second, they may compute either by acting in accordance with a program (or algorithm) or by following a dynamics that cannot be decomposed into intermediate computational steps. Purely classical systems (e.g., digital computers, McCulloch–Pitts nets) have the first two characteristics (fixed structure and acting in accordance with a program). Many non-classical systems have the second two characteristics (trainability and continuous dynamics), but there are also systems that are non-classical only in one way (e.g., the original Perceptrons and Adalines, which are trainable but have discrete dynamics).

If this is right, why do many connectionists speak of connectionist algorithms, as they often do, even when talking about systems with continuous dynamics? The term 'connectionist algorithm' is used in several ways. The present account allows us to distinguish and elucidate all of them.

Sometimes, 'connectionist algorithm' is used for the procedure that trains a connectionist system, that is, the procedure for adjusting the dynamical relations between units. This is typically an algorithm in the classical sense—a list of instructions for manipulating strings of digits.[8] Training algorithms, of course, are not what actually drives any given connectionist computation: connectionist computations are driven by the state of and dynamical relations between a system's units. Thus, from the fact that a system is trained using an algorithm, it does not follow that the system itself computes by following an algorithm.

But 'connectionist algorithm' is also often used for the process by which a connectionist system produces its output. Based on what we have seen, typically this is not an algorithm in the classical sense. Many connectionist systems do not have the kind of discrete dynamics defined over strings of digits that can be accurately described by algorithms in the classical sense. Therefore, this second usage of 'connectionist algorithm' is equivocal. But given how common it has become, it may be considered a new sense of the term 'algorithm'—not to be confused with the classical sense.

Finally, the process by which a series of connectionist systems, each one feeding into the next, generates a final output through intermediate connectionist processes is sometimes called a 'connectionist algorithm'. This may well be an algorithm in the classical sense – or something close – provided that each intermediate process is definable as a transformation of strings of digits.

## 6. Other connectionist processes

In the previous two sections, I gave an account of what it takes for a connectionist system to perform computations in the sense of computability theory: its inputs and outputs must be strings of digits, and its input–output relationship must accord with a rule defined over the digits (and possibly internal states). Implicitly, such an account also specifies conditions under which a connectionist system does *not* compute—all it takes is for a system to fail to satisfy the conditions under which a system computes. This happens when its input–output relationship does not accord with an appropriate rule, its inputs and outputs are not strings of digits, or both.

The best example of a system that does not act in accordance with a rule at all is a system that generates random outputs. Since its outputs are random, there is no rule that relates the outputs to the inputs. There are also systems that act in accordance with a rule, but the rule is not defined over digits. An example is a mechanical loom, which performs the same actions regardless of what its inputs are like (or even whether it receives any inputs at all). Neither random systems nor mechanical looms are computing systems, because they do not act in accordance with the right kind of rule.

It is certainly possible to define connectionist systems with random outputs or outputs that are unrelated to properties of the inputs. They would be atypical. Typical connectionist systems respond to the properties of their inputs in a regular way. But there are many connectionist systems whose inputs and outputs, for one reason or another, are not strings of digits.

[8] Well, at the very least, implementations of training algorithms in digital computers are algorithms in the classical sense. Some training "algorithms", such as backpropagation, are actually defined over continuous values rather than anything that is reducible to strings. Strictly speaking, digital implementations are only approximations of these "algorithms". Thus, strictly speaking, "algorithms" like backpropagation are not algorithms in the classical sense, though they are algorithms in the sense of the next paragraph. I am indebted to Corey Maley on this point.



Please cite this article in press as: Piccinini, G. Some neural networks compute, others don't. Neural Networks (2008), doi:10.1016/j.neunet.2007.12.010

For instance, some connectionist systems manipulate continuous variables (Chen & Chen, 1993). Understanding the processing of continuous variables requires specialized mathematical tools, which differ from the discrete mathematics normally employed by computability theorists and computer scientists. At the very least, there is no straightforward way to assign computation power, in the standard sense defined over effectively denumerable domains, to systems that manipulate continuous variables. Because of this, it seems appropriate to conclude that connectionist systems that process continuous variables do something other than computing, at least in the sense of 'computing' employed in computer science and computability theory. This is in line with the present account of computation: since continuous variables are not strings of digits, their manipulation does not count as computation.[9]

More generally, anything that violates the conditions sketched in the previous section, conditions which allow a network's inputs and outputs to be characterized as strings of digits, does not perform computations. This point has special relevance to neuroscience and psychology, where the question of whether the brain computes is central to many debates.

In the case of *artificial* connectionist systems, it is somewhat open to stipulation which of their properties count as inputs and outputs. We may consider only networks with properties that are conducive to classifying their inputs and outputs as strings of digits. We may choose to send inputs to our network all at once at well-defined times, consider only network dynamics that stabilize on stable states, group inputs and outputs into finitely many equivalence classes, and impose a well-defined order on the input and output units, so as to characterize inputs and outputs as strings of digits.

When we switch to studying *natural* systems such as brains, we must not import such stipulations into our theories without justification. Brain theories must be based on empirical evidence about which properties of neural activity are functionally significant, not on which putative properties can be conveniently characterized in terms of computability theory.

Above, I pointed out that Rosenblatt, who first characterized his neural network theory as connectionist, was building on McCulloch and Pitts's work. McCulloch and Pitts, in turn, were building on work by the mathematical biophysics group led by Nicholas Rashevsky (cf. Piccinini (2004b)).

Rashevsky and his collaborators, young Walter Pitts among them, studied mathematically the properties of certain idealized neural systems. They defined and studied neural systems using differential and integral equations that represented, among other variables, the frequency of neuronal pulses travelling through idealized neuronal fibers. They analyzed systems with different structural and functional properties, characterizing their behavior and offering explanations of phenomena such as discrimination, perception, reflexes, learning, and thinking (Householder & Landahl, 1945; Rashevsky, 1938, 1940). From a modern perspective, they should count as pioneers of connectionism. Yet they never claimed that their networks compute.

The present account of computation makes sense of this fact. The mathematical tools Rashevsky and his collaborators used to analyze their networks did not include logic or computability theory. Furthermore, the main variable in terms of which they characterized neuronal activity was the frequency of neuronal pulses in continuous time—roughly corresponding to what today is called 'firing rate'. Since Rashevsky's time, theoretical neuroscientists have learned to study firing rates using more sophisticated mathematical tools. Now as then, firing rates appear to be the most important neurophysiological variable. Now, as then, theoretical neuroscientists who study spike trains and firing rates (Dayan & Abbott, 2001) make no significant use of logic or computability theory. This is not an accident: as in the case of continuous variables, there does not seem to be any clear or theoretically useful way to characterize firing rates varying in continuous time as strings of digits.[10] Therefore, it seems appropriate to deny that the resulting neural networks compute.

If it turns out that neural processes are not (or "do not implement") computations, some theories of cognition will have to change. Many theorists assume that cognition is computation (in the sense here explicated). This includes classicists, whose view is that cognition is classical computation, as well as many connectionists, whose view is that cognition is non-classical computation. Their theories postulate some computation or other to explain cognitive phenomena, but they usually gloss over the issue of how such computations are implemented in the brain. This may be ok as long as neural processes are (or "implement") computations: *somehow*, neural computations must implement the computations postulated by cognitive theorists.

But if neural processes are (or "implement") something other than computations, cognition must be explained by that something else—theorists of cognition must learn how nervous systems work and formulate theories in terms that can be implemented by actual neural processes. There are those who have been doing this all along, and there are those who are moving in that direction. Perhaps it is time for the rest of the community to join them.

## 7. Conclusion

Nowadays, many authors use 'computation' loosely. They use it for any processing of signals, for "information processing," for whatever the brain does, or even more broadly. In these loose senses, all connectionist systems – brains included – compute. But the resulting theses are rather uninformative: they do not lead to applying the mathematical theory of computation to analyze connectionist systems, their power, and their limitations. They should not be confused with the interesting proposal first made by McCulloch and Pitts.

[9] The same point applies to so-called 'analog computers' (in the sense of Pour-El (1974)), which manipulate continuous variables too. For a more detailed discussion of analog computers and why they do not compute in the same sense as digital computers, see Piccinini (in press b).

[10] I lack the space to defend this claim. For a slightly more extended (though still brief) discussion, see Piccinini (2007b).

When McCulloch and Pitts claimed that their networks compute, their claim was strong and informative. They claimed that what we now call 'computability theory' would allow us to analyze and evaluate the power of neural networks. This is the kind of claim that I have explicated in this article.

If the above considerations are on the right track, several conclusions can be drawn. First, many connectionist systems do *compute*: they manipulate strings of digits in accordance with a rule defined over the inputs. This allows us to apply the mathematical theory of computation to them and evaluate their computation power. Second, some connectionist systems (such as McCulloch–Pitts networks) compute *in a classical way*: they compute by operating in accordance with a program (or algorithm) for generating successive strings of digits, one step at a time. Digital computers, which are very large networks of logic gates, belong in this class of connectionist systems. Third, other connectionist systems compute *in a non-classical way*: they are trainable, turn their input into their output in virtue of their continuous dynamics (which cannot be broken down into intermediate computational steps), or both. Fourth, yet other connectionist systems (such as Rashevsky's networks) *do not perform computations*: their inputs and outputs are not strings of digits. Brains may well fall into this last class.

## Uncited references

Q7 Anderson & Hinton, 1981.

## Acknowledgments

This paper is a revised and expanded descendant of Piccinini (2007c). Sections 6 and 7, most of Sections 1 and 4, and part of Section 5 are new. Thanks to Carl Craver, Corey Maley, Marcin Milkowski, Martin Roth, Oron Shagrir, Dan Weiskopf, the referees, and especially Anna-Mari Rusanen for helpful comments.

## References

Anderson, J. A., & Hinton, G. E. (1981). Models of information processing in the brain. In G. E. Hinton, & J. A. Anderson (Eds.), *Parallel models of associative memory* (pp. 9–48). Hillsdale, NJ: Erlbaum.

Anderson, J. A., & Rosenfeld, E. (1988). *Neurocomputing: Foundations of research*. Cambridge, MA: MIT Press.

Bechtel, W., & Abrahamsen, A. (2002). *Connectionism and the mind: Parallel processing, dynamics, and evolution in networks*. Malden, MA: Blackwell.

Chen, T., & Chen, H. (1993). Approximations of continuous functionals by neural networks with application to dynamic systems. *IEEE Transactions on Neural Networks*, *4*(6), 910–918.

Churchland, P. M. (1989). *A neurocomputational perspective*. Cambridge, MA: MIT Press.

Churchland, P. S., & Sejnowski, T. J. (1992). *The computational brain*. Cambridge, MA: MIT Press.

Cowan, J. D. (1990). McCulloch–Pitts and related neural nets from 1943 to 1989. *Bulletin of Mathematical Biology*, *52*(1–2), 73–97.

Craver, C. F. (2007). *Explaining the brain*. Oxford: Oxford University Press.

Cummins, R. (1977). Programs in the explanation of behavior. *Philosophy of Science*, *44*, 269–287.

Cummins, R. (1983). *The nature of psychological explanation*. Cambridge, MA: MIT Press.

Cummins, R. (1989). *Meaning and mental representation*. Cambridge, MA: MIT Press.

Cummins, R., & Schwarz, G. (1991). In T. Horgan, & J. Tienson (Eds.), *Connectionism and the philosophy of mind*, *Connectionism, computation, and cognition* (pp. 60–73). Dordrecht: Kluwer.

Dayan, P., & Abbott, L. F. (2001). *Theoretical neuroscience: Computational and mathematical modeling of neural systems*. Cambridge, MA: MIT Press.

Edelman, G. M. (1992). *Bright air, brilliant fire: On the matter of the mind*. New York: Basic Books.

Eliasmith, C. (2003). Moving beyond metaphors: Understanding the mind for what it is. *Journal of Philosophy*, *C*(10), 493–520.

Feldman, J. A., & Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive science*, *6*, 205–254. Reprinted in Anderson and Rosenfeld (1988), pp. 484–508.

Fodor, J. A. (1975). *The language of thought*. Cambridge, MA: Harvard University Press.

Fodor, J. A., & Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture. *Cognition*, *28*, 3–71.

Freeman, W. J. (2001). *How brains make up their minds*. New York: Columbia University Press.

Gallistel, C. R., & Gibbon, J. (2002). *The symbolic foundations of conditioned behavior*. Mahwah, NJ: Lawrence Erlbaum Associates.

Globus, G. G. (1992). Towards a noncomputational cognitive neuroscience. *Journal of Cognitive Neuroscience*, *4*(4), 299–310.

Hebb, D. O. (1949). *The organization of behavior: A neuropsychological theory*. New York: Wiley.

Horgan, T., & Tienson, J. (1996). *Connectionism and the philosophy of psychology*. Cambridge, MA: MIT Press.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, *79*, 2554–2558. Reprinted in Anderson and Rosenfeld (1988), pp. 460–464.

Householder, A. S., & Landahl, H. D. (1945). *Mathematical biophysics of the central nervous system*. Bloomington: Principia.

Koch, C. (1999). *Biophysics of computation: Information processing in single neurons*. New York: Oxford University Press.

Macdonald, C., & Macdonald, G. (1995). *Connectionism: Debates on psychological explanation*: *Vol. 2*. Oxford: Blackwell.

Marr, D., & Poggio, T. (1976). Cooperative computation of stereo disparity. *Science*, *194*, 283–287.

McCulloch, W. S., & Pitts, W. H. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *7*, 115–133.

Minsky, M. L., & Papert, S. A. (1969). *Perceptrons: An introduction to computational geometry*. Cambridge, MA: MIT Press.

Newell, A., & Simon, H. A. (1976). Computer science as an empirical enquiry: Symbols and search. *Communications of the ACM*, *19*, 113–126.

O'Brien, G., & Opie, J. (2006). How do connectionist networks compute?. *Cognitive Processing*, *7*, 30–41.

Perkel, D. H. (1990). In E. L. Schwartz (Ed.), *Computational neuroscience*, *Computational neuroscience: Scope and structure* (pp. 38–45). Cambridge, MA: MIT Press.

Piccinini, G. (2004a). Functionalism, computationalism, and mental contents. *Canadian Journal of Philosophy*, *34*(3), 375–410.

Piccinini, G. (2004b). The first computational theory of mind and brain: A close look at McCulloch and Pitts's 'logical calculus of ideas immanent in nervous activity'. *Synthese*, *141*(2), 175–215.

Piccinini, G. (2007a). Computational modeling vs. computational explanation: Is everything a Turing machine, and does it matter to the philosophy of mind?. *Australasian Journal of Philosophy*, *85*(1), 93–115.

Piccinini, G. (2007b). Computational explanation and mechanistic explanation of mind. In M. De Caro, F. Ferretti, & M. Marraffa (Eds.), *Cartographies of the mind: Philosophy and psychology in intersection* (pp. 23–36). Dordrecht: Springer.

Piccinini, G. (2007c). Connectionist computation. In *International joint conference on neural networks 2007 conference proceedings*. CD-ROM, International Neural Network Society and IEEE Computational Intelligence Society.

Piccinini, G. (2008). Computation without representation. *Philosophical Studies*, *137*(2).

Piccinini, G. (in press a). Computing mechanisms. Philosophy of Science. Q8

Piccinini, G. (in press b). Computers. Pacific Philosophical Quarterly.

Pinker, S. (1997). *How the mind works*. New York: Norton.

Port, R., & van Gelder, T. (1995). Introduction. In R. Port, & T. van Gelder (Eds.), *Mind as motion: Explorations in the dynamics of cognition*. MIT Press: Cambridge, MA.

Pour-El, M. B. (1974). Abstract computability and its relation to the general purpose analog computer (Some connections between logic, differential equations and analog computers). *Transactions of the American Mathematical Society*, *199*, 1–28.

Pylyshyn, Z. W. (1984). *Computation and cognition*. Cambridge, MA: MIT Press.

Rashevsky, N. (1938). *Mathematical biophysics: Physicomathematical foundations of biology*. Chicago: University of Chicago Press.

Rashevsky, N. (1940). *Advances and applications of mathematical biology*. Chicago: University of Chicago Press.

Rey, G. (1997). *Contemporary philosophy of mind: A contentiously classic approach*. Cambridge, MA: Blackwell.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*, 386–408. Reprinted in Anderson and Rosenfeld (1988), pp. 92–114.

Rosenblatt, F. (1962). *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Washington, DC: Spartan.

Roth, M. (2005). Program execution in connectionist networks. *Mind and Language*, *20*(4), 448–467.

Schwartz, J. T. (1988). The new connectionism: Developing relationships between neuroscience and artificial intelligence. *Daedalus*, *117*(1), 123–141.

Shagrir, O. (2006). Why we view the brain as a computer. *Synthese*, *153*(3), 393–416.

Siegelmann, H. T. (1999). *Neural networks and analog computation: Beyond the Turing limit*. Boston, MA: Birkhäuser.

Šíma, J., & Orponen, P. (2003). General-purpose computation with neural networks: A survey of complexity theoretic results. *Neural Computation*, *15*, 2727–2778.

Smolensky, P., & Legendre, G. (2006). *The harmonic mind: From neural computation to optimality-theoretic grammar. Vol. 1: Cognitive architecture; Vol. 2: Linguistic and philosophical implications*. Cambridge, MA: MIT Press.

Thorndike, E. L. (1932). *The fundamentals of learning*. New York: Teachers College, Columbia University.

van Gelder, T. (1995). "What might cognition be, if not computation?". *The Journal of Philosophy*, *XCII*(7), 345–381.

van Gelder, T. (1997). Connectionism, dynamics, and the philosophy of mind. In M. Carrier, & P. K. Machamer (Eds.), *Mindscapes: Philosophy, science, and the mind* (pp. 245–269). Pittsburgh: University of Pittsburgh Press.

van Gelder, T. (1998). The dynamical hypothesis in cognitive science. *Behavioral and Brain Sciences*, *XXI*, 615–665.

Werbos, P. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph.D. Dissertation. Cambridge, MA: Harvard University.

Widrow, B., & Hoff, M.E. (1960). Adaptive switching circuits. 1960 IRE WESCON convention record (pp. 96–104). New York: IRE. Reprinted in Anderson and Rosenfeld (1988), pp. 126–134.

Wimsatt, W. C. (2002). Functional organization, analogy, and inference. In A. Ariew, R. Cummins, & M. Perlman (Eds.), *Functions: New essays in the philosophy of psychology and biology* (pp. 173–221). Oxford: Oxford University Press. Q9



Please cite this article in press as: Piccinini, G. Some neural networks compute, others don't. Neural Networks (2008), doi:10.1016/j.neunet.2007.12.010