Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

Neural networks in the CSA model

E. Eberbach

Jodrey School of Computer Science, Acadia University Wolfville, Nova Scotia, Canada B0P 1X0

ABSTRACT

Neural Networks provide a powerful tool for new generation computers. The biggest problem of neural networks is the lack of representational power. We propose to analyze neural networks using the approach which is more general than neural networks. A Calculus of Self-modifiable Algorithms is a universal model for intelligent and parallel systems, integrating different styles of programming, and applied in different domains of future generation computers. Applying this model to neural networks gives some hints how to increase a representational power of neural networks.

INTRODUCTION

Neural Network processing provides a new way of thinking about perception, memory, learning and thinking [19]. Artificial Intelligence community is split between advocates of powerful symbolic representations that lack efficient learning procedures, and advocates of connectionist ideas which provide relatively simple learning procedures that lack the ability to represent complex structures effectively [9].

We believe that both approaches, a symbolic conventional computing and neural network processing are, in fact, compatible [6,7]. However they use different techniques, models, terminology, and representation. The differences lead to the situation that some classes of problems are easier to express in a connectionist model, other using a symbolic approach. The relations of symbolic computing with neural nets have been discussed in more details in [6,7].

The paper suggests a very general notation for describing neural networks and explains how various aspects of neural nets map into this notation. The notation seems appropriate for describing systems of a large number of interacting and fairly complex modules. For this purpose the Calculus of Self-modifiable Algorithms has been applied.

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

The Calculus of Self-modifiable Algorithms (CSA for short) [1-7] is a universal theory for parallel and intelligent systems. It integrates different styles of programming and is applied in various domains of future generation computers.

In section 2 we briefly outline CSA. In section 3 we present neural networks in the CSA interpretation. In section 4 we demonstrate how neural networks work as self-modifiable algorithms. For the illustration the most popular type of neural networks has been chosen: a multi-layer perceptron using the back propagation learning algorithm.

A CALCULUS OF SELF-MODIFIABLE ALGORITHMS

A Calculus of Self-modifiable Algorithms (CSA [1-7]) is a universal theory for parallel and intelligent systems, integrating different styles of programming and applied in various domains of future generation computers. We will decipher its name. The name "Calculus" should be understood rather not in the pure mathematical sense, but by the analogy to Milner's Calculus of Communicating Systems [16] and Hoare's Communicating Sequential Processes [10]. The second part "Self-modifiable Algorithms" does not mean pure algorithms only. Self-modifiable Algorithms model adaptive computer systems (programs, formal languages, architectures) and their name is by an analogy to the mathematical models of programs without self-modifiability, so-called Mazurkiewicz algorithms [15].

To basic notions of the CSA belong: *Extended Boolean Algebra formulas (EBA-formulas), Self-Modifiable Algorithm net (SMA-net), a cost system, extended regular expressions, and self-modifiable algorithm* itself. EBA-formulas are a basic construct of the SMA-net, which in turn provides a formal environment for self-modifiable algorithms. Two special cases of the SMA-net are the Rule-Based and Logic net, and the Neural Network and Connectionist net. A cost system is used for the solution of the optimization process of self-modifiable algorithms (in particular for learning or fault-tolerance). Extended regular expressions are a special case of the SMA-net. Self-modifiable algorithms are a mathematical abstraction of adaptive computer systems. The input-output behavior of self-modifiable algorithms is described as the solutions (the fixed points) of some sets of equations over SMA-net. The equations are built using the SMA-net control states as variables, transits as parameters, and SMA-net operators as equation continuous operators.

The scope of potential and realized applications of CSA is very wide, including expert systems, machine learning, adaptive systems, neural networks, fault-tolerant systems, robot plan generation, distributed and concurrent computing, and new generation computer architectures and languages [2-7].

The EBA-formulas are useful to deal with systems with incomplete information, and such systems are characteristic for AI and neurocomputing. Some basic elements of self-modifiable algorithms, i.e. control states, pre- and postconditions in transits are predicates taking values in the Extended Boolean Algebra (EBA-formulas). EBA-formulas are in some extent similar to the predicate calculus used in logic programming. The main differences are that predicates can take three and not two values, namely: true, false and unknown; and that be-

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

sides traditional disjunction, conjunction, and negation operators, there are new "dynamic" operators *verification of matching*, *strings*, *updating condition*, and describing sequential and parallel behavior.

A SMA-net (from: Self-Modifiable Algorithm net) defines a mathematical environment for self-modifiable algorithms. Two special cases of the SMA-net are an RBL-net (from: Rule-Based and Logic net) and an NNC-net (from: Neural Network and Connectionist net). The SMA-net provides the means for procedural and data abstraction, i.e. the way how to build the algorithm from lower-level elements using basic programming operators. Elements of SMA-net consist of preconditions PRE, activity ACT and postcondition $POST$. We assume that an element of SMA-net $PRE \rightarrow ACT \rightarrow POST$ matches if its precondition is true. In other words, if EBA-formula in precondition has value 1, action ACT can be executed. If PRE is equal 0, ACT cannot be executed; and if PRE is unknown then there are two possibilities: either suspension or conditional execution with verification of PRE during or after execution. This has an analogy with the reasoning by default from logic programming. After correct termination of action ACT, $POST$ becomes true.

By a **SMA-net** we mean the following system:

$$(X, \{pre, act, post\}, \{\uplus, \sqcup, \sqcap, \circ, \ , , \Re, !\}, \{\bot, \top, \varepsilon, \gamma\}),\ \text{where}$$

(a) X is a set of atomic elements, called a *universum* (domain) of the net. $X = \Phi \times \Phi \times \Phi$, where Φ are EBA-formulas. In other words, elements of X are triples in the form (ϕ_1, ϕ_2, ϕ_3), where ϕ_1 is a precondition, ϕ_2 is an activity, and ϕ_3 is a postcondition. Atomic elements connected by the SMA-net operators form SMA-expressions Exp(X).

(b) for every SMA-expression of $Exp(X)$ $pre, act, post$ are the projection functions allowing to find precondition, activity, and postcondition, respectively,

(c) $\uplus, \sqcup, \sqcap, \circ, \ , , \Re, !$ are the operators allowing to build higher-order elements, and

$\uplus$ is a *nondeterministic choice*,
$\sqcup$ is a *general choice*,
$\sqcap$ is an *intersection*,
$\circ$ is a *sequential composition*,
, is a *parallel composition*,
$\Re$ is a *general recursion*,
! is a *skip operator*,

(d) $\bot, \top, \varepsilon, \gamma \in X$ are distinguished elements of X, and $\bot$ is a *zero* element, $\top$ is *top*, ε is a *sequential unity*, and γ is a *parallel unity*. □

As a special case of recursion $\Re$ operator there are distinguished different types of *iteration* operators. In particular we can distinguish a (sequential) **iteration** *, a **modifiable iteration** $^{\circledast}$, a **partation** (parallel iteration) $^{\underline{*}}$, and a **modifiable partation** $^{\underline{\circledast}}$ - .□

A more complete formal definition of the SMA-net operators and their interpretations can be found in [4,7].

Self-modifiable algorithms are a mathematical model of processes (programs) with possibilities of modifications to its behavior. They describe a single (recursive or iterative) modifiable program or systems of cooperating modifiable programs. An articulation of a self-modifiable algorithm has been introduced in [1], and the current definition is based on [6,7]. Self-modifiable algorithms have many different roots, but undoubtedly their nearest analogue was a mathematical model of programs without self-modifiability, so-called Mazurkiewicz algorithm [15].

A self-modifiable algorithm, due to its open structure (based on frames, records, and objects), the ability to remember the history of its previous realizations and built-in the "optimization mechanism" that rules directions of modifications, possesses a certain self-knowledge. By the SMA model we can express and investigate both parallelism, nondeterminism, recursion, operations on data and instructions, learning and fault-tolerance.

The main idea of the *"self-modifiable algorithm"* depends on goal-directed modifications, which are able to create a completely new form of algorithm. Modifications are a kind of "vital engine" searching for the best structure of algorithm (in the sense of minimal costs) to achieve the assumed goal.

By a **self-modifiable algorithm** SMA over a SMA-net $(X, \{pre, act, post\}, \{\uplus, \sqcup, \sqcap, \circ, \ ,, \Re, !\}, \{\bot, \top, \varepsilon, \gamma\})$ we mean any pair:

$$SMA = (S, T) \text{ , where}$$

S - is a nonempty set of **control states**. Control states have the form of EBA-formulas. S is dynamic, in the sense that during execution there are created new and destroyed existing control states. At the beginning, usually, $S = \{\sigma, \omega\}$, where σ is the **initial control state**, and ω is the **terminal control state** (goal) of the algorithm.

T - is a set of **transits** . $T \subseteq X \times R^{\infty}_{\eta}$, where X is the domain of the net, and R^{∞}_{η} is the set of so called *beyond-real numbers* (in particular, real numbers). During execution of SMA new transits are created, some are modified and some destroyed. Transits consist of **data** D , **actions** A , and **modifications** M ($T = D \cup A \cup M$). The main criterion in the distinction between data, actions and modifications is a domain in which a given set operates. Actions A operate on data D exclusively. Modifications M are a generalization of actions and the essence of SMA , and they operate on data, actions and modifications. Every transit $t \in T$ has the form of a quadruple $t = (pre(t), act(t), post(t), cost(t))$, where $pre(t)$ and $post(t)$ are *precondition* and *postcondition* in the form of EBA-formulas, $act(t)$ denotes an *activity* (contents) of the transit, and $cost(t)$ is the transit weight in the form of a certain real (beyond-real) number. *pre* and *post* are responsible for pattern matching, i.e. for binding control states with appropriate transits, and *cost* is used as a criterion for adaptation and learning of SMA. The structure of transits is open in the sense that it is possible to extend it by adding new components as, for instance, generalizations of

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

transits, specializations, domain, analogies, history, informal definition, etc. The design or work of SMA (design because SMA modifies itself) consists of 3 phases: *select, examine, and execute*, and is based on the optimization mechanism with the use of the costs of transits. □

A self-modifiable algorithm with the empty set of modifications ($M = \emptyset$) is simply a traditional algorithm (conventional program) and will be called a **modified algorithm** (by other algorithms, if such exist). A self-modifiable algorithm with the empty sets of data and actions ($A \cup D = \emptyset$) will be called a **modifying algorithm** (modifying other algorithms). It is possible to define various systems of cooperating self-modifiable algorithms.

Data can be simple (as for example integer or real numbers, characters, logical values) or they may be structured (as arrays, records, symbolic data (strings, trees), relations, etc.). Data are *passive* but they can change between particular control states. Actions are *active* and they operate exclusively on data D; and at the same time, they can be data themselves for modifications M. The set of actions can be identified with instructions of "classic" imperative programming languages. Modifications are *active* and operate upon data, actions and modifications. The foregoing means that modifications are applicable to actions and that there exist modifications applicable both to actions and to data. Modifications are simply a generalization of actions. They cause generation of new instructions and change old instructions (actions and modifications) of the algorithm. Analogues of modifications can be found in knowledge representation languages (see e.g. Lenat's RLL metarules [13], updating weight functions in neurocomputing [14], Holland's genetic algorithms [11], or compiler instructions transforming other instructions).

One of the important components of transits is the **cost**, which represents the weight of a transit. The costs, represented by real numbers (more precisely **beyond-real numbers** [1]), can have different interpretations, such as, for instance, weights in the neural-computing sense, values of transits, time of execution, real cost of execution, probability, beliefs, Lenat's level of interestingness [13], penalty, subjective impression, payoff from the game theory, Holland's fitness from genetics [11], reward from psychology, time or space complexity, a linear composition of all of the above factors, and so on. The beyond-real numbers are similar to complex numbers, and were introduced in order to specify the cost of iteration (recursion) with an unknown number of repetitions. The cost of iterated and non-iterated parts of the program are then represented by a beyond-real number, consisting of two ordered real numbers, respectively [1].

The input-output behavior of self-modifiable algorithms is described as the solutions (the fixed points) of some sets of equations. The equations are built using the SMA control states as variables, transits as parameters, and SMA-net operators as equations' continuous operators. Two SMA-net operators used in the construction of the equations (*general choice* $\sqcup$ and *intersection* $\sqcap$ are at the same time the least and greatest upper bounds in complete lattices [4], respectively. Thus we can solve such sets of equations in the sense of the least and the greatest fixed points (as in the classic recursive functions theory).

The sets of equations can be built starting from the initial control state (for-

ward equations) or from the terminal control state (backward equations). Their solutions contain redundant threads of transits; therefore they are optimized by removing operators connected with nondeterminism (for instance: by removing nondeterministic choice, modifiable iteration and modifiable partation). Optimization uses the costs of transits and operations as the main criterion to find the best algorithm to realize a given goal.

Self-modifiable algorithms can be hierarchically ordered according to the number of transit levels, which operate one upon another. It is useful, in particular, for the definition of necessary and sufficient conditions for learning or adaptation. It also permits one to understand the nature of "intelligent" behavior of self-modifiable algorithm programs compared to traditional ones. According to research orientation, it seems that the CSA is competent to deal with theoretical analysis and development of general learning algorithms independently of application. There is no restriction on the type of algorithm developed.

NEURAL NETWORKS IN THE CSA APPROACH

Neural computers are parallel computer architectures which emulate the organization and function of neurons and which provide the means for pattern processing. *Neural Network Processing Element* is the basic element of neural computers. Neural net models are specified by the net topology, node characteristics, and training or learning rules. These rules specify an initial set of weights and indicate how weights should be adapted during use to improve performance [14,19]. A neural network that learns patterns, does so by adjusting the weights between neurons, analogous to synaptic weights. Through these adjustments a neural network exhibits properties of generalization and classification.

Neural Network Processing Elements or nodes used in neural net models are nonlinear, and typically analog. The simplest node sums N weighted inputs and passes the result through a nonlinearity. The node is characterized by an internal threshold and the type of nonlinearity (hard limiters, threshold logic elements, and sigmoid functions). More complex nodes may include temporal integration or other types of time dependencies and more complex mathematical operations than summation.

A *neural network processing element* has 3 components: precondition PRE, activity ACT and postcondition $POST$. **Preconditon** consists of input signals X, weights W, an **activity** F(X,W) performs different transformations on inputs and weights, and **postcondition** consists of outputs Y and updated weights W'. A neuron performs the transformation F(X,W)=(Y,W'), where some sets can be empty.

In the literature (see for instance [19]) there are distinguished a *transfer function*, which defines the relation between the inputs and the output of a neuron, and a *learning function*, which processes a state of a neuron. The transformation F is universal in the sense that it subsumes the transfer function where W' is empty (for instance, the transfer function F can be of the form $f(\sum XW) = Y$, where f is a threshold function) and the learning function with empty Y (with F, for instance, of the form $W' = W + c\triangle Y$). Compared to a natural neuron, in our approach there is more than one output signal. This allows to use the

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

same model for higher-order neurons, i.e. neural networks having more than one output signal.

Using EBA-formulas, the three steps of the formal neuron computation can be described as

$$\textbf{if } (X = X_0) \wedge (W = W_0), \textbf{ then } F(X,W), \textbf{ fi } (Y = Y_0) \wedge (W' = W'_0).$$

The uniform artificial neuron is presented in figure 1. Such neurons can be connected in higher-order neurons. Neural networks are just higher-order neurons.

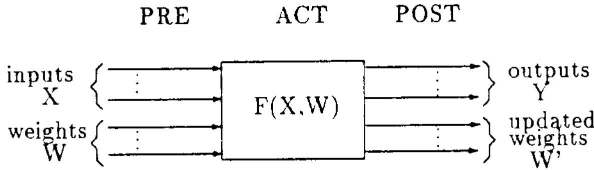

Figure 1. Neural Network Processing Element

Neural net models are specified by the net topology, node characteristics, and training or learning rules. These rules specify an initial set of weights and indicate how weights should be adapted during use to improve performance [16]. A neural network that learns patterns, does so by adjusting the weights between neurons, analogous to synaptic weights. Through these adjustments a neural network exhibits properties of generalization and classification. Typically in neural networks neurons are organized into **layers**, with each neuron in one layer having a weighted connection to each neuron in the next layer. This organization of neurons and weighted connections creates a neural network, also known as an artificial neural system (ANS).

The Neural Network and Connectionist net allows us to build higher-order neurons using operators of *compositions, choices, recursion*, etc. The advantage of this abstract approach is that using operators rather typical for programming languages, instead of hardware-like wiring of neuron connections, we obtain the common model both for hardware design and for a programming language for neurocomputers.

By an **NNC-net** (Neural Network and Connectionist net) we mean the following SMA-net:

$$(N, \{pre, act, post\}, \{\uplus, \sqcup, \sqcap, \circ, \ , , \Re, !\}, \{\bot, \top, \varepsilon, \gamma\}),$$

where N is a set of *neurons* (neural network processing elements). □

We can distinguish the following network models: Hopfield/Kohonen associative memories [12], Single-Layer Perceptron [17], Delta Rule Single-Layer Perceptron, called also Widrow-Hoff [14], Back Propagation [18], Boltzmann Machine [18], Counter Propagation [14], Self-organizing Map [14], and Neocognitron [8]. All these types of neural networks can be described using the same SMA-model. It has been demonstrated that *Neural Networks* are a subclass of self-modifiable algorithms, i.e. SMAs over the NNC-net.

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

Neural Networks are the SMAs over the NNC-net with the set of control states S represented by values of input, output and weight signals, and transits in the form of neurons. Data D are unity neurons with preconditions and postconditions from the set of input, output and weight signals, and the identity relation as the set of transforming functions. Actions A are represented by neurons performing transfer functions of inputs and weights onto outputs. Modifications M are neurons performing learning and error functions, i.e. updating weights and checking the end of training. The costs of neurons are elements to calculate the neural network *error function*, responsible for the termination of the learning process (weight modifications). The SMA optimization mechanism minimizes the error function to terminate the training phase.

A BACK PROPAGATION AS A SELF-MODIFIABLE ALGORITHM

For illustration of neural networks, we will consider the **Back Propagation Multi-Layer Perceptron.**

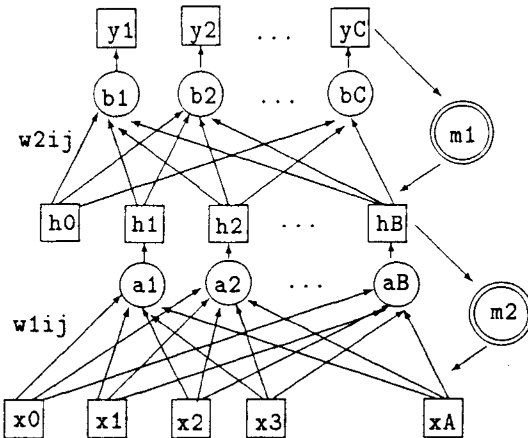

Figure 2. A two-layer perceptron using Back Propagation algorithm

- □ denotes data (datum d in the SMA example)
- ○ are actions (action a in the SMA example)
- ◎ are modifications (modification m in SMA)

The Back Propagation network is currently the most popular multi-layer perceptron system that employs supervised learning based on the generalized delta rule [18]. The basic idea of the Back Propagation model is to propagate errors backwards to hidden units that receive no direct feedback from training patterns

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

in the outside world. Output neurons calculate their errors based on the difference between the target output and the computed output, and on the derivative of the threshold function. These errors are then backpropagated to the hidden neurons and used to evaluate their errors. The Back Propagation model uses a sigmoid threshold function and accepts continuous input values. Unsimilar to a single-layer perceptron [17], there is no guarantee that the Back Propagation will converge, but if it converges, according to the Kolmogorov theorem [14,18,19], it can model an arbitrary complex mapping.

By the **Multi-Layer Perceptron using the Back Propagation Learning Algorithm** we mean the following self-modifiable algorithm over the NNC-net

$$SMA = (S, T)\ ,\ \text{where}$$

$S = \{\sigma, \omega, \sigma', \omega'\}$

$\sigma = \{select, all_transits_excluding_SET_EQ, FIX, OPT_FIX, SEL, EXAM, EXEC\}$

$\omega = end_of_execution$

$\sigma' = (set_of_inputs_x_0, x_1, ..., x_{N-1}, _desired_outputs_d_0, d_1, ..., d_{M-1}, _and_weights_w_{00}, w_{01}, ..., w_{K-1K-1} \wedge a.cost > \epsilon) \vee (a.cost \leq \epsilon)$

$\omega' = a.cost \leq \epsilon$

/* $a.cost$ denotes the cost of action a and ϵ is a very small positive real number*/

$T = \{D = \{d, SET_EQ, FIX, OPT_FIX\}, A = \{a\}, M = \{m, SEL, EXAM, EXEC\}\}$

datum(d) /* a set of inputs, weights and desired outputs */

pre: $set_of_inputs_x_0, x_1, ..., x_{N-1}, _desired_outputs_d_0, d_1, ..., d_{M-1}, _and_weights_w_{00}, w_{01}, ..., w_{K-1K-1} \wedge a.cost > \epsilon$

act: /* empty activity */

post: $set_of_inputs_x_0, x_1, ..., x_{N-1}, _desired_outputs_d_0, d_1, ..., d_{M-1}, _and_weights_w_{00}, w_{01}, ..., w_{K-1K-1}$

cost: /* empty */

datum(SET_EQ) /* set of equations for the select phase */

pre: $all_transits_excluding_SET_EQ, FIX, OPT_FIX, SEL, EXAM, EXEC$

act: /* empty activity */

post: $set_of_equations$

cost: /* empty */

datum(FIX) /* the least fixed point of the equations from the select phase*/

pre: $set_of_equations$

act: /* empty activity */

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

post: *optimal_least_fixed_point*

cost: /* empty */

datum(OPT_FIX) /* the optimal least fixed point (with the least cost)*/

pre: *optimal_least_fixed_point*

act: /* empty activity */

post: 1

cost: /* empty */

action(a) /* calculation of neurons' outputs */

pre: *set_of_inputs_*$x_0, x_1, ..., x_{N-1}$, *_desired_outputs_*$d_0, d_1, ..., d_{M-1}$, *_and_weights_*$w_{00}, w_{01}, ..., w_{K-1K-1}$

act: forward propagation of input signals through layers to calculate outputs $y_0, y_1, ..., y_{M-1}$, using the sigmoid nonlinearity

$$y_j' := \frac{1}{1+e^{-\sum_i w_{ij} x_i'}},$$

where x_i' are the inputs of internal hidden or output neurons, y_j' are the outputs of internal hidden or output neurons. The input vector can be new on each trial, or samples from a training set can presented cyclically until weights stabilize

post: *calculated_outputs_*y_j'

cost: $\sum_k \sum_j |d_{kj} - y_{kj}|$ /* error function between desired and calculated outputs for all outputs j of the output layer neurons and input samples k */

modification(m) /* modifications updating neurons' weights */

pre: *calculated_outputs_*y_j'

act: back propagation of the new values of weights (adjusting weights) starting from the output neurons to the input neurons using the formula

$$w_{ij} := w_{ij} + c\delta_j x_i',$$

where w_{ij} is the weight from hidden or input neuron i to neuron j, x_i' is the input of neuron i, c is a gain term, and δ_j is an error term for neuron j given by the following formulas:

if neuron j is an output neuron, then

$$\delta_j := y_j(1 - y_j)(d_j - y_j),$$

where d_j is the desired output of neuron j, and y_j is the actual output;

if neuron j is an internal hidden neuron, then

$$\delta_j := x_j'(1 - x_j') \sum_k \delta_k w_{jk},$$

where k is over all neurons in the layers above neuron j;

update cost (error) for sample k

$$a.cost_k := \sum_j |d_{kj} - y_{kj}|$$

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

post: $set_of_inputs_x_0, x_1, ..., x_{N-1}, _desired_outputs_d_0, d_1, ...$
$, d_{M-1}, _and_weights_w_{00}, w_{01}, ..., w_{K-1K-1} \wedge a.cost > \epsilon$

cost: /* empty */

modification(SEL) /* select phase */

pre: *select*

act: build the equations forward or backward with the initial control state σ' and the terminal control state ω'

post: *examine*

cost: /* empty */

modification($EXAM$) /* examine phase */

pre: *examine*

act: solve the equations from the select phase; calculate the costs of the least fixed points; perform minimization of costs of a subset of transits chosen for optimization (in our case, the cost of neuron a); if the goal of optimization has been achieved (i.e. $a.cost \leq \epsilon$), then remove modifications responsible for optimization (m will be replaced by transparent ε and a, d, σ', ω' modified), else wait and check the condition again

post: *execute*

cost: /* empty */

modification($EXEC$) /* execute phase */

pre: *execute*

act: execute the least fixed point from the examine phase

post: *end_of_execution*

cost: /* empty */

In the above SMA transits SET_EQ,FIX,OPT_FIX,SEL,$EXAM$, $EXEC$ belong to the inference engine of SMA and they are characteristic for all SMAs independently of the task to solve. Remaining transits d, a, m are the proper back-propagation multi-perceptron transits and are characteristic (this means changeable) for each specific problem.

The inference engine of the SMA works in the following way:

$x(\sigma) = (SEL\ , SET_EQ) \circ x(s_1)$
$x(s_1) = (EXAM\ , FIX) \circ x(s_2)$
$x(s_2) = (EXEC\ , OPT_FIX) \circ x(\omega)$
$x(\omega) = \varepsilon$

where

$\sigma = \{select, all_transits_excluding_SET_EQ, FIX, OPT_FIX, SEL, EXAM, EXEC\}$
$s_1 = \{examine, set_of_equations\}$
$s_2 = \{execute, optimal_least_fixed_point\}$
$\omega = end_of_execution$

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

The solution of this set of equations describes a single loop of the SMA inference engine:

$$x(\sigma) = (SEL\,, SET_EQ) \circ (EXAM\,, FIX) \circ (EXEC\,, OPT_FIX)$$

This means that modification SEL works in the select phase on its datum SET_EQ building the set of equations by matching control states and transits. Next, modification $EXAM$ works on its datum FIX in the examine phase, (i.e., it solves the set of equations SET_EQ in the sense of the least fixed points, optimizes the solution, and passes the optimal least fixed point OPT_FIX to the execute phase. In the execute phase modification $EXEC$ executes optimal solution OPT_FIX.
In the general case, this loop is iterated, i.e.

$$x(\sigma) = ((SEL\,, SET_EQ) \circ (EXAM\,, FIX) \circ (EXEC\,, OPT_FIX))^*$$

This means that the problem to solve is subdivided into subproblems, which are solved in successive inference engine loops.

Data SET_EQ, FIX and OPT_FIX are modified during work by appropriate transits $SEL, EXAM$ and $EXEC$, and their transient values are presented below.

Datum SET_EQ is built in the **select phase** of SMA by matching control states $(\sigma', \alpha, \beta, ...)$ with transits $(d, a, ...)$, which can be written in the form of the below equations:

Select phase: matching control states with transits

$x(\sigma') = d \circ x(\alpha) \sqcup x(\omega')$
$x(\alpha) = a \circ x(\beta)$
$x(\beta) = m \circ x(\sigma')$
$x(\omega') = \varepsilon$

where

$\sigma' = (set_of_inputs_x_0, x_1, ..., x_{N-1}, _desired_outputs_d_0, d_1, ..., d_{M-1},$
$\quad _and_weights_w_{00}, w_{01}, ..., w_{K-1K-1} \wedge a.cost > \epsilon) \vee (a.cost \leq \epsilon)$
$\alpha = set_of_inputs_x_0, x_1, ..., x_{N-1}, _desired_outputs_d_0, d_1, ..., d_{M-1},$
$\quad _and_weights_w_{00}, w_{01}, ..., w_{K-1K-1}$
$\beta = calculated_outputs_y_j{}'$
$\omega' = a.cost \leq \epsilon$

Examine phase: depends on the calculation of the fixed points, costs and optimization. The process of the Multi-Layer Perceptron with the Back Propagation learning has the symbolic form as the least fixed point (datum FIX) of the above set of equations:

$$x(\sigma') = (d \circ a \circ m)^*$$

Execute phase: the optimal least fixed point from the examine phase is executed, i.e m is removed, and d,a,σ' and ω' are modified: $d \circ a$

Together this means that the whole work of our SMA, learning correct weigth values, can be represented as follows:

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

$$x(\sigma) = \underbrace{(SEL\,, SET_EQ) \circ (EXAM\,,\ \underbrace{(d \circ a \circ m)^*}_{\textit{the least fixed point}}\ )}_{\textit{learning phase}} \circ$$

$$\underbrace{\circ\ (EXEC\,,\ \underbrace{(d \circ a))}_{\textit{optimal least fixed point}}}_{\textit{taught multi-layer perceptron}}$$

Note that the same formalism (symbolic equations, the least fixed points and optimization), has been used to describe the work of the SMA inference engine, and the learning neural network itself. In the optimization process, the cost of only one transit a has been minimized (therefore costs of all other transits were irrelevant-empty). In the general case, the SMA optimization mechanism can minimize costs of all transits (including costs of transits from the inference engine itself). This means that either partial or total optimization problems are solved. The above also means that using self-modifiable algorithms we can minimize costs of learning, i.e. we can consider learning on higher levels ("learn how to learn").

Weight modification could be included to the modification $EXAM$, but then it would be invisible, therefore it has been written separately as modification m.

From the example it follows that the CSA approach is more general than neural networks. The open problems remains how to model the inference engine of SMAs (transits SET_EQ, FIX, OPT_FIX, SEL, $EXAM$, $EXEC$) as neurons. We believe that it is possible. If so then we obtain a powerful tool to span conventional computing with parallel distributed processing.

CONCLUSIONS

In the present paper we show that the Calculus of Self-modifiable Algorithms could be used for the design of neural networks. The SMA-net operators provide the means to build higher order neurons, and existing neural network models do not have such a hierarchical representation. We believe that such a representation is typical for human brain, because biological neurons form other higher-order structures responsible for vision, motor skills, abstract thinking. Additionally, the structuralization links connectionist ideas with symbolic computing. The symbolic computing is so powerful because of the many levels of abstraction. Why do not allow this for neural networks? Why should be separate neurons-hardware, and powerful, but having an unknown representation, learning algorithms? The current neural networks learning algorithms are not presented in the form of neurons but also not as software, because according to connectionist ideas neural networks do not require programming. What are they? Something beyond the neuron model? The answer of CSA is simple and clear: learning algorithms are also higher-order neurons. More precisely, neurons working upon other neurons, i.e. SMA modifications. Neural networks learning algorithms are very simple: a few synchronized steps of parallel operations repeated iteratively. In the CSA we can express more complex learning algorithms. We can model neural networks too. Thus we can conclude that in the connectionist approach there is still a room

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

for more complex learning as well (if it is desirable). CSA provides the means to believe that neurons perform recursive computations (a SMA-net general recursion operator and its special cases: iterative operators). In the current state of knowledge, we do not know however, how to build stable multi-layer recursive neural networks. Maybe by adding the clocking signals like in conventional sequential circuits? CSA can be treated as a formal language for neural networks. Because of such an abstract model, we obtain the conclusion that it is irrelevant whether neural networks will be treated as hardware elements only, or also as software-like constructs.

ACKNOWLEDGEMENTS
The work of the author was partially supported by a grant from the Natural Sciences and Engineering Research Council of Canada, No. OGP0046501, and a general NSERC grant.

REFERENCES

1. Eberbach E., Algorithms with Possibilities of Selflearning and Selfmodification, Fundamenta Informaticae 6, 1 (1983), 1-44.

2. Eberbach E., Self-Modifiable Algorithms and Their Applications, Research Note RN/88/27, Department of Computer Science, University College London, (June 1988).

3. Eberbach E., Selected Aspects of the Calculus of Self-Modifiable Algorithms Theory, Lect. Notes in Computer Science 468, Springer-Verlag, 1990, 34-43.

4. Eberbach E., CSA: Two Paradigms of the Language for Adaptive Expert Systems, Proc. of the 19th Annual ACM Computer Science Conference CSC'91, San Antonio, Texas, (1991), 570-581.

5. Eberbach E., Neural Network Processing Elements as a New Generation of Flip-Flops, in: Advances in Computing and Information - ICCI'91, Lect. Notes in Computer Science 497, Springer-Verlag, Ottawa, (1991), 687-698.

6. Eberbach E., Adaptive Expert Systems and Neural Networks Applications of CSA, Proc. of the Fourth Intern. Conf. on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems IEA/AIE-91, vol.I, Koloa, Kauai, Hawaii, (1991), 365-374.

7. Eberbach E., Neural Networks and Adaptive Expert Systems in the CSA Approach, Special Issue on Neural Networks, Intern. Journal of Intelligent Systems, vol. 8 (4), 1993, in printing.

8. Fukushima K., A Neural Network for Visual Pattern Recognition, IEEE Computer, March 1988, 65-75.

9. Hinton G.E., Connectionist Symbol Processing, MIT Press, 1991.

10. Hoare C. A. R., Communicating Sequential Processes, Prentice-Hall, 1985.

Transactions on Information and Communications Technologies vol 2, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

11. Holland J.H., Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, Univ. of Michigan Press, Ann Arbor, 1975.

12. Hopfield J.J., Neural Networks and Physical Systems with Emergent Collective Computational Abilities, Proc.Nat.Acad.Sci., vol.79, 1982, 2554-2558.

13. Hayes-Roth F., Waterman D.A., Lenat D.B., Building Expert Systems, Addison-Wesley, 1983.

14. Lippmann R.P., An Introduction to Computing with Neural Nets, IEEE ASSP Magazine, April 1987, 4-22.

15. Mazurkiewicz A., Iteratively Computable Relations, Bulletin of the Polish Academy of Sciences, Ser. Math. Astron. Phys., 20, 9, (1972), 799-803.

16. Milner R., A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol.92, Springer-Verlag, 1980.

17. Rosenblatt R., Principles of Neurodynamics, New York, Spartan Books, 1959.

18. Rumelhart D.E., McClelland J.L., Parallel Distributed Processing: Explorations in the Microstructure of Cognition, MIT Press, Cambridge,Mass., vol. 1 & 2, 1986.

19. Souček B., Souček M., Neural and Massively Parallel Computers. The Sixth Generation, John Wiley & Sons, 1988.