STEPS TOWARDS 'SELF-REFERENTIAL' NEURAL LEARNING: A THOUGHT EXPERIMENT
Technical Report CU-CS-627-92

Jürgen Schmidhuber
Department of Computer Science
University of Colorado
Campus Box 430, Boulder, CO 80309

November 11, 1992

Abstract

A major difference between human learning and machine learning is that humans can reflect about their own learning behavior and adapt it to typical learning tasks in a given environment. To make some initial theoretical steps toward 'introspective' machine learning, I present – as a thought experiment – a 'self-referential' recurrent neural network which can run and actively modify its own weight change algorithm. The network has special input units for observing its own failures and successes. Each of its connections has an address. The network has additional special input and output units for sequentially addressing, analyzing and manipulating *all* of its own adaptive components (weights), including those weights responsible for addressing, analyzing and manipulating weights. Due to the generality of the architecture, there are no theoretical limits to the sophistication of the modified weight change algorithms running on the network (except for unavoidable pre-wired time and storage constraints). In theory, the network's weight matrix *can learn not only to change itself, but it can also learn the way it changes itself, and the way it changes the way it changes itself* — and so on *ad infinitum.* No endless recursion is involved, however. For one variant of the architecture, I present a simple but general *initial* reinforcement learning algorithm. For another variant, I derive a more complex exact gradient-based algorithm for supervised sequence learning. A disadvantage of the latter algorithm is its computational complexity per time step which is independent of the sequence length and equals $O(n_{conn}^2 log n_{conn})$, where n_{conn} is the number of connections. Another disadvantage is the high number of local minima of the unusually complex error surface. The purpose of my thought experiment, however, is not to come up with the most efficient or most practical 'introspective' or 'self-referential' weight change algorithm, but to show that such algorithms are possible at all.

1 INTRODUCTION

Every machine learning researcher knows that learning algorithms that work well for one class of problems are not always well suited for a different class. In contrast, humans can reflect about their own learning behavior and tailor it to the needs of various types of learning problems[1].

The first step toward a theory of 'introspective' or 'self-referential' machine learning[2] is the design of a general finite-size hard-wired architecture with 'self-referential' potential. The second step is the design of *initial* learning algorithms that find useful self-manipulating algorithms for the architecture, where 'usefulness' is strictly defined by performance evaluations provided by the environment.

In the following thought experiment I will choose an artificial recurrent neural network as basis for a 'self-referential meta-learning' architecture. Two reasons for this choice are: *(a) Generality.* Recurrent neural nets are simple yet powerful – like a Turing machine or a conventional digital computer they can be programmed to generate arbitrary mappings from input sequences to output sequences (loosely speaking, a finite network is like a Turing machine with finite tape). This general property will allow us to build a net that can run its own arbitrarily complex weight change algorithm. *(b) Simplicity of avoiding catastrophic errors.* A lisp program that is allowed to change itself may be likely to crash due to catastrophic syntactic errors (e.g. [4]). It is easy to avoid similar situations in the context of self-changing weight-matrices (the programs of recurrent nets).

This paper is intended to go beyond certain previous and less general approaches to 'meta-learning'. For instance, Chalmer's recent architecture (like other similar architectures) is based on hierarchical levels and could not be called 'self-referential' – in his case, a genetic algorithm on a higher level tries to find a simple learning rule for a lower level neural feed-forward network [2]. In the paper at hand, however, I want to bury all 'meta-levels' within the same finite network. In addition, with previous approaches the search space of possible learning algorithms is usually very limited – Chalmer, for instance, limits his weight change algorithms to simple linear combinations of certain parameters of his feed-forward network. In contrast, I do not want to impose any non-trivial limitations on the complexity of evolving learning algorithms.

Outline. The remainder of this paper is structured as follows: Section 2 starts with a general set-up for an agent interacting with its environment. Then it introduces a finite, 'self-referential' architecture that controls the agent. This architecture involves a recurrent neural-net that can potentially implement any computable function that maps input sequences to output sequences — the only limitations being unavoidable time and

[1] Here are some very 'high-level' examples of 'self-referential' behavior: We can make statements about our own learning procedures, such as, "Today I am going to sit down and learn 20 Japanese words". Not only can we talk about, but we can also manipulate the way we learn. For instance, to a large extent we can choose our own training examples and which associations we want to memorize. And we actually can learn how to improve our own learning procedures, profitting from previous learning experiences like this: "When I began learning to juggle, practicing with all three balls was not helpful. But after first practicing with one, and then with two, I could finally start making use of all three. This time I am going to learn to juggle with three bottles instead of balls, and I will start with one bottle instead of three". For many problem classes we develop quite specific representations and efficient learning strategies. These specific strategies may include certain forms of 'analogy matching' (see [8] for a neural approach to inter-task transfer learning), 'chunking', 'one-shot learning', 'active learning', 'query learning' etc., but there is much reason to suspect that there are many specific strategies which are not adequately described by any of these familiar terms After some years of self-observation, I have come to believe that processes for 'learning how to learn' frequently occur on much lower cognitive levels than the ones corresponding to the examples above, though it is often difficult to describe lower levels in words. Based on these observations, I have come to believe such processes are essential for the scaling behavior of any realistic big learning system.

[2] Intuitively speaking, the ultimate goal of research on 'self-referential' and 'introspective' learning algorithms (as I see it) is something I would like to call a *'learning germ'*. The *'germ'* is a hypothetical, initially simple, but general learning algorithm running on realistic finite-size hardware with limited time and storage resources. With a given environment demanding the solution of certain tasks, the germ gradually refines and specializes itself in a 'bootstrap' fashion in order to develop specialized representations and strategies for attacking typical learning tasks, thereby learning to make 'optimal' use of the limited temporal and storage resources of the unchangeable hardware. The germ might do so by collecting information about typical problems, the relations between typical problems, and how to use these relations to create efficient ways of solving typical problems. (In the context of concept learning, this is sometimes referred to as finding the proper *'inductive bias'* [3][16].) Specialized strategies may include but need not be limited to things known as 'learning by analogy', 'learning by chunking', 'one-shot learning', 'active learning', 'query learning', etc. I do not claim to have arrived at the ultimate goal. (That's why the title is "steps toward 'self-referential' neural learning" instead of "self-referential neural learning", and why the word 'self-referential' is put between apostrophes.)

storage constraints imposed by the architecture's finiteness. These constraints can be extended by simply adding storage and/or allowing for more processing time. The major novel feature of the system is its 'self-referential' capability. The network is provided with special input units for explicitly observing performance evaluations. In addition, it is provided with the basic tools for explicitly reading and changing *all* of its own adaptive components (weights): It has special output and input units for sequentially addressing, analyzing and manipulating any weight, including those weights responsible for addressing, analyzing and manipulating weights. These unconventional properties allow the network (in principle) to compute any computable function mapping *algorithm components* and *performance evaluations* to *algorithm modifications* — the only limitations again being unavoidable time and storage constraints. This implies that algorithms running on that architecture (in principle) *can change not only themselves but also the way they change themselves, and the way they change the way they change themselves, etc.*, essentially without theoretical limits to the sophistication (computational power) of the self-modifying algorithms. The architecture should be viewed as only one of many possible similar architectures (using different schemes for addressing and manipulating connections). In section 3, *initial* weight change algorithms will be designed such that the 'self-referential' network can learn (possibly self-modifying) algorithms that improve their performance on given tasks[3]. The system starts out as *tabula rasa*. The initial learning procedure favors algorithms that make sensible use of the 'introspective' potential of the hard-wired architecture, where 'usefulness' is solely defined by conventional performance evaluations. For a variant of the network, section 3.1 derives an exact gradient-based initial algorithm for 'self-referential' supervised sequence learning. For another variant, section 3.2 describes an even more general but less informed (and less complex) reinforcement learning algorithm.

The thought experiment presented in this paper is intended to show the theoretical possibility of a certain kind of 'self-referential' neural networks – the systems described herein are not meant to be very practical ones. For instance, due to the complexity of the activation dynamics of the 'self-referential' network to be described in the next section, the error function to be minimized by the supervised sequence learning algorithm derived in section 3.1 would probably be riddled with local minima. Another disadvantage of this algorithm is its computational complexity per time step which is independent of the sequence length and equals $O(n_{conn}^2 log n_{conn})$, where n_{conn} is the number of connections in the net. The purpose of this paper, however, is not to come up with the most efficient or most practical 'introspective' or 'self-referential' weight change algorithm, but to show that such algorithms are possible at all. More practical variants of the basic idea will be left for separate papers (e.g. [15]).

2 THE 'SELF-REFERENTIAL' SYSTEM

An information processing 'agent' is embedded in a general dynamic environment. At a given time step t, the agent's time-varying input is $x(t)$. Its current output $o(t)$ is computed from previous inputs and may influence the environmental state. This may in turn affect the environmental inputs. To isolate the essential information processing aspects of the agent, any 'motoric effectors' that influence the environmental state by interpreting the outputs $o(t)$ are considered to belong to the environment. The same holds for 'sensory perceptors' providing the inputs $x(t)$. Occasionally, some evaluative mechanism provides a measure of success or failure, $eval(t)$. $eval(t)$ may be quite informative (like a detailed analysis of what went wrong) or quite uninformative (like a simple pain or pleasure signal). The agent's goal is to find action sequences that lead to 'desirable' evaluations, where the environment defines what is 'desirable' and what is not. Without loss of generality, it may be assumed that $x(t), o(t), eval(t)$ are represented as real-valued vectors[4] with dimensions n_x, n_o, n_{eval}, respectively.

Obviously, the architecture of the agent constrains the set S of legal algorithms it can perform. This paper focuses on finite architectures that impose only very general, natural, and essentially unavoidable time

[3] It should be mentioned that practically all conventional computer architectures do allow 'self-referential' algorithms with essentially unconstrained power. The existing machine learning architectures (implemented on such computers), however, are more specialized and more restricted.

[4] Of course, only a subset of all real vectors can be represented by any realistic hardware. It should be noted that binary vectors could be used instead of real-valued ones without losing generality.

and storage limitations on S. A convenient way of designing such an architecture is to use a recurrent neural network as basic 'hardware'.

With this network architecture there will be algorithms in S that can produce *arbitrary* changes for any component of algorithms in S such that the results of the changes[5] are algorithms in S. This will be achieved by (1) providing the network with input units for specifically observing its own failures and successes, (2) introducing an address for each connection of the network, (3) providing the network with output units for sequentially addressing *all* of its own connections (including those connections responsible for addressing connections), (4) providing input units for analyzing the weights addressed by the network, and (5) providing output units for manipulating the weights addressed by the network.

Subsection 2.1. will list conventional aspects of the network. Subsection 2.2. will list the novel 'self-referential' features of the net. While reading the remainder of this section, you may wish to refer to figure 1 and to relevant notation listed in table 1.

2.1 CONVENTIONAL ASPECTS OF THE NET

$o(t)$ is computed from $x(\tau), \tau < t$, by a discrete time recurrent network with $n_I > n_x$ input units and n_y non-input units. A subset of the non-input units, the 'normal' output units, has a cardinality of $n_o < n_y$.

For notational convenience, I will sometimes give *different* names to the real-valued activation of a particular unit at a particular time step. z_k is the k-th unit in the network. y_k is the k-th non-input unit in the network. x_k is the k-th 'normal' input unit in the network. o_k is the k-th 'normal' output unit. If u stands for a unit, then u's activation at time t is denoted by $u(t)$. If $v(t)$ stands for a vector, then $v_k(t)$ is the k-th component of $v(t)$ (this is consistent with the last sentence).

Each input unit has a directed connection to each non-input unit. Each non-input unit has a directed connection to each non-input unit. Obviously there are $(n_I + n_y)n_y = n_{conn}$ connections in the network. The connection from unit j to unit i is denoted by w_{ij}. For instance, one of the names of the connection from the j-th 'normal' input unit to the the k-th 'normal' output unit is $w_{o_k x_j}$. w_{ij}'s real-valued weight at time t is denoted by $w_{ij}(t)$. Before training, all weights $w_{ij}(1)$ are randomly initialized.

The following definitions will look familiar to the reader knowledgeable about conventional recurrent nets (e.g. [19]). The environment determines the activations of a 'normal' input unit x_k. The activations of the remaining input units will be specified in section 2.2, which lists the 'self-referential' aspects of the architecture. For a non-input unit y_k we define

$$net_{y_k}(1) = 0, \quad \forall t \geq 1 : y_k(t) = f_{y_k}(net_{y_k}(t)),$$

$$\forall t > 1 : \quad net_{y_k}(t) = \sum_l w_{y_k l}(t-1) l(t-1), \tag{1}$$

where f_i is the activation function of unit i. I have not yet specified whether the f_i are differentiable and/or deterministic.

[5] With previous machine learning approaches I am aware of, the adaptive (non-hardwired) components of the agent algorithms can be modified *only* by some hardwired *pre-specified* and *unchangeable* learning algorithm. Lenat's complex EURISKO system may be an exception to this rule (Lenat reports that his system found heuristics for finding heuristics [4]). EURISKO's mechanism, however, has drawn criticism for being notoriously hard to evaluate and for the fact that the programmer interacted with the system in a way that remained partly opaque. In contrast, the much simpler 'subsymbolic' system described here does *not* require occasional interventions by the programmer and has clearly defined performance evaluations.

It should be noted that essentially the only general learning algorithm (for *any* environment) which is guaranteed to find the 'most desirable' algorithms in S is *exhaustive search* among all elements of S. The problem with exhaustive search is, of course, that it tends to be prohibitively inefficient when S is large or when it takes a lot of effort to evaluate elements of S. For these reasons, many more sophisticated and problem-specific machine learning algorithms have been designed (there are just too many to start citing them all). The problem-specific algorithms are often much more efficient than general algorithms like exhaustive search, but usually are limited to a much narrower range of applications. With many problem-specific learning algorithms, the *'real'* learning is going on in the researcher who puts a lot of thought into useful input representations and bias, appropriate problem-specific internal architectures, etc. In turn, sometimes the agent's remaining 'learning' task is reduced to building a collection of simple statistics. In contrast, the motivation of this paper is to think about learning algorithms that start out as general algorithms but specialize themselves according to the needs of typical learning tasks.

The current algorithm of the network is given by its current weight matrix (and the current activations). Note, however, that I have not yet specified how the $w_{ij}(t)$ are computed. In previous recurrent networks, weight changes are exclusively caused by some fixed learning algorithm with many specific limitations. For instance, the popular gradient based weight change algorithms (as well as all other known learning algorithms) suffer from many well-known drawbacks (unsatisfactory scaling behavior, problems with long time lags, etc.). Again, one would like to have weight change algorithms that are less limited. This motivates the next section.

2.2 SELF-REFERENTIAL ASPECTS OF THE NET

This section describes yet unspecified interpretations of certain inputs and outputs that make this net the first *'self-referential'* neural network with explicit potential control over *all* adaptive parameters governing its behavior. Since there may be potentially useful kinds of self-modification that are not yet known, and since appropriate self-modification procedures might be arbitrarily complex, the network supports *any* computable self-modifying algorithm that maps *algorithm components* and *performance evaluations* into *algorithm modifications* (modulo time and storage limitations).

The following is a list of four unconventional aspects of the resulting system, which should be viewed as just one example of many similar systems.

1. The network sees performance evaluations. The network receives performance information through the *eval units.* The eval units are special input units which are not 'normal' input units. $eval_k$ is the k-th eval unit (of n_{eval} such units) in the network. This feature is relatively simple compared to some of the following features, though it represents an essential contribution for achieving 'self-referential learning'.

2. Each connection of the net gets an address. One way of doing this which I employ in this paper (but certainly not the only way) is to introduce a *binary* address, $adr(w_{ij})$, for each connection w_{ij}. This will help the network to do computations concerning its own *weights* in terms of *activations*, as will be seen next.

3. The network may analyze any of its weights. ana_k is the k-th *analyzing unit* (of n_{ana} such units) in the network. The analyzing units are special non-input units which are not 'normal' output units. They serve to indicate which connections the current algorithm of the network (defined by the current weight matrix plus the current activations) will access next. It is possible to endow the analyzing units with enough capacity to address *any* connection, *including connections leading to analyzing units.* One way of doing this is to set

$$n_{ana} = ceil(log_2 n_{conn}) \tag{2}$$

where $ceil(x)$ returns the first integer $\geq x$.

A special input unit that is used in conjunction with the analyzing units is called val. $val(t)$ is computed according to

$$val(1) = 0, \quad \forall t \geq 1: \ val(t+1) = \sum_{i,j} g[||ana(t) - adr(w_{ij})||^2] w_{ij}(t), \tag{3}$$

where $|| \dots ||$ denotes Euclidean length, and g is a function emitting values between 0 and 1 that determines how close a connection address has to be to the activations of the analyzing units in order for its weight to contribute to val at that time. Such a function g might have a narrow peak at 1 around the origin and be zero (or nearly zero) everywhere else. This would essentially allow the network to pick out a single connection at a time and obtain its current weight value without receiving 'cross-talk' from other weights.[6]

4. The network may modify any of its weights. Some non-input units that are not 'normal' output units or analyzing units are called the *modifying units.* mod_k is the k-th modifying unit (of n_{mod} such units) in the

[6] Note that we need to have a compact form of addressing connections: One might alternatively think of something like 'one analyzing unit for each connection' to address all weights in parallel, but obviously this would not work — we always would end up with more weights than units and could not obtain 'self-reference'. It should be noted, however, that the binary addressing scheme above is by far not the most compact scheme. This is because real-valued activations allow for representing theoretically unlimited amounts of information in a single unit. For instance, theoretically it is possible to represent *arbitrary simultaneous changes* of all weights within a single unit. In practical applications, however, there is nothing like unlimited precision real values. And the purpose of this paper is not to present the most compact 'self-referential' addressing scheme but to present at least one such scheme.

network. The modifying units serve to address connections to be modified. Again, it is possible to endow the modifying units with enough capacity to sequentially address *any* connection, *including connections leading to modifying units.* One way of doing this is to set

$$n_{mod} = ceil(log_2 n_{conn}) \tag{4}$$

A special output unit used in conjunction with the modifying units is called $\triangle$. $f_{\triangle}$ should allow both positive and negative activations of $\triangle(t)$. Together, $mod(t)$ and $\triangle(t)$ serve to explicitly change weights according to

$$w_{ij}(t+1) = w_{ij}(t) + \triangle(t)\ g[\ \|adr(w_{ij}) - mod(t)\|^2\ ]. \tag{5}$$

Again, if g has a narrow peak at 1 around the origin and is zero (or nearly zero) everywhere else, the network will be able to pick out a single connection at a time and change its weight without affecting other weights. It is straight-forward, however, to devise schemes that allow the system to modify more than one weight in parallel. See again footnote 6.

Together, (1), (3), and (5) make up the hard-wired system dynamics.

There are many 'self-referential' networks as above. Let us refer to all units that are neither 'normal' input units, eval units, the *val* input unit, 'normal' output units, analyzing units, modifying units, nor the $\triangle$ unit, as *hidden units.* There are $n_h = n_y - n_o$ such hidden units. There is an infinity of ways of finding values n_c (denoting the number of addressable connections) satisfying the condition

$$n_c \geq (n_h + n_o + 2ceil(log_2 n_c) + 1)(n_h + n_o + 2ceil(log_2 n_c) + 1 + n_x + 1 + n_{eval}). \tag{6}$$

One example where the '=' sign is valid in (6) is given by

$$n_x = 27, n_o = n_{eval} = 4, n_{ana} = n_{mod} = 11, n_h = 5.$$

2.3 COMPUTATIONAL POWER OF THE NET

Throughout the remainder of this paper, to save indices, I consider a single limited pre-specified time-interval of discrete time-steps during which the agent interacts with its environment. I assume that the input sequence observed by the agent during this time has length $n_{time} = n_s n_r$ (where $n_s, n_r \in \mathbf{N}$) and can be divided into n_s equal-sized blocks of length n_r during which the input pattern $x(t)$ does not change. This does not imply a loss of generality — it just means speeding up the agent's hardware such that each input pattern is presented for n_r time-steps before the next pattern can be observed. This gives the agent n_r time-steps to do some sequential processing (including immediate weight changes) before seeing a new pattern of the input sequence. Although the architecture may influence the state of the environment within such a block of n_r time steps, the changes will not affect its input until the beginning of the next block.

With appropriate constant (time-invariant) $w_{ij}(t)$, simple conventional (threshold or semi-linear) activation functions f_k, sufficient n_h 'hidden' units, and sufficient block-size n_r, by repeated application of (1), the network can compute any function (or combination of functions)

$$f : \{0,1\}^{n_x+1+n_{eval}+n_o+n_{ana}+n_{mod}+1} \rightarrow \{0,1\}^{n_o+n_{ana}+n_{mod}+1} \tag{7}$$

computable within a constant finite number n_{cyc} of machine cycles by a given algorithm running on a given conventional digital (sequential or parallel) computer with limited temporal and storage resources[7]. This is because information processing in conventional computers can be described by the repeated application of boolean functions that can easily be emulated in recurrent nets as above.

With the particular set-up of section 2.2, at least the $\triangle$ output unit and the *val* input unit should take on not only binary values but real values. It is straight-forward, however, to show that the range $\{0,1\}$ in (7) may be replaced by $\mathbf{R}$ for any unit (by introducing appropriate simple activation functions).

[7] Or by a given Turing machine operating for limited time (during which it can use only a finite portion of its tape) — or by a given finite state automaton, for that matter.

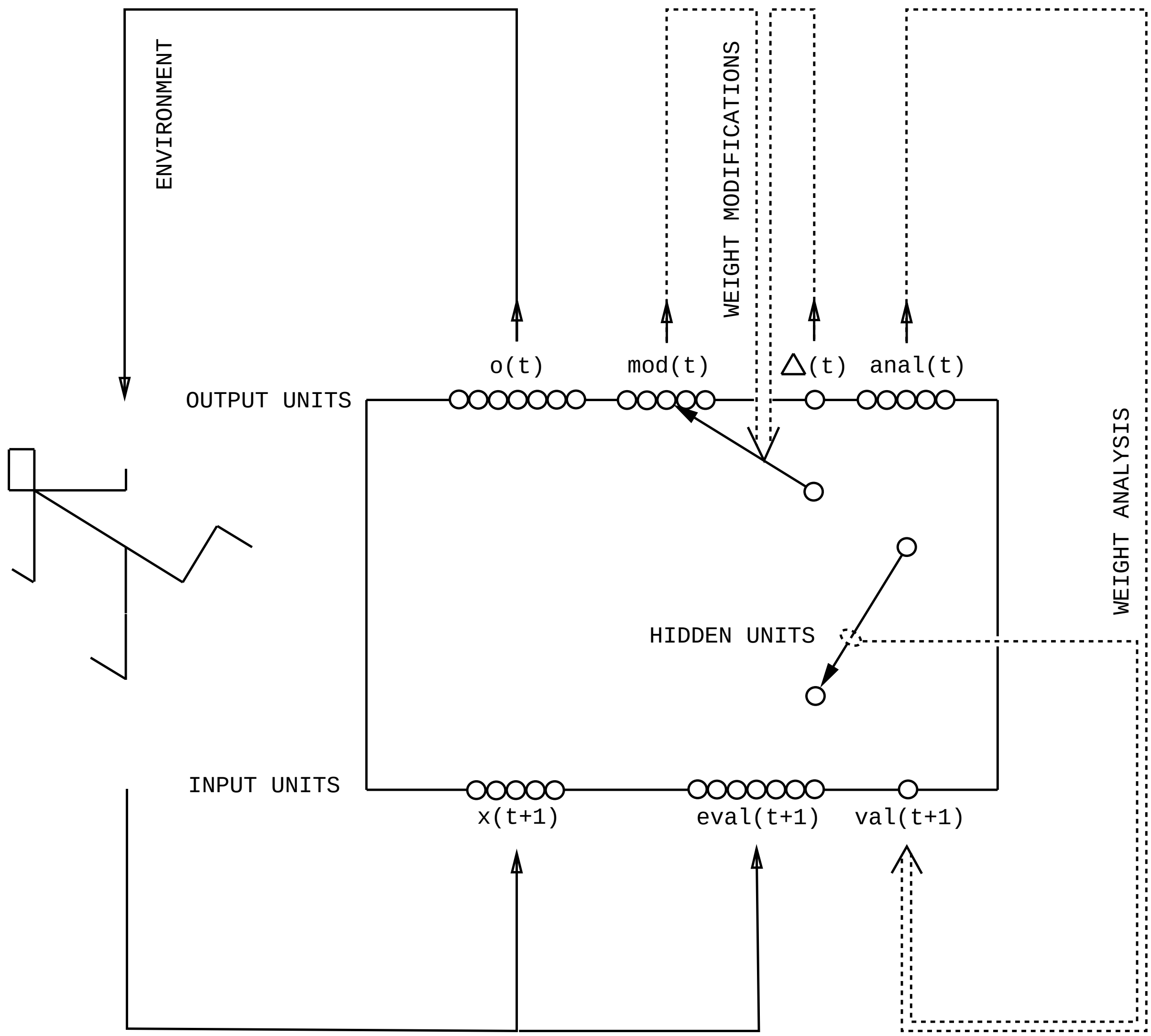

Figure 1: The special input vector $eval(t+1)$ informs the fully recurrent sequence-processing network (only two connections are shown) about external performance evaluations. The special outputs $ana(t)$ indicate which weights are to be read into the special input unit val. The special outputs $mod(t)$ and $\triangle(t)$ serve to compute immediate weight changes for the network's own connections. See text for details.

symbol	description
$\|v\|$	Euclidean length $\sqrt{v^T v}$ of vector v
t	time index, ranges from 1 to n_{time}
x_k	k-th 'normal' input unit
y_k	k-th non-input unit
o_k	k-th 'normal' output unit
z_k	k-th unit
$eval_k$	k-th eval unit (for observing performance evaluations)
ana_k	k-th analyzing unit (for addressing the net's own connections)
mod_k	k-th modifying unit (for addressing the net's own connections)
val	special input unit for analyzing current weight values
$\triangle$	special output unit for changing current weight values
$u(t)$	activation of u at time t, if u denotes unit
$v_k(t)$	k-th component of $v(t)$, if $v(t)$ vector (this is consistent with line above)
$x(t)$	agent's 'normal' input vector at time t
$o(t)$	agent's 'normal' output vector at time t
$eval(t)$	environments evaluation of performance at time t
$ana(t)$	special output vector indicating connection addresses
$mod(t)$	special output vector indicating connection addresses
n_x	$dim(x(t))$, number of 'normal' input units
n_o	$dim(o(t))$, number of 'normal' output units
n_{eval}	$dim(eval(t))$, number of eval units
n_I	$n_x + n_{eval} + 1$, number of *all* input units
$ceil(x)$	smallest integer $\geq x$
n_{mod}	$dim(mod(t)) = ceil(log_2 n_{conn})$, number of modifying units
n_{ana}	$dim(ana(t)) = ceil(log_2 n_{conn})$, number of analyzing units
n_y	$n_o + n_{ana} + n_{mod} + 1$, number of non-input units
n_h	number of hidden units
w_{ij}	connection from unit j to unit i
n_{conn}	$n_y(n_y + n_I)$, number of connections
$adr(w_{ij})$	address of w_{ij}
g	function defining 'closeness' of addresses, with narrow peak around origin
$val(t+1)$	$\sum_{i,j} g[\|ana(t) - adr(w_{ij})\|^2] w_{ij}(t)$
$w_{ij}(t)$	weight of w_{ij} at time t
$w_{ij}(t+1)$	$w_{ij}(t) + \triangle(t)\ g[\ \|adr(w_{ij}) - mod(t)\|^2\ ]$
n_r	constant block-size, $x(t)$ remains invariant during blocks of length n_r
n_s	number of blocks in sequence
$n_{time} = n_r n_s$	number of time steps in sequence
S	set of legal agent algorithms

Table 1: *Definitions of symbols describing the 'self-referential' architecture.*

We now can clearly identify the storage constraint n_h and the time constraint n_r with two parameters, without having to take care of any additional hardware-specific limitations constraining the computational power of the net[8].

We are left with a general, fixed-size, 'self-referential' network with unlimited computational power (within unavoidable time and storage constraints) that cannot only map arbitrary input sequences onto arbitrary output sequences but also can (in principle) analyze and change its own weight matrix, including those parts of the weight matrix that are responsible for analyzing and changing the weight matrix. The analyzing and modifying process itself may be arbitrarily complex. There are no theoretical limits on the sophistication of the weight changing algorithms implementable in the network. The same is true for the algorithm that changes the weight changing algorithm, and the algorithm that changes the algorithm that changes the weight changing algorithm, etc. This is because every '*meta*-level' is buried in the same network[9].

3 INITIAL LEARNING ALGORITHMS

To find useful 'self-modifying' learning algorithms, we need something like an initial learning algorithm. Certain aspects of the initial learning algorithm may not be modified. There is no way of making *everything* adaptive — for instance, algorithms leading to desirable environmental performance evaluations must always be favored over others. We may not allow the system to change this basic rule of the game. So the *hardwired* unchangeable aspects of the initial learning algorithm (alternatives will be given in the following subsections) will favor algorithms that modify themselves in a useful manner (a manner that leads to 'more desirable' evaluations).

An interaction sequence (see section 2.3) actually may be the concatenation of many 'conventional' training sequences for conventional recurrent networks. This will (in theory) help our 'self-referential' net to find regularities among solutions for *different* tasks.

For a variant of the 'self-referential' architecture, section 3.1 derives an exact gradient-based algorithm for supervised learning tasks. For another variant of the architecture, section 3.2 describes a more general but less informed and less complex reinforcement learning algorithm.

3.1 SUPERVISED LEARNING ALGORITHM

With supervised learning, $eval(t)$ provides information about the desired outputs at certain time steps. Arbitrary time lags may exist between inputs and later correlated outputs.

For the purposes of this section, f_k and g must be differentiable. This will allow us to compute gradient-based *directions* for the search in algorithm space.

In what follows, *unquantized variables are assumed to take on their maximal range.* For $o_k(t)$ there may be a target value, $d_k(t)$, specified at time t. Although it is of no significance whatsoever for the following derivation, in order not to limit the temporal resources of the net unnecessarily, target values $d_k(t)$ may occur only at the end of the n_s blocks with n_r time steps (see section 2.3). We set $n_{eval} = n_o$, which means that there are as many eval units as there are 'normal' output units. The current activation of a particular eval unit provides information about the error of the corresponding output unit at the previous time step (see

[8] It should be mentioned that since n_{conn} grows with n_h, n_{ana} and n_{mod} also grow with n_h (if they are not chosen large enough from the beginning). However, n_{ana} and n_{mod} grow much slower than n_h.

[9] *There is an alternative without explicit weight changing abilities.* Since recurrent nets are general purpose devices with essentially unlimited computational power (see section 2.3), they may run arbitrary algorithms, including arbitrary *learning* algorithms *represented in terms of activations instead of weights.* Therefore we may use a conventional recurrent net to obtain a simpler 'self-referential' system than the one this paper focuses on, by renouncing the relatively complex, explicit weight-modifying capabilities (involving $ana(t), val(t), mod(t), \triangle(t)$). This alternative system, however, *cannot* explicitly manipulate all its adaptive parameters. Only a tiny fraction of the system variables, namely the activations (as opposed to activations *and* weights), are accessible to self-manipulation. Still, the system is (at least in theory) able to keep self-modifying aspects of its algorithm in its *activations.* It is important to keep one of the above-mentioned modifications of conventional recurrent nets, however: We have to reserve a special set of input units for feeding back evaluations $eval(t)$ such that the network receives all the information about its successes and failures.

equation (11)). We assume that inputs and target values do not depend on previous outputs (via feedback through the environment).

I will focus on the architecture described in sections 2.1 and 2.2 — the corresponding learning algorithm for the simpler architecture without explicit weight changing capabilities (footnote 9, section 2.3) is just a modification of conventional gradient-based algorithms for recurrent nets (e.g. [10], [17], [6], [7], [19]). To obtain a better overview, let us summarize the system dynamics in compact form:

$$net_{y_k}(1) = 0, \quad \forall t \geq 1: \quad x_k(t) \leftarrow environment, \quad y_k(t) = f_{y_k}(net_{y_k}(t)),$$

$$\forall t > 1: \quad net_{y_k}(t) = \sum_l w_{y_k l}(t-1) l(t-1), \tag{8}$$

$$\forall t \geq 1: \quad w_{ij}(t+1) = w_{ij}(t) + \triangle(t)\ g[\ \|adr(w_{ij}) - mod(t)\|^2\ ], \tag{9}$$

$$val(1) = 0, \quad \forall t \geq 1: \ val(t+1) = \sum_{i,j} g[\|ana(t) - adr(w_{ij})\|^2] w_{ij}(t), \tag{10}$$

The following aspect of the system dynamics is specific for supervised learning and therefore has not yet been defined in previous sections:

$$eval_k(1) = 0, \quad \forall t \geq 1: eval_k(t+1) = d_k(t) - o_k(t) \quad if\ d_k(t)\ exists,\ and\ 0\ else. \tag{11}$$

Objective function. As with typical supervised sequence-learning tasks, we want to minimize

$$E^{total}(n_r n_s), \quad where\ E^{total}(t) = \sum_{\tau=1}^{t} E(\tau), \quad where\ E(t) = \frac{1}{2}\sum_k (eval_k(t+1))^2.$$

Note that elements of algorithm space are evaluated solely by a conventional evaluation function[10].

The following algorithm for minimizing E^{total} is partly inspired by (but more complex than) conventional recurrent network algorithms (e.g. [10], [17], [6], [7], [19]).

Derivation of the algorithm. We use the chain rule to compute weight increments (to be performed *after* each training sequence) for all *initial* weights $w_{ab}(1)$ according to

$$w_{ab}(1) \leftarrow w_{ab}(1) - \eta \frac{\partial E^{total}(n_r n_s)}{\partial w_{ab}(1)}, \tag{12}$$

where η is a constant positive 'learning rate'. Thus we obtain an *exact* gradient-based algorithm for minimizing E^{total} under the 'self-referential' dynamics given by (8)-(11). To reduce writing effort, I introduce some short-hand notation partly inspired by [18]:

For all units u and all weights w_{ab} we write

$$p_{ab}^{u}(t) = \frac{\partial u(t)}{\partial w_{ab}(1)}. \tag{13}$$

For all pairs of connections (w_{ij}, w_{ab}) we write

$$q_{ab}^{ij}(t) = \frac{\partial w_{ij}(t)}{\partial w_{ab}(1)}. \tag{14}$$

[10] It should be noted that in quite different contexts, previous papers have shown how 'controller nets' may learn to perform appropriate lasting weight changes for a second net [14][5]. However, these previous approaches could not be called *'self-referential'* — they all involve at least some weights that can *not* be manipulated other than by conventional gradient descent. Attempting to solve a different kind of task, the system described in [9] allows *any* weight in the system to be manipulated dynamically, but since new units must be created to do this, only a fraction of the total number of weights can ever be modified at any time.

To begin with, note that

$$\frac{\partial E^{total}(1)}{\partial w_{ab}(1)} = 0, \quad \forall t > 1: \ \frac{\partial E^{total}(t)}{\partial w_{ab}(1)} = \frac{\partial E^{total}(t-1)}{\partial w_{ab}(1)} - \sum_k eval_k(t+1) p_{ab}^{o_k}(t). \tag{15}$$

Therefore, the remaining problem is to compute the $p_{ab}^{o_k}(t)$, which can be done by incrementally computing all $p_{ab}^{z_k}(t)$ and $q_{ab}^{ij}(t)$, as we will see.

At time step 1 we have

$$p_{ab}^{z_k}(1) = 0. \tag{16}$$

Now let us focus on time steps after the first one. For $t \geq 1$ we obtain the recursion

$$p_{ab}^{x_k}(t+1) = 0, \tag{17}$$

$$p_{ab}^{eval_k}(t+1) = -p_{ab}^{o_k}(t), \quad \textit{if } d_k(t) \textit{ exists, and } 0 \textit{ otherwise}, \tag{18}$$

$$p_{ab}^{val}(t+1) = \sum_{i,j} \frac{\partial}{\partial w_{ab}(1)} [\ g(\|ana(t) - adr(w_{ij})\|^2) w_{ij}(t)\ ] =$$

$$\sum_{i,j} \{\ \ q_{ab}^{ij}(t) g[\|ana(t) - adr(w_{ij})\|^2)] +$$

$$w_{ij}(t)\ [\ \ g'(\|ana(t) - adr(w_{ij})\|^2) 2 \sum_m (ana_m(t) - adr_m(w_{ij})) p_{ab}^{ana_m}(t)\ \ ]\ \ \} \tag{19}$$

(where $adr_m(w_{ij})$ is the m-th bit of w_{ij}'s address),

$$p_{ab}^{y_k}(t+1) = f'_{y_k}(net_{y_k}(t+1)) \sum_l \frac{\partial}{\partial w_{ab}(1)} [l(t) w_{y_k l}(t)] =$$

$$f'_{y_k}(net_{y_k}(t+1)) \sum_l w_{y_k l}(t) p_{ab}^l(t) + l(t) q_{ab}^{y_k l}(t), \tag{20}$$

where

$$q_{ab}^{ij}(1) = 1 \ \textit{if } w_{ab} = w_{ij}, \textit{ and } 0 \textit{ otherwise}, \tag{21}$$

$$\forall t > 1: \ \ q_{ab}^{ij}(t) = \frac{\partial}{\partial w_{ab}(1)} \left[w_{ij}(1) + \sum_{\tau < t} \triangle(\tau) g(\|mod(\tau) - adr(w_{ij})\|^2) \right] =$$

$$q_{ab}^{ij}(t-1) + \frac{\partial}{\partial w_{ab}(1)} \triangle (t-1) g(\|mod(t-1) - adr(w_{ij})\|^2) =$$

$$q_{ab}^{ij}(t-1) + p_{ab}^{\triangle}(t-1) g(\|mod(t-1) - adr(w_{ij})\|^2) +$$

$$2 \triangle (t-1)\ g'(\|mod(t-1) - adr(w_{ij})\|^2) \sum_m [mod_m(t-1) - adr_m(w_{ij})] p_{ab}^{mod_m}(t-1). \tag{22}$$

According to equations (16)-(22), the $p_{ab}^j(t)$ and $q_{ab}^{ij}(t)$ can be updated incrementally at each time step. This implies that (15) can be updated incrementally at each time step, too. The storage complexity is independent of the sequence length and equals $O(n_{conn}^2)$. The computational complexity per time step is $O(n_{conn}^2 log n_{conn})$.

Why is there no endless recursion? Because the algorithm does not really throw away the concept of gradient descent — actually it performs gradient descent in the initial weights (at the beginning of each training sequence). That's where the recursion stops[11].

[11] Section 5 will mention a simple alternative, however.

Again: The initial learning algorithm uses gradient descent to find weight matrices that minimize a conventional error function. This is partly done just like with conventional recurrent net weight changing algorithms. Unlike with conventional networks, however, the network algorithms themselves may choose to change some of the network weights in a manner unlike gradient descent (possibly by doing something smarter than that) — but only if this helps to minimize E^{total}. In other words, the 'self-referential' aspects of the architecture *may* be used by certain 'self-modifying' algorithms to generate desirable evaluations. Therefore the whole system may be viewed as a 'self-referential' *augmentation* of conventional recurrent nets. Further speed-ups (like the ones in [18], [19], [13]) are possible but not essential for the purposes of this paper.

3.2 A REINFORCEMENT LEARNING ALGORITHM

If we want to be *really* general, then we have to focus on *reinforcement learning* tasks instead of *supervised learning* tasks. With *general* reinforcement learning, $o(t)$ influences the environmental state and $x(t)$, and $eval(t)$ is a scalar or vector, real-valued or binary *reinforcement signal*. Note that the system may choose its own inputs and training examples. The agent's goal is to maximize $\sum_t \|eval(t)\|^2$. The *'credit assignment problem'* may be of arbitrary complexity. Arbitrary time lags may exist between actions and later consequences. Unlike with supervised learning, no 'teacher' tells the network which 'normal' outputs to produce at which time step. There is not even the assumption of a *'Markovian'* interface between agent and environment [12]. As a consequence, reinforcement learning algorithms inspired by dynamic programming (e.g. [1]) will be of no use. This is about the most general setting I can think of.

General settings require general initial learning schemes. The following scheme is a general but simple one — it pays for its generality by starting out with much less informed weight-changes than the gradient-based search of section 3.1.

To allow for non-deterministic 'exploratory' behavior, I introduce binary[12] probabilistic units by defining the f_k as follows: $f_\triangle$ is a function that returns values between -1 and 1. For $k \neq \triangle$, $f_k(x)$ returns 1 with probability $f^*(x)$, and 0 with probability $1 - f^*(x)$, where

$$f^*(x) = \frac{1}{1+e^{-x}}. \tag{23}$$

Here is a simple, but safe and general, hardwired initial learning scheme.

1. Before training, the initial weights are defined as the current best algorithm. The current best evaluation is defined as minus infinity.

2. Randomly perturb the weights of the current best algorithm to obtain the current algorithm.

3. Test the current algorithm by evaluating its performance on test data.

4. If the evaluation is higher than the current best evaluation, then set the current best evaluation equal to the current evaluation, and set the current best algorithm equal to the current algorithm.

5. Go to step 2.

Obviously, the scheme does nothing but simple 'guided' random search in algorithm space, always keeping and mutating the best 'self-referential' algorithm so far. In the long run, the performance can only improve. I do not believe that there is a significantly better initial learning algorithm for *general* environments.

The expectation is, of course, that in environments with (initially unknown) regularities, the system will eventually discover relationships between similar learning problems, and the self-modifying capabilities of the architecture will eventually lead to *non*-random and more informed problem-specific weight-changes. This expectation is justified because algorithms creating 'smart' weight changes (self-modifications) will lead faster to higher evaluations than 'dumb' algorithms, thus being favored over the 'dumb' ones.

[12] As a consequence, the binary weight addressing scheme of section 2.2 actually *is* the most compact one. See footnote 6, section 2.2.

4 CONCLUDING REMARKS

In a nutshell, the network I have described can, besides learning to solve problems posed by the environment, also use its own weights as input data and can learn new algorithms for modifying its weights in response to the environmental input and evaluations. This effectively embeds a chain of *'meta-networks'* and *'meta-meta-...-networks'* into the network itself.

Bias. This universe does not allow us a bias-free learning architecture. The scheme above includes a lot of bias: The selection of the basic architecture (indeed, one can find many different 'self-referential' architectures, once one starts looking for them), the scheme for addressing connections and talking about algorithm components, the hardwired basic learning algorithm, etc. The system I have described above certainly does not reflect the most practical bias for typical learning tasks in typical real-world environments. It should instead demonstrate that there are ways of making *all* adaptive parameters of learning systems accessible to self-manipulation, and that it is possible to come up with reasonable hardwired learning algorithms for picking 'good' self-manipulating algorithms. The system described herein serves simply as a single example of many similar systems that have the following two things in common: Universal computational power (constrained only by unavoidable time and storage limitations), and explicit access to *all* modifiable algorithm components.

Biological plausibility. It seems that I cannot explicitly tell each of my 10^{15} synapses to adopt a certain value. I seem able only to affect my own synapses indirectly — for instance, by somehow actively creating 'keys' and 'entries' to be associated with each other. Therefore, at first glance, the neural net in my head appears to embody a different kind of self-reference than the artificial net of section 2.1 and 2.2[13]. But does it really? The artificial net also does not have a concept of its n-th weight. All it can do is to find out how to talk about weights in terms of activations — without really knowing what a weight is (just like humans who did not know for a long time what synapses are). Therefore I cannot see any evidence that brains use fundamentally different kinds of 'introspective' algorithms. On the other hand, I am not aware of any biological evidence supporting the theory that brains have some means for addressing single synapses by creating appropriate activation patterns [14].

Ongoing and future research. Due to the complexity of the activation dynamics of the 'self-referential' network, one would expect the error function derived in section 3.1 to have many local minima. [15] describes a variant of the basic idea (involving a biologically more plausible weight manipulating strategy) which is less plagued by the problem of local minima (and whose initial learning algorithm has lower computational complexity than the one from section 3.1).

A major criticism of the learning algorithms in section 3 is that they are based on the concept of fixed interaction sequences. All the hard-wired learning algorithms do is find *initial* weights leading to 'desirable' cumulative evaluations. After each interaction sequence, the *final* weight-matrix (obtained through self-modification) is essentially thrown away. *A simple alternative would be to run (after each interaction sequence) the final weight matrix against the best algorithm so far and keep it if it is better*[15]. Again, performance cannot get worse but can only improve over time. I would, however, prefer a hypothetical 'self-referential' learning system that is not initially based on the concept of training sequences at all. Instead, the system should be able to learn to actively segment a single continuous input stream into useful training sequences. Future research will be directed towards building provably working, hard-wired initial-learning-algorithms for such hypothetical systems.

Although the systems described in this paper have a mechanism for 'self-referential' weight changes, they must still learn to use this mechanism. Experiments are needed to discover how practical an approach this

[13] Perhaps an even *simpler* kind of self-reference, as with the alternative network of footnote 9, section 2.3.

[14] As mentioned before, however, this paper does not insist on addressing every weight in the system individually. (See again footnote 6.) There are many alternative, sensible ways of choosing g and redefining equations (3) and (5) (e.g. [15]).

[15] With the algorithms of section 3, the weight changes for the *initial* weights (at the beginning of a training sequence) are hard-wired. The alternative idea of testing the final weight matrix (at the end of some sequence) against the best previous weight matrix corresponds to the idea of letting the system change its initial weights, too. With the alternative network from footnote 9, section 2.3, this would amount to *not* resetting the activations of the net for the test phase following each training sequence. This is essential, because the *activations* at the end of a sequence might represent a useful 'self-referential' learning algorithm (running on a system with essentially constant weights).

is. This paper[16], however, presents a thought experiment and does not focus on experimental evaluations; it is intended only to show the theoretical possibility of certain kinds of 'self-referential' weight change algorithms. Experimental evaluations of alternative 'self-referential' architectures (with alternative more practical self-addressing schemes, e.g. [15]) will be left for the future.

5 ACKNOWLEDGEMENTS

Thanks to Mark Ring, Mike Mozer, Daniel Prelinger, Don Mathis, and Bruce Tesar, for helpful comments on drafts of this paper. This research was supported in part by a DFG fellowship to the author, as well as by NSF PYI award IRI–9058450, grant 90–21 from the James S. McDonnell Foundation, and DEC external research grant 1250 to Michael C. Mozer.

References

[1] A. G. Barto. Connectionist approaches for control. Technical Report COINS Technical Report 89-89, University of Massachusetts, Amherst MA 01003, 1989.

[2] D. Chalmers. The evolution of learning: An experiment in genetic connectionism. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, editors, *Proc. of the 1990 Connectionist Models Summer School*, pages 81–90. San Mateo, CA: Morgan Kaufmann, 1990.

[3] D. Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

[4] D. Lenat. Theory formation by heuristic search. *Machine Learning*, 21, 1983.

[5] K. Möller and S. Thrun. Task modularization by network modulation. In J. Rault, editor, *Proceedings of Neuro-Nimes '90*, pages 419–432, November 1990.

[6] B. A. Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269, 1989.

[7] F. J. Pineda. Time dependent adaptive neural networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 710–718. San Mateo, CA: Morgan Kaufmann, 1990.

[8] L. Y. Pratt. Non-literal transfer of information among inductive learners. In R. J. Mammone and Y. Y. Zeevi, editors, *Neural Networks: Theory and Applications*, volume 2. 1992. In press.

[9] M. B. Ring. Incremental development of complex behaviors through automatic construction of sensory-motor hierarchies. In L. Birnbaum and G. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop*, pages 343–347. Morgan Kaufmann, 1991.

[10] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

[11] J. H. Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook, 1987. Institut für Informatik, Technische Universität München.

[12] J. H. Schmidhuber. Reinforcement learning in markovian and non-markovian environments. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 500–506. San Mateo, CA: Morgan Kaufmann, 1991.

[16] This paper is partly inspired by some older ideas about 'self-referential learning' — [11] describes a 'self-referential' genetic algorithm, as well as a few other 'introspective' systems.

[13] J. H. Schmidhuber. A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation*, 4(2):243–248, 1992.

[14] J. H. Schmidhuber. Learning to control fast-weight memories: An alternative to recurrent nets. *Neural Computation*, 4(1):131–139, 1992.

[15] J. H. Schmidhuber. On decreasing the ratio between learning complexity and number of time varying variables in fully recurrent nets. Technical report, Dept. of Comp. Sci., University of Colorado at Boulder, 1992. In preparation.

[16] P. Utgoff. Shift of bias for inductive concept learning. In *Machine Learning*, volume 2. Morgan Kaufmann, Los Altos, CA, 1986.

[17] P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1, 1988.

[18] R. J. Williams. Complexity of exact gradient computation algorithms for recurrent neural networks. Technical Report Technical Report NU-CCS-89-27, Boston: Northeastern University, College of Computer Science, 1989.

[19] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In *Back-propagation: Theory, Architectures and Applications*. Hillsdale, NJ: Erlbaum, 1992, in press.