On the Need for a Neural Abstract Machine

Diego Sona, Alessandro Sperduti

Dipartimento di Informatica
Università di Pisa

1 Introduction

The complexity of learning tasks and their variety, as well as the number of different neural networks models for sequence learning is quite high. Moreover, in addition to architectural details and training algorithms peculiarities, there are other relevant factors which add complexity to the management of a neural network for the adaptive processing of sequences. For example, training heuristics, such as adaptive learning rates, regularization, and pruning, are very important, as well as insertion of a priori domain knowledge. All these issues must be considered and matched with the complexity of the application domain at hand. This means that the successful application of a neural network to a real world domain has to answer to several questions on the type of architecture, training algorithms, training heuristics, and knowledge insertion, according to the problem complexity.

At present, these questions cannot be easily answered, due to the lack of a computational tool encompassing all the relevant issues. We observe that some authors (Nerrand et al., 1993) (Tsoi, 1998b) (Wan and Beaufay, 1998) (Berthold and Fischer, 1997) (Fischer et al., 1998a) (Koch et al., 1998) (Fischer et al., 1998b) (Frasconi et al., 1998) tried to unify different architectures and learning algorithms. However, none of their proposals is complete. The same situation is encountered when considering software simulators and neural specification languages: all of them are restricted to specific models and do not allow the user to develop new models.

On the basis of these observations, we argue for the need for a Neural Abstract Machine, i.e., a formal, and precise definition of the basic (and relevant) objects as well as operations which are manipulated and performed, respectively, by neural computation.

To define the Neural Abstract Machine, we suggest to use Abstract State Machines (ASMs). ASMs have been extensively used for the formal design and analysis of various hardware systems, algorithms and programming languages semantics. They allow to specify formal systems in a very simple way, while preserving mathematical soundness and completeness.

In Section 2 we briefly review the different issues arising when considering learning sequences. On the basis of this review, in Section 3, we argue about the need for a Neural Abstract Machine. Abstract State Machines are briefly presented in Section 4 and a very simple example of how they can be applied

to neural networks is discussed in Sections 5 and 6. Conclusions are drawn in Section 7.

2 A Brief Overview on Sequence Learning by Neural Networks

In the following we will briefly outline the main issues in sequence learning by neural networks. The presence of a large number of different neural architectures and learning algorithms is pointed out. The main computational and complexity known results on architecture power and learning are discussed. The most important issues concerning training heuristics and knowledge insertion in recurrent networks are briefly reported. Moreover, we argue about the difficulties in developing successful neural solutions for application problems.

2.1 Domains, Tasks, and Approaches to Sequence Learning

Informally, a sequence is a serially ordered set of atomic entities. Sequences (the simplest kind of dynamic data structure) typically occur in learning domains with temporal structure, where each atom corresponds to a discrete *time* point. For example, variables in a financial forecasting problem are sampled at successive instants, yielding an instance space formed by discrete-time sequences of observations. Automatic speech recognition systems contain front-end acoustic modules that learn to translate sequences of acoustic attributes into sequences of phonetic symbols. Other examples of temporal data can be found in problems of automatic control or digital signal processing. In all these cases, data gathering involves a digital sampling process. Hence, serial order in sequences is immediately associated with the common physical meaning of time. There are however other kinds of data that can be conveniently represented as sequences. For example, consider a string of symbols obtained after preprocessing in syntactic approaches to pattern recognition. Also consider problems of molecular biology, in which DNA chains are represented as strings of symbols associated to protein components. Such strings can also be effectively represented as sequences, although *time* in these cases do not play any role in a strictly physical meaning. Time, in the domain of sequences, has therefore the more abstract meaning of *coordinate* used to address simple entities which are *serially ordered* to form a more complex structure. More complex situations arises when considering sequences combining both symbolic and numerical data, as may happen in medical applications.

There are different learning tasks involving sequences. Typical tasks are classification, time series prediction (with different order of prediction), sequential transduction, and control. Sometimes it is also useful to try to approximate probability distributions over sequence domains, as well as to discover meaningful clusters of sequences. This is particularly useful when performing Knowledge Discovery and Data Mining.

The complexity and variety of problems in sequence learning is so high that it would be naive to think that a single approach suffices to master the field. According to the nature of data and learning tasks, different approaches has been defined and explored, such as Recurrent Neural Networks, Hidden Markov Models, Reinforcement Learning (dynamic programming), Evolutionary Computation, Rule-based Systems, Fuzzy Systems, and so on. Recently, the feeling that a combination of more approaches to face real-world problems is needed is emerging in the scientific community. Unfortunately, foundations on how to rigorously proceed with this combination have yet to emerge.

2.2 Representations and RNN Architectures

Neural networks architectures for sequence learning can be broadly classified into two classes, according to the way time is represented: *explicitly* or *implicitly*.

Explicit time representation is also referred to as algebraic representation of time, since input and output events at different time steps are explicitly represented as unrelated variables on which an algebraic model operates, i.e., the *whole* input subsequence from time 1 to time t is mapped into the output using a *static* relationship. From a practical point of view, this means that a buffer holding the external inputs to the system, as they are received, must be used. Basically, with an explicit representation of time, *temporal* processing problems are converted into *spatial* processing problems, thus allowing one to use simpler static models, such as feed-forward neural networks. Typical architectures belonging to this class of networks are feed-forward networks looking at the input sequences through a window of prefixed size and Time-Delay networks, which exploit this window approach also for hidden activations. The networks in this class have been related with FIR filters, since they can be considered as nonlinear versions of these filters. Moreover, from a computational point of view, this class of networks is strictly related to Definite-Memory Sequential Machines.

Implicit time representation assumes *causality*, i.e., the output at time t only depends on the present and past inputs. If causality holds, then the *memory* about the past can be stored into an *internal state*. From a practical point of view, internal representations can be obtained by *recurrent connections*. To this class of networks belong Fully Connected networks, NP networks, NARX networks, Recurrent Cascade Correlation networks, and so on. According to the type of topology involving the recurrent connections, different types of memory can be implemented (e.g., input (transformed) memory, hidden (transformed) memory, output (transformed) memory). Moreover, in discrete-time networks, different kinds of temporal dependencies can be expressed by resorting to different Discrete-Time Operators, such as the standard *shift operator*, the *delta operator*, the *gamma operator*, the *rho operator*, and so on. Because of the internal state, this class of networks is strictly related to IIR filters, and to several classes of sequential machines (such as Finite State Sequential Machines, Finite-Memory Sequential Machines, etc.). A good overview of all these different architectural aspects can be found in (Tsoi, 1998b).

2.3 Training Algorithms

There is a huge variety of training algorithms for recurrent neural networks. Almost all training algorithms for recurrent neural networks are based on gradient descent. Among these the most popular algorithms are Back-propagation Through Time (BPTT) (McClelland and Rumelhart, 1987), Real Time Recurrent Learning (RTRL) (Williams and Zipser, 1988) developed for on-line training, Kalman (Extended) Filter (EKF) (Williams, 1992) (Puskorius and Feldkamp, 1994), and Temporal Difference (Sutton, 1988) (Tesauro, 1992). As for feed-forward networks (Battiti, 1992), second order or quasi-second order methods can be defined for recurrent neural networks (see (Tsoi, 1998a) for an overview).

Moreover, there is a class of constructive algorithms which exploit the gradient to build up the network architecture during training, according to the training data complexity. Within this class we can mention Recurrent Cascade-Correlation (Fahlman, 1991), a partition algorithm using Radial Basis Functions (Tsoi and Tan, 1997), and Recurrent Neural Trees (Sperduti and Starita, 1997).

There is also a class of stochastic learning algorithms for recurrent networks. For example, EM (Dempster et al., 1977) (or GEM) can be used to train feed-forward (Amari, 1995) and recurrent networks (Ma and Ji, 1998). Also Evolutionary Algorithms (Genetic Algorithms) can be used to train recurrent networks (Saunders et al., 1994) (Angeline et al., 1994). Notice that Evolutionary Algorithms can be considered constructive algorithms, since with a suitable representation of the recurrent network, more and more complex networks can evolve within a population of networks.

2.4 Training Heuristics

The successful training of a neural network can not usually be obtained by just running the selected training algorithm on any configuration for the network architecture and learning parameters. There are several additional issues which must be considered. This is particularly true for recurrent networks.

For example, the size of the state variable is very relevant. Moreover, an important issue is which kind of delay lines should be used in the network, since a correct choice may help in capturing the right temporal dependencies, hidden into the training data. So usually it is useful to have multi-step delay lines within the recurrent network.

In addition, in order to facilitate training, it may be important to choose the right representation for the starting state or even to have the possibility to learn it (Forcada and Carrasco, 1995). Also connectivity can be very critical: fully connected networks allow for the discovery of high-order correlations, while a sparse connectivity can significantly speed up training and return very good solutions where high-order correlations are not relevant. Similarly, training times can be reduced by using a learning rule which exploits truncated gradients, or by using an adaptive learning rate.

Finally, in order to have some guarantee that the trained network will show some generalization capability, regularization (Wu and Moody, 1996) (Tsoi, 1998a) and/or pruning (during and post training) (Giles and Omlin, 1994) (Pedersen and Hansen, 1995) (Tsoi, 1998a) should be used.

These are just some of the issues which must be taken in consideration when training a recurrent neural network. Thus training a recurrent network is not just a problem of choosing a suitable architecture and learning algorithm: several different heuristics should be applied to fill in the missing information about the learning task.

2.5 Computational Power and Learning Facts

Training heuristics are useful, however it is important to discover computational and complexity limitations and strengths of network architectures and learning algorithms, since these may help us in avoiding to loose time and resources with computational devices which are not suited for the learning task at hand. Several results concerning computational power and learning complexity for recurrent neural networks have be obtained.

Concerning computational power, among positive results we can mention that recurrent networks can model any first order discrete-time, time invariant, non-linear system (see for example (Seidl and Lorenz, 1991) (Sontag, 1993)). In addition, it has been observed that recurrent neural networks with just 2 neurons can exhibit chaos (Tino et al., 1995) (Casey, 1996). This last result is interesting since it testifies that even a trivial network can show a very complex behavior, thus implicitly demonstrating that, in principle, very complex computations could be performed by this network. Finally, some recurrent architectures, such as fully recurrent and NARX networks, are Turing equivalent (Siegelmann and Sontag, 1991) (Siegelmann and Sontag, 1995) (Siegelmann et al., 1997).

On the other side, some recurrent architectures have limited computational power. For example, single layer recurrent networks cannot represent all Finite State Automata (Goudreau et al., 1993) (while Elman networks can because of the presence of the output layer (Kremer, 1995)). Similarly, some constructive methods for RNN (i.e., Recurrent Cascade Correlation) generate networks which are computationally limited (Giles et al., 1995) (Kremer, 1996).

Concerning learning, it has been proved that gradient descent based algorithms for RNNs converge for bounded sequences and constraints on learning rate (Kuan et al., 1994). Unfortunately, however, the loading problem (Judd, 1989) for RNNs (i.e., finding a set of weights consistent with the training data) is unsolvable (Wiklicky, 1994)! Moreover, even if several recurrent architectures have the computational capability to represent arbitrary nonlinear dynamical systems, gradient-based training suffers long-term dependencies (Bengio et al., 1994) (Lin et al., 1996), showing difficulties in learning even very simple dynamical behaviors. The problem of long-term dependencies can be understood as the inability of gradient descent algorithms (when used on several of the most common RNN architectures) to store error information concerning past inputs which are far in time from the present input. Some heuristics have been proposed to

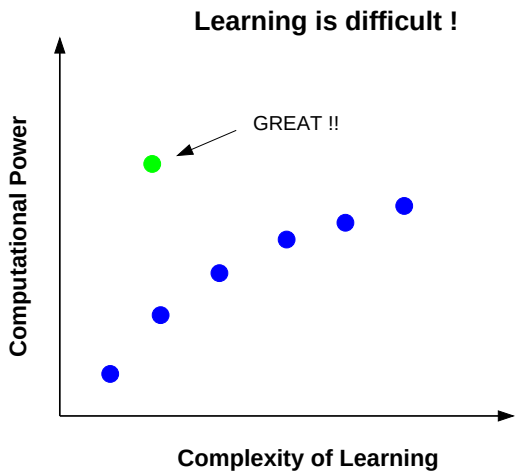


Fig. 1. Learning is difficult: as soon as the computational power of the RNN increases, the complexity of training the network increases exponentially. It would be great to have a powerful RNN which is easy to train.

try to reduce the problem of vanishing gradient information (Giles and Omlin, 1993a) (Schmidhuber, 1992) (Lin et al., 1996) (Hochreiter and Schmidhuber, 1997), however, none of them is able to completely remove it.

2.6 Knowledge Insertion and Refinement

From Section 2.5 it is clear that RNNs which are computationally very powerful are also very difficult to train. In fact, it would be wonderful to have a recurrent architecture which is computational complete, i.e., Turing equivalent, *and* also easy to train (see Figure 1).

Some authors have proposed to exploit a priori information on the application domain to master learning complexity. From the point of view of generalization, this idea is supported by theoretical results on the decomposition of the error of a neural network into Bias and Variance (Geman et al., 1992). These results suggest that in order for a neural network to properly learn the desired function, some significant prior structure should be given to the network.

There are different ways to implement the above idea. One possibility is to give some *hints* to the network on specific properties of the desired function (see for example (Abu-Mostafa, 1990) (Al-Mashouq and Reed, 1991) (Simard et al., 1992) (Abu-Mostafa, 1993b) (Abu-Mostafa, 1993a) (Abu-Mostafa, 1995b) (Abu-Mostafa, 1995a)) or to *insert* prior knowledge in form of rules into the neural network and then to train it using a standard learning algorithm (for the case of dynamically-driven recurrent neural networks see (Das et al., 1992) (Frasconi et al., 1991) (Alquézar and Sanfeliu, 1995) (Frasconi et al., 1995) (Omlin and Giles, 1996)). A variant of this approach considers the possibility to *refine*

rough knowledge (see for example (Maclin and Shavlik, 1992) (Giles and Omlin, 1993b) (Giles and Omlin, 1993a)) by *extracting* knowledge coded into the neural networks through algorithms which take in input a neural network and return a set of rules or a FSM for sequence domains (see for example (Omlin et al., 1992) (Giles et al., 1992) (Towell and Shavlik, 1993) (Casey, 1996)). These rules are then inserted back into the neural network and the cycle insertion/training/extraction is repeated several times.

It is usually believed that the above approach helps learning since it may reduce the number of functions that are candidate for the desired function. Moreover, due to the inserted knowledge, training times should be reduced. Unfortunately, while the above statements may be true for specific and (often) small domains, in general there are at least two reasons for the above approach to fail. First of all, the insertion of rules or FSM into neural networks implies that a good amount of the network structure is predetermined, as well as several of the values for the weights. This turns out to create some difficulties to the learning process (especially if the learning algorithm is based on gradient descent), which has to satisfy the additional constraints imposed by the knowledge insertion, i.e., very often the neural units are saturated and thus they are difficult to train by using a gradient descent approach. Moreover, rule (or DFA) extraction from neural networks is not so easy as it was expected. For example, some criticisms about the reliability of DFA extraction from recurrent neural networks have been raised in (Kolen, 1994). Although a partial solution to these criticisms has been given in (Wiles and Bollard, 1996), the problem of reliable extraction of knowledge from neural networks is still a research subject. For example, recently, an approach based on reinforcement learning for extracting complete action plans from sequences has been proposed (Sun and Session, 1998)

2.7 Application Requirements

Application fields are many and diverse. This diversity implies the need for different approaches, ranging from symbolic to sub-symbolic techniques. Given a specific problem, several are the questions which need to be answered for a successful, flexible, and portable solution. When using RNNs, some typical questions are:

What is the appropriate RNN architecture(s) for the problem to be solved ?
What is the appropriate RNN training algorithm(s) ?
How can a priori knowledge be best used ?
What to do if no existing architecture/algorithm is suited for the problem to be solved (development of new architecture/algorithm ?)
How can a RNN be integrated with other approaches ?

The possibility to give correct answers to these questions in a short time is related to the availability of specification languages for prototyping and experimentation. Unfortunately, it must be stressed that even if many neural network simulators have been developed (e.g., Aspirin/Migraines, Rochester Connectionist Simulator, NNSYSID, Stuttgart Neural Network Simulator, Toolkit

for Mathematica, just to mention a few), as well as specification languages for neural networks (e.g., EpsiloNN, Neural Simulation Language), they implement a restricted set of specific models for dealing mostly with static data. At our knowledge, there is no single specification language which may support the user in giving an answer to all the above questions, especially when considering the development of new architectures and/or algorithms. The situation is even worst when considering the integration of RNNs with symbolic approaches.

2.8 Unifying Theories

The lack of a "universal" neural specification language is mainly due to a lack of synthesis of the main concepts and results in the neural network field. Up to now, several different architectures and training algorithms have been devised. Many of the new improvements however are just small and insignificant changes to existing architectures and/or algorithms. Furthermore, since the research is not based on a general framework, it is difficult to focus on the study of background important properties. For this reason some researchers have felt the need to try to develop, especially for dynamical systems, a general framework able to describe the foundations of both architectures and learning algorithms.

Nerrand et al. (Nerrand et al., 1993) describe a general framework that encompasses algorithms for feed-forward and recurrent neural networks, and algorithms for parameter estimation of non-linear filters. Specifically, feed-forward networks are viewed as transversal filters, while recurrent networks as recursive filters. Their approach is based on the definition of a canonical form which can be used as a building block for the training algorithms based on gradient estimation. A similar approach is followed also by Santini et al. (Santini et al., 1995).

Tsoi and Back (Tsoi and Back, 1997) and Tsoi (Tsoi, 1998b) propose a unifying view of discrete time feed-forward and recurrent network architectures, basing their work on systems theory with linear dynamics. In this case canonical forms are used to group similar architectures and a unifying description is obtained by exploiting a notation based on matrices.

Wan and Beaufays (Wan and Beaufays, 1996) (Wan and Beaufay, 1998) suggest an approach, exploiting *flow graph theory*, to construct and manipulate block diagrams representing neural networks. Gradient algorithms for temporal neural networks are derived on the basis of a set of simple block diagram manipulation rules.

A computational approach is followed by Berthold et al. (Berthold and Fischer, 1997) (Fischer et al., 1998a) (Koch et al., 1998) (Fischer et al., 1998b). In their works Graph Grammars (see (Rozemberg et al., 1997)) are used to formally specify neural networks and their corresponding training algorithms. One of the benefits of using this formal framework is the support for proving properties of the training algorithms. Moreover, the proposed methodology can be used to design new network architectures along with the required training algorithms.

A proposal for the unification of deterministic and probabilistic learning in structured domains, thus including as special cases feed-forward and recurrent

neural networks, has been proposed by Frasconi et al. (Frasconi et al., 1998), where graphical models are used to describe in a unified framework both neural and probabilistic (Bayesian Networks) transductions involving data structures. The basic idea is to represent functional dependencies within an adaptive device by graphical models. The graphical models are then "unfolded" over the data structure to make explicit all the functional dependencies into the data. This process generates an encoding network which can be implemented either by a neural or a Bayesian network.

3 Need for a Neural Abstract Machine

As argued in Section 2.8, current neural network simulators and specification languages have too many drawbacks in order to be a valid tool for the development of successful neural solutions to applications problems. First of all they are too specific, since they typically implement a restricted set of neural models. Then, it is usually not possible, or very difficult, to slightly modify the specification of a given standard model, to combine different models, to insert a priori knowledge, and to develop a new model. Only some of them can deal with sequences (or structures). Finally, as pointed out in (Frasconi et al., 2000), in several application domains, it is not possible to assume both *causality* and *stationarity*, and at our knowledge none of them is able to cope with these requirements.

What we really need is to further develop the unifying approaches described in the previous section so to reach the full specification of a Neural Abstract Machine, i.e., a "universal" theory of neural computation, where all the basic and relevant neural concepts, and only them, are formally defined and used. For example, we must have the possibility to give a specification of input data types (static vectors, sequences, structures) and how to represent them (e.g., one-hot encoding, distributed representation, etc.), i.e., the object to be manipulated by the neural network. Then we must have the possibility to specify the operation types and their representation, e.g., functional dependencies, deterministic o probabilistic functions, gradient propagation, growing operators, compositional rules, pruning, knowledge insertion, shift operators, weight sharing, and so on. Finally, we have to master basic computational concepts and their implementation, e.g., model of computation, unrolling in time, unfolding on the structure, (non-)causality, (non-)stationarity, and so on.

When all these entities are defined and ways of implementing them are specified, to face an application problem we just have to write a few lines of "neural code" based on the neural abstract machine!! What we suggest is to perform a computational synthesis of the relevant issues in neural computing, recognizing the basic atoms of neural computation and how these basic atoms can be combined in order both to reproduce known neural models and to develop new architectures and learning algorithms, without the need to recode everything in a standard programming language such as `Java`, or `C++`. By using an analogy with the history of computers, we have to move from combinatorial or sequen-

tial circuits (current neural networks) to a Von Neumann machine (the Neural Abstract Machine).

4 Abstract State Machines

In order to devise a formal description of the *Neural Abstract Machine*, we need to decide which type of specification method to use. It is difficult to take such a decision choosing among many formal methods, since a lot of available theories are not practical for the description of complex dynamic real-world systems (Börger, 1999). We think that the *Evolving Algebras*, devised by Gurevich (Gurevich, 1995), present some useful features for the *Neural Abstract Machine* formal development. Furthermore, this formalism has been intensively used for the formal design and analysis of various hardware systems, algorithms and programming languages semantics, showing an astonishing simplicity while preserving mathematical soundness and completeness. The main reason for its simplicity is the imperative specification style that, in contrast to conventional algebraic specification methods, allows an easier understandability. This leads to an easy definition of formal and informal requirements, turning them into a satisfactory *ground model* (Börger, 1999), i.e. a formal model that satisfies all the system requirements. Moreover, as stated in Börger's work (Börger, 1995), the evolving algebras have many other features that can help when devising a system or defining a language semantics:

- The *freedom of abstraction*, that allows incremental development of systems by stepwise refinement through a vertical hierarchy of intermediate models;
- The *information hiding* and the *precise definition of interfaces*, that helps the horizontal structuration of modules. In practice, when using a function we do not care about its implementations, we are interested only in the interface;
- The *scalability* to complex real-world systems;
- The easy *learning* and *usability* of the model.

4.1 Domains and Dynamic Functions

When specifying a system or a language semantics some entities must be defined. In particular, the basic object classes and the set of elementary operations on objects, i.e. the basic *domains* and *functions*. Each domain represents a category of elements that contributes to the definition of the whole system. These domains are completely abstract, since they represent some sort of information that will be specified later. For example, a generic neural network is built up by a linked set of computational units. In order to formalize a neural network with an Evolving Algebra (*EA*), we may define two basic domains: the *NEURONS* set and the *CHANNELS* set. Note that this is only one of all possible formalizations of the basic elements of a neural network. Even if usually within the EA framework the domains are static, it is also possible to have dynamic domains (Gurevich, 1995).

This is very useful for our project, since there are many neural architectures that grow or shrink (through pruning) during training. Furthermore, the time unfolding of recurrent networks and the graph unfolding of recursive networks is done at run-time, thus we can not assume to have static networks.

Once the system domains have been defined, the set of all properties and elementary operations must be specified by corresponding functions. A function is defined in a mathematical sense as:

> A function is a set of $(n+1)$-tuples, where the $(n+1)$-th element is functionally dependent from the first n elements (its arguments) (Börger, 1999).

Typically, each system operation updates a value (e.g. a memory location in an hardware system) given a set of other values (e.g registers). In the ASM framework this corresponds to the *dynamic function update* or *destructive assignment* defined as:

$$f(t_1, \ldots, t_n) \leftarrow t$$

where f is an arbitrary n-ary function as defined above, $t_1, \ldots, t_n$ are the function parameters defined in some domains, and t is the value at which the function is set. The *EA*s are so general that each used term could be of any complexity or abstraction. Continuing with the previous neural example, we can define the function that given a channel returns its strength parameter as:

$$weight : CHANNELS \rightarrow I\!R \ .$$

Since during training the weights of the network are changed, the update of one channel weight can be formalized as:

$$weight(C_i) \leftarrow weight(C_i) + \ \delta_w(C_i) \ ,$$

where C_i is an object belonging to the domain *CHANNELS*, and δ_w is a function that returns the amount of weight update associated to the specified channel (a real number).

4.2 Abstract States and Transition Instructions

Thanks to the *EA*'s freedom of abstraction and information hiding properties, it is possible to produce rigorous high level specifications without worrying about the future design. This is accomplished using the concepts of *abstract state* and *abstract transition function*. It is for this reason that Evolving Algebras are nowadays also termed *Abstract State Machines* (ASM). The ASM concept of abstract state should not be confused with the notion of state used in finite state automata:

> The ASM abstract state is a collection of domains and dynamic functions defined on the domains.

The abstract states are subject to integrity constraints that partially describe the machine behavior. More clearly, when a system is in a state, the set of all reachable states is limited by the machine specification. Even if the notation used for the constraints formulation is not limited by any programming language, in order to describe the ASM behavior a set of basic actions have been designed. These actions constitute the set of abstract transition functions also termed *machine abstract instructions*. The dynamic function updates are the basic operations by which the behavior of a system can be described. However, they are inadequate for a complete system description, so the model needs to be enriched with control operators, frequently termed *rules*. The most general of these operators is the *guarded assignment*:

if *Cond* **then** *Updates*

where *Cond* is a condition (the constraint), and *Updates* consists of finitely many function updates, which are executed simultaneously. At this point we have all the ingredients for an ASM system behavior description (Börger, 1995, 1999):

> **Definition of Abstract State Machines.** An ASM $\mathcal{M}$ is a finite set of guarded function updates, used for evolving step by step the machine. When $\mathcal{M}$ is in a state $\mathcal{S}$, all guard conditions are evaluated (with standard logic), and all instructions with the verified guard condition are selected. Then, all the *update functions* of all selected instructions are simultaneously executed, transforming the state $\mathcal{S}$ into a new state $\mathcal{S}'$. This procedure of the ASM $\mathcal{M}$ is iteratively applied as long as possible (i.e. until there are not verified rule conditions). The ASM *run* can be defined as the set of all transitions that bring the machine $\mathcal{M}$ from the initial state $\mathcal{S}$ to the final state $\mathcal{S}'$, where no more rules have the verified guard.

The above definition shows how simple is the ASMs concept, since it is the only notion one has to know about the ASM semantics in order to be able to use this formalism.

4.3 ASM Rules and Graphical Representation

As previously stated, all the needed ingredients for a system behavior description with ASMs have been introduced. Nevertheless, the model is so general that a free use of programming notation is permitted. Even if a general rule for ASMs is that, unless there is an important reason, it is better to avoid the use of complex non-standard concepts or notations, in order to simplify the designer work, some simple and generic rules have been introduced in the ASMs formalism:

- **skip**;
- $p(t_1, \ldots, t_m)$;
- **forall** f **in** *Dom* **such that** *Cond(f)*
  Rule;

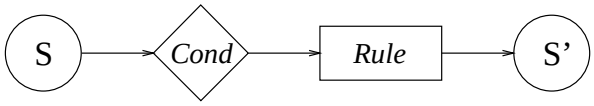
– **choose** f **in** Dom **such that** $Cond(f)$
  $Rule$;

where, f is a function signature, Dom is a domain to which the given signature must belong, and $Cond$ is an arbitrary condition on the function. The first rule, obviously, does nothing. The second rule represents a collection of rules. The third and the fourth rules allow a selection of rules to be executed. Note that with the last rule a sort of explicit nondeterminism has been introduced. An alternative nondeterministic solution could be to modify the ASMs semantic definition by allowing the firing at each step of only one of all fireable updates.

In order to further simplify the design phase of a system also a graphical representation for the previous rules has been introduced. The guarded update is an instruction that allows the ASM transition from a state $\mathcal{S}$ to a state $\mathcal{S}'$, thus the guarded update rule can be rewritten as:

if $(currentstate = \mathcal{S})$ & $(Cond)$ **then**
  $Updates$
  $currentstate \leftarrow \mathcal{S}'$

Which can be graphically represented by the following diagram:



Note that the rectangular box may also contain a set of rules. This is very helpful when formalizing a problem with a top-down approach, because a complex concept can be expressed by a box, and left for future expansion.

5 A Sketch of Feed-Forward Network Specification

Our approach to the *Neural Abstract Machine* (NAM) is based on the idea of a kernel able to process feed-forward neural networks. In this section we provide a simplified high level formal description of such kernel, adopting a joined bottom-up and top-down approach, and showing some operational details with ASMs. Note that this is only a (not rigorous) exercise in order to give an idea about how ASMs technology can be applied. In particular we show the main features on which we are working for the NAM project.

5.1 Neurons and Channels Domains Specification

As stated in Section 4.1, in order to design an ASM, the set of basic domains must be defined. In our framework, since a net is viewed as a collection of neurons and connections (channels), we define the *NEURONS* and the *CHANNELS* domains, which represent the basic block categories compounding a neural network.

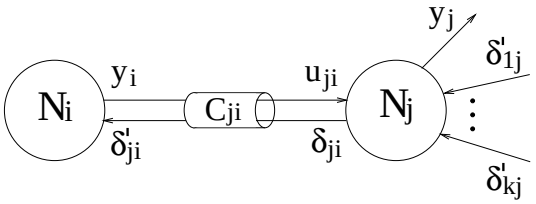


Fig. 2. A graphical representation of the required communication messages between channels and neurons. The signals u and y are the neuron input and output, δ is the gradient information computed by the neuron, and $\delta' = \delta w$ is the gradient information computed by the channel.

In view of the learning process, besides to neurons, we can assume that channels are "active" entities[1], i.e., able to update autonomously the associated weight, on the basis of the gradient information. In Figure 2 all the required communications between a channel and a neuron unit are shown.

Our basic assumption is that the kernel of the system is based on a *Neural Control Machine* (NCM), which dynamically generates the neural network connecting the basic blocks (channels and neurons), controls the flow of computation, furnishes the input data, and eventually the error information. The NCM should be able to manipulate also constructive algorithms, such as Cascade Correlation. In order to have this ability, the NCM should be able to communicate with each channel or neuron during the operative phase, sending signals such as *freeze* (or *de-freeze*). Moreover, the NCM needs to be able to dynamically create or destroy units during training, thus changing the neural architecture. The advantage of using the ASMs formal specification method is that such operations can be easily formalized with suitable function assignments.

5.2 Architecture Manipulation Functions

As previously stated, we have two basic block categories, represented by the *NEURONS* and *CHANNELS* domains.

In order to control the basic behavior of neurons and channels we need to define some functions over the domains *NEURONS* and *CHANNELS*. For instance, we need functions that, given a computational object, belonging to one of the two domains, return the "names" of all the other objects connected to it. Thus, for each domain, we need two functions, *source* and *dest*, which return the sources and the destinations for a given unit (N) or channel (C) during the forward phase:

$$source_C : CHANNELS \rightarrow NEURONS$$

[1] The assumption that a channel is an active entity is not a requirement. According to the actual implementation, it may also be a passive entity accessed and modified by a *Neural Control Machine*.

$$dest_C : CHANNELS \rightarrow NEURONS$$

$$source_N : NEURONS \rightarrow \mathscr{P}(CHANNELS)$$
$$dest_N : NEURONS \rightarrow \mathscr{P}(CHANNELS)$$

where $\mathscr{P}$ denotes the power set. In our formalization while a neuron may have many source and destination channels, a channel has only one source and one destination neuron. Nevertheless, higher order connections, which have multiple sources and/or multiple destinations, can be easily modeled by the following new definition:

$$source_C : CHANNELS \rightarrow \mathscr{P}(NEURONS).$$

Since the Neural Control Machine needs to access all resources of the implemented neural network (neurons and channels), two functions are required:

$$all\_neurons : \mathscr{P}(NEURONS)$$
$$all\_channels : \mathscr{P}(CHANNELS)$$

which return the set of all neurons and the set of all channels used by the network. Furthermore, in order to know which are the input and the output units of the neural network the NCM requires the following two functions:

$$in\_layer \ : \mathscr{P}(NEURONS)$$
$$out\_layer : \mathscr{P}(NEURONS)$$

We assume that the NCM uses a data flow computational paradigm, i.e. the computation starts when the NCM transmits data to the *in_layer* units, and finishes when all units in the *out_layer* have computed the output, and no computation for other units needs to be performed. Moreover, it is responsibility of the NCM to implement the backward propagation of the gradient information across the network. Even in this case, a data flow computation is performed.

The previously defined functions allow the generation of a neural network. Actually, after the creation of a set of computational units $n_i \in NEURONS$ and $c_{ij} \in CHANNELS$, the network can be built up by a set of simple assignments like the followings:

$$source_C(c_{ij}) \leftarrow n_j$$
$$dest_C(c_{ij}) \leftarrow n_i$$

$$source_N(n_i) \leftarrow \{c_{ih}, \ldots, c_{ik}\}$$
$$dest_N(n_i) \leftarrow \{c_{mi}, \ldots, c_{ni}\}$$

5.3 Units Computation Functions

Each computational unit behavior may be described by three basic functions for each direction of the flow of computation (either forward or backward): the *input*

function, the *state transition function* and the *output function*. In the following we show the functions characterizing the forward computation of the neural units.

The input function of the neural units during the forward phase could be described by the following interface:

$$in\_forw_N : NEURONS \rightarrow INPUT,$$

where $INPUT$ is a new domain introduced for generalize the data description. With this approach the specifications can be left for future refinements, however, we can imagine that each channel transmits to a neuron a pair of data, formed by the signal conveyed by the channel and the weight associated to the channel. In this way, each neuron can use any internal computation function. For this reason we can assume $INPUT \equiv \mathscr{P}((\mathbb{R}, \mathbb{R}))$. As a result of this choice, accessing the input of a neuron with the function *in_forw*, a list of couples is returned.

Obviously, data are returned only if previously sent to the neuron by all channels. For this reason when data are ready on the output interface of all channels they are copied in the input function with the following assignment:

$$in\_forw_N(n_i) \leftarrow \{out\_forw_C(c_{ih}), \ldots, out\_forw_C(c_{ik})\}$$

where $c_{ki}, \ldots, c_{hi}$ could be previously determined using the function $source_N(n_i)$, and out_forw_C is the channel output function. This way of processing emphasizes the data-flow paradigm implicitly assumed by our system. In fact, the internal computation can be done only when all data are ready in the input side of the neuron, i.e. when the neural unit possesses all the required inputs.

At this point the neurons can carry out the internal computations of the input data. In order to do this the following functions are required:

$$compute\_forw_N : INPUT \rightarrow STATE$$
$$output\_function_N : STATE \rightarrow OUTPUT$$

The function $compute\_forw_N$ computes the internal state of the neuron (e.g. the net value), and the function $output\_function_N$ computes the output value starting from the internal state. As previously stated for the $INPUT$ domain, also the $STATE$ and the $OUTPUT$ domains can be left unspecified for future refinements, however, for the sake of presentation, we assume $STATE \equiv OUTPUT \equiv \mathbb{R}$. Note that, the state is not strictly necessary in feed-forward networks, however it allows the storing of information needed by the backward phase.

Two other functions are needed for accessing the $STATE$ and $OUTPUT$ values of the neuron:

$$state\_forw_N : NEURONS \rightarrow STATE$$
$$out\_forw_N : NEURONS \rightarrow OUTPUT$$

They can be instantiated as follows:

$$state\_forw_N(n_i) \leftarrow compute\_forw_N(in\_forw_N(n_i))$$
$$out\_forw_N(n_i) \leftarrow output\_function_N(state\_forw_N(n_i))$$

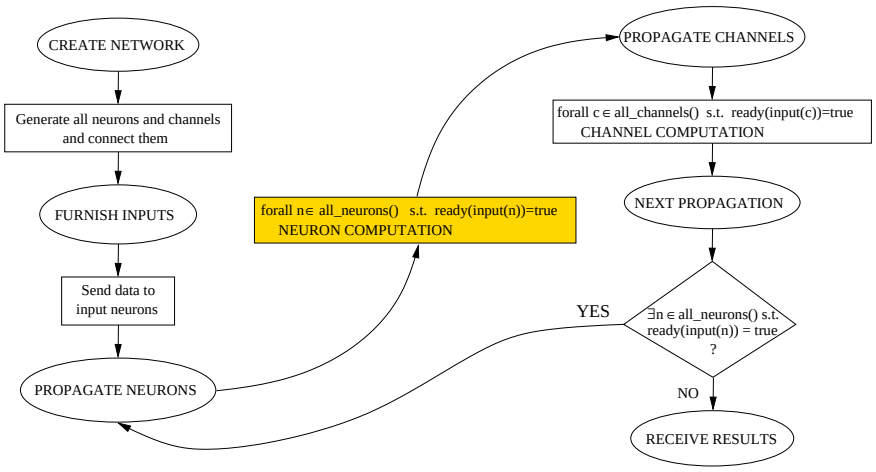


Fig. 3. This flowchart shows the part of NCM involved in the forward propagation of data in a feed-forward neural network. The NCM generates a neural network with a specified topology, then it furnishes an input to the network and propagates it through the network.

Note that the function $compute\_forw_N$ does not take into account the previous state, so there is no memory of the past. If the memory is required, as in neural networks with short term memory, the function interface and the state transition may be changed in the following way:

$$compute\_forw_N \ : \ STATE \times INPUT \rightarrow STATE$$
$$state\_forw_N(n_i) \leftarrow compute\_forw_N(state\_forw_N(n_i), in\_forw_N(n_i))$$

Even if the proposed functions give just a partial definition for the feed-forward computation of a neural unit, they show how a neural model can be devised with ASMs. The backward propagation of the neuron unit is not much different from the forward propagation, and also the channel functions (either forward or backward) are similar to the neuron functions. It is theoretically interesting to note that, with such a block approach, neurons and channels are very similar. In an object oriented programming language the *NEURONS* and *CHANNELS* domains could be two classes inheriting from the same class.

5.4 The Neural Control Machine

Here, we are going to describe a possible formalization of the feed-forward part of the Neural Control Machine. In Figure 3 we show the part of the NCM, that starting from a clear blackboard, creates a network and forward propagates a given input data till an output result is returned.

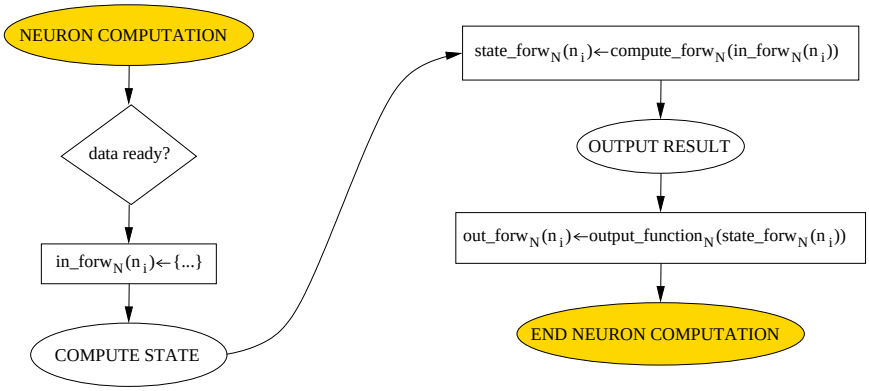


Fig. 4. This flowchart is an example of refinement of the NEURON COMPUTATION left unexplained in the flowchart of Figure 3. When a data is ready in input, it is assigned to the internal memory working area, then the internal state is computed, and finally the output result of the neuron is computed.

The first step of the NCM is the creation of the neural network. This can be accomplished creating all the neurons and all the required channels. Then, all the objects are linked. When the network is ready, the forward propagation of one input data can be accomplished. In order to do this, the data is sent to the input neurons of the network (known through the function *in_layer*). After that, the propagation of data starts. The machine iteratively propagates data through all neurons and channels. When the network output is ready, the NCM can collect it from the output units (known through the function *out_layer*).

In order to show how the freedom of abstraction property of the ASMs can help when incrementally developing by stepwise refinement, we have refined the neuron computation procedure left undefined in Figure 3. The specification of this computation is given in Figure 4.

When for a neuron the data is ready in input, it is copied into an internal working area for fast and easy access during the further computation. At this point the internal state is computed using the function $compute\_forw_N$ and the result is internally stored assigning it to the function $state\_forw_N$. Finally, the output result can be computed using the function $output\_function_N$ over the internal state and the result is stored by means of the function out_forw_N.

6 Some details of the Neural Abstract Machine for Sequences

We have previously explained the basic behavior of the NAM kernel able to process feed-forward networks. In this section we provide a simple high level formal

description of the part of Neural Abstract Machine devoted to the treatment of recurrent networks for structured data.

As previously stated (see Section 3), in order to define the NAM we need a general unifying theory of all different types of neural networks architectures and training algorithms. Even if some unification works have already appeared, we use here only a limited set of such ideas, since most of them are addressed to the internal organization of networks and their learning algorithms.

In particular, the work by Tsoi and Back (Tsoi and Back, 1997) and Tsoi (Tsoi, 1998b) showed that several recurrent networks can be reduced to a canonical recurrent form. Furthermore, if all cycles in the network are controlled by a delay operator, the forward and learning operations of a recurrent network can be easily reduced to the forward and learning operations of a feed-forward encoding network with identical behavior and weights[2]. For example, when considering a recurrent network where internal loops are controlled by a delay operator of one time step (i.e q^{-1}), the system transitions could be represented by the following Mealy model:

$$X_t = f_t(X_{t-1}, U_t) \quad \text{and} \quad Y_t = g_t(X_t, U_t), \tag{1}$$

where X_t is the internal state at time t, U_t is the input at time t, Y_t is the computed output at time t, f_t and g_t are the transition function and the output function at time t. Note that, even if the state transition is recursive, the functions f_t and g_t are intrinsically non-recursive, in fact the past information (i.e. the previous system computation) does not belong to the function f_t but is given to it as an external information. This show that the recursive system can be easily modeled by a suitable composition of non-recursive functions. Note also that the given specification assumes the possibility of a non stationary system, where functions f_t and g_t change over time.

It is clear at this point that given a data set and a (recurrent) network specification, the *Neural Abstract Machine* should be able to devise the parametric non-recursive functions f_t and g_t by which to define the feed-forward neural network that behaves as the recurrent network. In other words the NAM should be able to use the formal description[3] of a (recurrent) neural network in conjunction with the data set in order to find the optimal solution for the network parameters.

For a better understanding of the expected behavior of the NAM, let us reconsider Equations 1, which explain how the recurrent network can be dynamically unfolded over a sequence, generating a feed-forward network (also called *encoding network*). Informally, the encoding network is derived by dynamically unfolding the temporal operations of the recurrent network over each element of the observed sequence. Specifically, the recurrent network without the delayed links is replicated for each element of the sequence, and then each delayed link from neural unit i to unit j is mapped into a corresponding link from unit i to

[2] The weights of the recurrent network are shared among all layers of the transformed net.

[3] In order to give a formal description of a neural network, also a *Neural Language* should be defined.

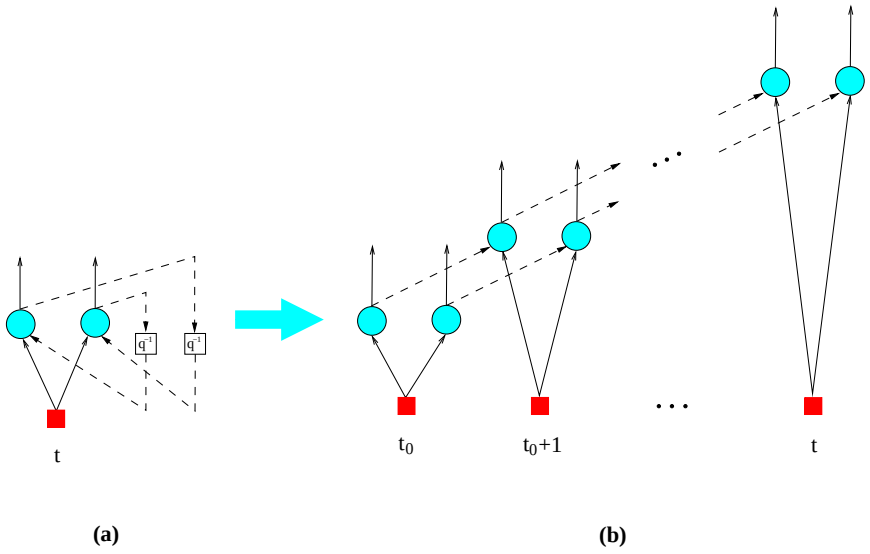


Fig. 5. Given an input sequence, a recurrent network (a) can be unfolded in time, originating an equivalent feed-forward network called encoding network (b).

unit j of adjacent layers in the new feed-forward network (see Figure 5). In this way, for each input sequence a different feed-forward network is generated.

Since we can assume that all recurrent neural networks can be reduced to feed-forward networks, we now try to formalize a machine able to apply this transformation.

6.1 The Unfolding Machine

Let us study the high-level description of the unfolding process when the input data is constituted by sequences. The process to which we are interested in can be easily expressed by the (ASMs based) flowchart shown in Figure 6, where the first part of a sequence processing is shown.

Notice that the machine needs to access a database of sequences located somewhere. We are not interested in details such as how it is made and where it is stored. As first operation the machine "loads" a sequence from the database into the machine working area. This is then followed by the unfolding operation. The cycle that controls the unfolding is based on the idea that a sequence can be represented as a directed graph composed of nodes connected as shown in the following:

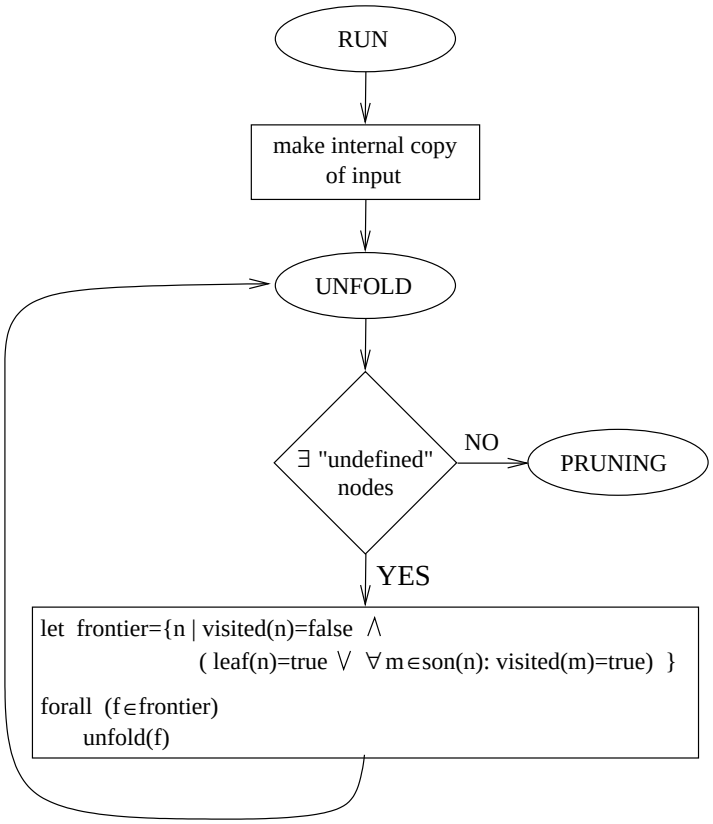
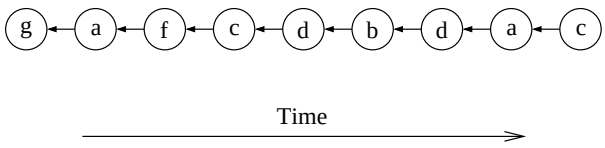


Fig. 6. Flowchart for the processing of sequences by a recurrent network. The feed-forward encoding network, for each sequence, is generated by unfolding the recurrent network on the input sequence. Notice that the machine is defined to work also for structured data, such as trees.



Specifically, note that we assume the inverse orientation of the arcs with respect to the time of elaboration. The reason is that, using this approach, we can further extend the machine to the processing of structures, such as trees, forests and more in general DOAGs (Directed Ordered Acyclic Graphs). In fact,

a recurrent network can be applied to structured data by unfolding it over the structure in a way which is very similar to the one used for sequences[4].

In the flowchart there are not operational details about the algorithm, there is only a (very) high level description of the unfolding operation. The first step only asserts that a sequence must be loaded for future processing. The second step is based on the assumption that all the elements (termed nodes from now on) of the loaded sequence can be marked as "visited" or "not visited". The second step is a loop over all not visited nodes, stating that at each iteration, for each element of the frontier, the encoding network must be extended unfolding the recurrent network.

During the first iteration, the frontier is constituted by all leaves of the data structure (i.e. the first element of the sequence). From the second step on, the frontier is determined by all those nodes still not visited for which all sons have been visited. The loop stops when the roots are reached (i.e. the last element of the sequence). When there are no more undefined elements (i.e. all elements of the data have been used for the unfolding operation) the *Unfolding Machine* leaves the control to the *Pruning Machine*, which, in the case of complex data structure, removes portions of the encoding network, for which no gradient information must be computed. At the end of this process the encoding network is ready to be used.

The mathematical description of the frontier computation is based on some simple functions. The function $visited(n)$ returns a boolean value indicating whether the node n has been visited during the unfolding loop, the function $leaf(n)$ returns a boolean value that signal whether a node is a leaf of the structure, and finally, $son(n)$ returns the set of all nodes m that have the node n as father.

The most important and complex part of the *Unfolding Machine* is the function $unfold(n)$ that, for each node n of the structured data (e.g. an element of a sequence), must dynamically create the feed-forward encoding network.

7 Conclusions

Sequence Processing is a very complex task. According to the learning problem different approaches can be used. Neural networks for sequence are especially useful when data is numerical and noisy, or when uncertainty characterizes the learning task.

Training a recurrent neural network, however, is not a trivial task. Several choices about the network architecture, as well as the training algorithm must be taken. Moreover, heuristics must be used to set the learning rates, the regularization tools, the pruning procedure, and so on.

When present, a priori knowledge should be used to reduce the burden of training a network. In some cases, this may speed up learning and improve the generalization ability of the network. However, in general, learning remains difficult, also considering the long-term dependencies.

[4] Note that sequences are a particular instance of DOAG.

To successfully apply recurrent neural networks in real world domains, all the above choices must be taken in the shortest time and in the most reliable way. At present, no software tool is available to support all the aspects described above, including the rapid development of ad hoc new neural network models.

Recalling some recent attempts to find a unifying theory for (recurrent) neural networks, we suggested to push this challenge further, since this would create the theoretical basis on which to build a *Neural Abstract Machine*. This machine could be used to formally define a sort of neural programming language for fast prototyping and development of neural solutions to sequence (and structured) learning problems. As computational formalism we suggested to use the *Abstract State Machines* (ASM), which allow to describe the *Neural Abstract Machine* at different levels of abstraction and detail, while preserving simplicity of presentation and comprehension. A very brief and preliminary example of how to use ASM for neural networks was discussed.

Finally, we stress that the development of a *Neural Abstract Machine* would: *1)* help in understanding what is really needed and worth to be used in (recurrent) neural networks; *2)* create a universal computational language for neural computation; *3)* improve the possibility to integrate, at a computational level, neural networks with other approaches such as expert systems, Bayesian networks, fuzzy systems, and so on.

Bibliography

Abu-Mostafa, Y. S., 1990. Learning from Hints in Neural Networks. *Journal of Complexity* 6:192–198.

Abu-Mostafa, Y. S., 1993a. Hints and the VC Dimension. *Neural Computation* 5, no. 2:278–288.

Abu-Mostafa, Y. S., 1993b. A Method for Learning From Hints. In *Advances in Neural Information Processing Systems*, eds. S. J. Hanson, J. D. Cowan, and C. L. Giles, vol. 5, pp. 73–80. Morgan Kaufmann, San Mateo, CA.

Abu-Mostafa, Y. S., 1995a. Financial Applications of Learning from Hints. In *Advances in Neural Information Processing Systems*, eds. G. Tesauro, D. Touretzky, and T. Leen, vol. 7, pp. 411–418. The MIT Press.

Abu-Mostafa, Y. S., 1995b. Hints. *Neural Computation* 7, no. 4:639–671.

Al-Mashouq, K. A. and Reed, I. S., 1991. Including Hints in Training Neural Nets. *Neural Computation* 3, no. 3:418–427.

Alquézar, R. and Sanfeliu, A., 1995. An Algebraic Framework to Represent Finite State Machines in Single-Layer Recurrent Neural Networks. *Neural Computation* 7, no. 5:931–949.

Amari, S., 1995. Information Geometry of the EM and em Algorithms for Neural Networks. *Neural Networks* 8, no. 9:1379–1408.

Angeline, P. J., Saunders, G. M., and Pollack, J. P., 1994. An Evolutionary Algorithm That Constructs Recurrent Neural Networks. *IEEE Transactions on Neural Networks* 5, no. 1:54–65.

Battiti, T., 1992. First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method. *Neural Computation* 4, no. 2:141–166.

Bengio, Y., Simard, P., and Frasconi, P., 1994. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks* 5, no. 2:157–166.

Berthold, M. and Fischer, I., 1997. Formalizing Neural Networks Using Graph Transformations. In *Proceedings of the IEEE International Conference on Neural Networks*, vol. 1, pp. 275–280. IEEE.

Börger, E., 1995. Why Use Evolving Algebras for Hardware and Software Engineering? In *SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, ed. J. W. Miroslav BARTOSEK, Jan STAUDEK, vol. 1012 of *Lecture Notes in Computer Science*, pp. 236–271. Berlin Heidelberg New York: Springer-Verlag.

Börger, E., 1999. High Level System Design and Analysis using Abstract State Machines. In *Current Trends in Applied Formal Methods (FM-Trends 98)*, eds. D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, vol. 1641 of *Lecture Notes in Computer Science*, pp. 1–43. Berlin Heidelberg New York: Springer-Verlag.

Casey, M., 1996. The Dynamics of Discrete-Time Computation, with Application to Recurrent Neural Networks and Finite State Machine Extraction. *Neural Computation* 8, no. 6:1135–1178.

Das, S., Giles, C. L., and Sun, G. Z., 1992. Learning Context-free Grammars: Limitations of a Recurrent Neural Network with an External Stack Memory. In *Proceedings of The Fourteenth Annual Conference of the Cognitive Science Society*, pp. 791–795. San Mateo, CA: Morgan Kaufmann Publishers.

Dempster, A. P., Laird, N. M., and Rubin, D. B., 1977. Maximum likelihood from incomplete data via the EM algorithm (with discussion). *Journal of the Royal Statistical Society series B* 39:1–38.

Fahlman, S., 1991. The Recurrent Cascade-Correlation Architecture. In *Advances in Neural Information Processing Systems 3*, eds. R. Lippmann, J. Moody, and D. Touretzky, pp. 190–196. San Mateo, CA: Morgan Kaufmann Publishers.

Fischer, I., Koch, M., and Berthold, M. R., 1998a. Proving Properties of Neural Networks with Graph Transformations. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pp. 457–456. Anchorage, Alaska.

Fischer, I., Koch, M., and Berthold, M. R., 1998b. Showing the Equivalence of Two Training Algorithms - Part2. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pp. 441–446. Anchorage, Alaska.

Forcada, M. L. and Carrasco, R. C., 1995. Learning the Initial State of a Second-Order Recurrent Neural Network during Regular-Language Inference. *Neural Computation* 7, no. 5:923–930.

Frasconi, P., Gori, M., Maggini, M., and Soda, G., 1991. A Unified Approach for Integrating Explicit Knowledge and Learning by Example in Recurrent Networks. In *International Joint Confernece on Neural Networks*, pp. 811–816.

Frasconi, P., Gori, M., and Soda, G., 1995. Recurrent Neural Networks and Prior Knowledge for Sequence Processing: A Constrained Nondeterministic Approach. *Knowledge Based Systems* 8, no. 6:313–332.

Frasconi, P., Gori, M., and Sperduti, A., 1998. A General Framework for Adaptive Processing of Data Structures. *IEEE Transactions on Neural Networks* 9, no. 5:768–786.

Frasconi, P., Gori, M., and Sperduti, A., 2000. Integration of Graphical-Based Rules with Adaptive Learning of Structured Information. In *Hybrid Neural Symbolic Integration*, eds. S. Wermter and R. Sun. Springer-Verlag. To appear.

Geman, S., Bienenstock, E., and Doursat, R., 1992. Neural Networks and the Bias/Variance Dilemma. *Neural Computation* 4, no. 1:1–58.

Giles, C. L., Chen, D., Sun, G.-Z., Chen, H.-H., Lee, Y.-C., and Goudreau, M. W., 1995. Constructive Learning of Recurrent Neural Networks: Limitations of Recurrent Casade Correlation and a Simple Solution. *IEEE Transactions on Neural Networks* 6, no. 4:829–836.

Giles, C. L., Miller, C. B., Chen, D., Chen, H. H., Sun, G. Z., and Lee, Y. C., 1992. Learning and Extracted Finite State Automata with Second-Order Recurrent Neural Networks. *Neural Computation* 4, no. 3:393–405.

Giles, C. L. and Omlin, C. W., 1993a. Extraction, Insertion and Refinement of Symbolic Rules in Dynamically-Driven Recurrent Neural Networks. *Connection Science* 5, no. 3:307–337.

Giles, C. L. and Omlin, C. W., 1993b. Rule Refinement with Recurrent Neural Networks. In *1993 IEEE International Conference on Neural Networks (ICNN'93)*, vol. II, p. 810. Piscataway, NJ: IEEE Press.

Giles, C. L. and Omlin, C. W., 1994. Pruning Recurrent Neural Networks for Improved Generalization Performance. *IEEE Transactions on Neural Networks* 5, no. 5:848–851.

Goudreau, M. W., Giles, C. L., Chakradhar, S. T., and Chen, D., 1993. On Recurrent Neural Networks and Representing Finite State Recognizers. In *Third International Conference on Artificial Neural Networks*, pp. 51–55. The Institution of Electrical Engineers, London, UK.

Gurevich, Y., 1995. Evolvin Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, ed. E. Börger, pp. 9–36. Oxford University Press.

Hochreiter, S. and Schmidhuber, J., 1997. Long Short Term Memory. *Neural Computation* 9, no. 8:123–141.

Judd, J. S., 1989. *Neural Network Design and the Complexity of Learning*. MIT press.

Koch, M., Fischer, I., and Berthold, M. R., 1998. Showing the Equivalence of Two Training Algorithms - Part1. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pp. 441–446. Anchorage, Alaska.

Kolen, J. F., 1994. Fool's Gold: Extracting Finite State Machines from Recurrent Network Dynamics. In *Advances in Neural Information Processing Systems*, eds. J. D. Cowan, G. Tesauro, and J. Alspector, vol. 6, pp. 501–508. Morgan Kaufmann Publishers, Inc.

Kremer, S., 1996. Finite State Automata that Recurrent Cascade-Correlation Cannot Represent. In *Advances in Neural Information Processing Systems 8*, eds. D. Touretzky, M. Mozer, and M. Hasselno. MIT Press. 612-618.

Kremer, S. C., 1995. On the Computational Power of Elman-Style Recurrent Networks. *IEEE Transactions on Neural Networks* 6, no. 4:1000–1004.

Kuan, C.-M., Hornik, K., and White, H., 1994. A Convergence Result for Learning in Recurrent Neural Networks. *Neural Computation* 6, no. 3:420–440.

Lin, T., Horne, B. G., Tiño, P., and Giles, C. L., 1996. Learning Long-Term Dependencies in NARX Recurrent Neural Networks. *IEEE Transactions on Neural Networks* 7, no. 6:1329–1338.

Ma, S. and Ji, C., 1998. Fast Training of Reccurent Networks Based on the EM Algorithm. *IEEE Transactions on Neural Networks* 9, no. 1:11–26.

Maclin, R. and Shavlik, J. W., 1992. Refining Algorithms with Knowledge-Based Neural Networks: Improving the Chou-Fasman Algorithm for Protein Folding. In *Computational Learning Theory and Natural Learning Systems*, eds. S. Hanson, G. Drastal, and R. Rivest. MIT Press.

McClelland, J. L. and Rumelhart, D. E., 1987. *PARALLEL DISTRIBUTED PROCESSING, Explorations in the Microstructure of Cognition. Volume 1: Foundations Volume 2: Psychological and Biological Models*. MIT Press. The PDP Research Group, MIT.

Nerrand, O., Roussel-Ragot, P., Personnaz, L., Dreyfus, G., and Marcos, S., 1993. Neural Networks and Nonlinear Adaptive Filtering: Unifying Concepts and New Algorithms. *Neural Computation* 5, no. 2:165–199.

Omlin, C. and Giles, C., 1996. Constructing Deterministic Finite-State Automata in Recurrent Neural Networks. *Journal of the ACM* 43, no. 6:937–972.

Omlin, C. W., Giles, C. L., and Miller, C. B., 1992. Heuristics for the Extraction of Rules from Discrete-Time Recurrent Neural Networks. In *Proceedings International Joint Conference on Neural Networks 1992*, vol. I, pp. 33–38.

Pedersen, M. W. and Hansen, L. K., 1995. Recurrent Networks: Second Order Properties and Pruning. In *Advances in Neural Information Processing Systems*, eds. G. Tesauro, D. Touretzky, and T. Leen, vol. 7, pp. 673–680. The MIT Press.

Puskorius, G. V. and Feldkamp, L. A., 1994. Neurocontrol of Nonlinear Dynamical Systems with Kalman Filter Trained Recurrent Networks. *IEEE Transactions on Neural Networks* 5, no. 2:279–297.

Rozemberg, G., Courcelle, B., Ehrig, H., Engels, G., Janssens, D., Kreowski, H., and Montanari, U., eds., 1997. *Handbook of Graph Grammars: Foundations*, vol. 1. Workd Scientific.

Santini, S., Bimbo, A. D., and Jain, R., 1995. Block structured recurrent neural netorks. *Neural Networks* 8:135–147.

Saunders, G. M., Angeline, P. J., and Pollack, J. B., 1994. Structural and Behavioral Evolution of Recurrent Networks. In *Advances in Neural Information Processing Systems*, eds. J. D. Cowan, G. Tesauro, and J. Alspector, vol. 6, pp. 88–95. Morgan Kaufmann Publishers, Inc.

Schmidhuber, J., 1992. Learning Complex, Extended Sequences Using the Principle of History Compression. *Neural Computation* 4, no. 2:234–242.

Seidl, D. and Lorenz, D., 1991. A structure by which a recurrent neural network can approximate a nonlinear dynamic system. In *Proceedings of the International Joint Conference on Neural Networks*, vol. 2, pp. 709–714.

Siegelmann, H., Horne, B., and Giles, C., 1997. Computational capabilities of recurrent NARX neural networks. *IEEE Trans. on Systems, Man and Cybernetics* In press.

Siegelmann, H. T. and Sontag, E. D., 1991. Turing Computability with Neural Nets. *Applied Mathematics Letters* 4, no. 6:77–80.

Siegelmann, H. T. and Sontag, E. D., 1995. On the Computational Power of Neural Nets. *Journal of Computer and System Sciences* 50, no. 1:132–150.

Simard, P., Victorri, B., Le Cun, Y., and Denker, J., 1992. Tangent Prop—A Formalism for Specifying Selected Invariances in an Adaptive Network. In *Advances in Neural Information Processing Systems*, eds. J. E. Moody, S. J. Hanson, and R. P. Lippmann, vol. 4, pp. 895–903. Morgan Kaufmann Publishers, Inc.

Sontag, E., 1993. Neural Networks for control. In *Essays on Control: Perspectives in the Theory and its Applications*, eds. H. L. Trentelman and J. C. Willemsd, pp. 339–380. Boston, MA: Birkhauser.

Sperduti, A. and Starita, A., 1997. Supervised Neural Networks for the Classification of Structures. *IEEE Transactions on Neural Networks* 8, no. 3:714–735.

Sun, R. and Sessions, C., 1998. Extracting plans from reinforcement learners. *Proceedings of the 1998 International Symposium on Intelligent Data Engineering and Learning*, eds. L. Xu, L. Chan, I. King, and A. Fu, pp.243–248. Springer-Verlag.

Sutton, R. S., 1988. Learning to Predict by the Methods of Temporal Differences. *Machine Learning* 3:9–44.

Tesauro, G., 1992. Practical Issues in Temporal Difference Learning. *Machine Learning* 8:257–277.

Tino, P., Horne, B., and C.L.Giles, 1995. Fixed Points in Two–Neuron Discrete Time Recurrent Networks: Stability and Bifurcation Considerations. Tech. Rep. UMIACS-TR-95-51 and CS-TR-3461, Institute for Advance Computer Studies, University of Maryland, College Park, MD 20742.

Towell, G. G. and Shavlik, J. W., 1993. Extracting Refined Rules from Knowledge-Based Neural Networks. *Machine Learning* 13:71–101.

Tsoi, A., 1998a. Gradient Based Learning Methods. In *Adaptive Processing of Sequences and Data Structures: Lecture Notes in Artificial Intelligence*, eds. C. Giles and M. Gori, pp. 27–62. New York, NY: Springer Verlag.

Tsoi, A., 1998b. Recurren Neural Network Architectures: An Overview. In *Adaptive Processing of Sequences and Data Structures: Lecture Notes in Artificial Intelligence*, eds. C. Giles and M. Gori, pp. 1–26. New York, NY: Springer Verlag.

Tsoi, A. and Tan, S., 1997. Recurrent Neural Networks: A constructive algorithm and its properties. *Neurocomputing* 15, no. 3-4:309–326.

Tsoi, A. C. and Back, A., 1997. Discrete Time Recurrent Neural Network Architectures: A Unifying Review. *Neurocomputing* 15:183–223.

Wan, E. A. and Beaufay, F., 1998. Diagrammatic Methods for Deriving and Relating Temporal Neural Network Algorithms. In *Adaptive Processing of Sequences and Data Structures: Lecture Notes in Artificial Intelligence*, eds. C. Giles and M. Gori, pp. 63–98. New York, NY: Springer Verlag.

Wan, E. A. and Beaufays, F., 1996. Diagrammatic Derivation of Gradient Algorithms for Neural Networks. *Neural Computation* 8, no. 1:182–201.

Wiklicky, H., 1994. On the Non-Existence of a Universal Learning Algorithm for Recurrent Neural Networks. In *Advances in Neural Information Processing Systems*, eds. J. D. Cowan, G. Tesauro, and J. Alspector, vol. 6, pp. 431–436. Morgan Kaufmann Publishers, Inc.

Wiles, J. and Bollard, S., 1996. Beyond finite state machines: steps towards representing and extracting context-free languages from recurrent neural networks. In *NIPS'96 Rule Extraction from Trained Artificial Neural Networks Workshop*, eds. R. Andrews and J. Diederich.

Williams, R. J., 1992. Some Observations on the Use of the Extended Kalman Filter as a Recurrent Network Learning Algorithm. Tech. Rep. NU-CCS-92-1, Computer Science, Northeastern University, Boston, MA.

Williams, R. J. and Zipser, D., 1988. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. Tech. Rep. ICS Report 8805, Institute for Cognitive Science, University of California at San Diego, La Jolla, CA.

Wu, L. and Moody, J., 1996. A Smoothing Regularizer for Feedforward and Recurrent Neural Networks. *Neural Computation* 8, no. 3:461–489.