Towards the Verification and Validation of Online Learning Systems: General Framework and Applications

Ali Mili, GuangJie Jiang
New Jersey Inst. of Technology
Newark, NJ 07102
mili@cis.njit.edu

Bojan Cukic, Yan Liu
West Virginia University
Morgantown WV 26506-6109
{cukic,yanliu}@csee.wvu.edu

Rahma Ben Ayed
University of Tunis
Belvedere 1002 Tunisia
rahma_k@yahoo.com

September 29, 2003

Abstract

Online Adaptive Systems cannot be certified using traditional testing and proving methods, because these methods rely on assumptions that do not hold for such systems. In this paper we discuss a framework for reasoning about online adaptive systems, and see how this framework can be used to perform the verification of these systems. In addition to the framework, we present some preliminary results on concrete neural network models.

Keywords

Verification and Validation, Formal Methods, Refinement Calculi, On-Line Learning, Neural Networks, Adaptive Control, Radial Basis Functions, RBF neural networks, MLP neural networks.

1 Introduction: Position of the Problem

1.1 On-Line Learning: An Emerging Paradigm

Adaptive Systems are systems whose function evolves over time, as they improve their performance through learning. Online adaptive systems are attracting increasing attention in application domains where autonomy is an important feature, or where it is virtually impossible to analyze ahead of time all the possible combinations of environmental conditions that may arise. Examples of autonomous control applications are long term space missions where communication delays to ground stations are prohibitively long, and we have to depend on the systems' local capabilities to deal with unforeseen circumstances [11]. An example of the system dealing with complex environmental conditions are flight control systems, which deal with a wide range of parameters, and a wide range of environmental factors. Other proposed applications include collision avoidance systems, multi-vehicle cooperative control, intelligent scheduling in manufacturing [7], control systems for automobile steering based on feature recognition in images [6].

In recent years several experiments evaluated adaptive computational paradigms (neural networks, AI planners) for providing fault tolerance capabilities in control systems following sensor and/or actuator faults [23]. Experimental success suggests significant potential for future use. More recently, a family of neural networks, referred to as DCS (Dynamic Cell Structure) [12], have been used by NASA for on-line learning of aerodynamic derivatives [29] in a flight control system of an F-15.

The critical factor limiting wider use of neural networks and other soft-computing paradigms in process control applications, is our (in)ability to provide a theoretically sound and practical approach to their verification and validation. In the rest of the paper, we present a framework for reasoning about on-line learning systems in hope that it may become a candidate technology for their verification and validation.

1.2 Verifying On-Line Learning Systems

While they hold great technological promise, on-line learning systems pose serious problems in terms of verification and validation, especially when viewed against the background of the tough verification standards that arise

in their predominant application domains (flight control, mission control). Adaptive systems are inherently difficult to verify/ validate, *precisely because they are adaptive*. Specifically, consider that methods for software product verification are generally classified into three families [3]:

- *Fault Avoidance* methods, which are based on the premise that we can derive systems that are fault-free *by design*.
- *Fault Removal* methods, which concede that fault avoidance is unrealistic in practice, and are based on the premise that we can remove faults from systems after their design and implementation are complete.
- *Fault Tolerance* methods, which concede that neither fault avoidance nor fault removal are feasible in practice, and are based on the premise that we can take measures to ensure that residual faults do not cause failure.

Unfortunately, neither of these three methods is applicable *as-is* to adaptive systems, for the following reasons:

- *Fault Avoidance*. Formal design methods [10, 14, 22] are based on the premise that we can determine the functional properties of a system by the way we design it and implement it. While this holds for traditional systems, it does not hold for adaptive systems, since their design determines how they learn, but not what they will learn. In other words, the function computed by an online adaptive system depends not only on how the system is designed, but also on what data it has learned from.
- *Fault Removal: Verification*. Formal verification methods [1, 21, 16] are all based on the premise that we can infer functional properties of a software product from an analysis of its source text. While this holds for traditional systems, it does not hold for adaptive systems, whose behavior is also determined by their learning history.
- *Fault Removal: Testing*. All testing techniques [8, 15, 19] are based on the premise that the systems of interest will duplicate under field usage the behavior that they have exhibited under test. While this is true for traditional deterministic systems, it is untrue for adaptive systems, since the behavior of these systems evolves over time.
- *Fault Tolerance*. Fault tolerance techniques [2, 26] are based on an analysis of system functions, and the design of fault tolerant capabilities that provide redundancy with respect to these functions. With adaptive systems, it is not possible to analyze system functions because the functions of system components are not predetermined.

Because on-line learning systems are most often used in life-critical (e.g. flight control) and mission-critical (e.g. space) applications, they are subject to strict certification standards, leaving a wide technological gap —which we consider in this paper. In [20] we had surveyed existing approaches to the verification and validation of adaptive systems, and found that many of them fail to provide a sound basis for reasoning about them and for making provable statements about their behavior; other surveys of the domain have reached the same conclusion [27].

2 Tenets of a Refinement-Based Approach

In this paper, we present the general characteristics of our approach, then interpret it for some special cases of neural nets.

2.1 Characterizing Our Approach

Our approach to the verification of on-line learning systems can be summarized in the following premises:

- We establish the correctness of the system, not by analyzing the process by which the system has been designed, but rather by analyzing the functional properties of the final product, and how these functional properties evolve through learning.
- Qualifying the first premise, we capture the functional properties of the system not by the exact function that the system defines at any stage in its learning process, but rather by a *functional envelope*, which captures the range of possible functions of the system for a given learning history.
- In order to make testing meaningful, we need to ensure that the system evolves in a way that preserves or enhance its behavior under test. We call this *monotonic learning*, and we investigate it in some detail in section 4.1. Of course, on-line learning systems are supposed to get better as they acquire more learning data, but our definition of better is very specific: it means that the functional envelope of the system grows increasingly more refined with learning data (in the sense of refinement calculi [5, 9].
- In order to support some form of correctness verification, we must recognize that the variability of learning data and the focus on functional envelope (rather than precise function) weaken considerably the kinds of functional properties that can be established by correctness verification. Typically, all we can prove are minimal safety conditions; we refer to this as *safe learning* (proving that learning preserves

safety conditions), and we discuss it briefly in section 4.2.

In the sequel, we briefly introduce some mathematical background, which we use in the remainder of the paper.

2.2 Specification Structures

The functional verification of a system, whether adaptive or not, can only be carried out with respect to predefined functional properties, which we capture in *specifications*. In this paper, we model specifications by means of binary relations. This modeling decision builds on the tradition of functional semantics introduced by Mills et al [18, 17, 21] by adding the feature of non-deterministic specifications, along with a demonic interpretation of non-determinacy. A *relation* R from set X to set Y is a subset of the Cartesian product $X \times Y$. A *homogeneous* relation on S is a relation from S to S. We use relations to represent specifications. Among relational constants we cite the identity relation, denoted by I, and the universal relation, denoted by L. Among operations on relations we cite the product, which we represent by $R \circ R'$ or by RR' (when no ambiguity arises), the complement, which we represent by $\overline{R}$, the inverse, which we represent by $\widehat{R}$, and the set theoretic operations of union and intersection.

We wish to introduce an ordering between (relational) specifications to the effect that a specification is greater than another specification if and only if it captures stronger functional requirements. We refer to this ordering as the *refinement ordering*, we denote it by $R \sqsupseteq R'$, and we define it as

$$RL \cap R'L \cap (R \cup R') = R'.$$

This definition extends to non-deterministic relations (with a demonic interpretation of non-determinacy) the concept of refinement in functional semantics, whereby a function f refines a function g if and only if $f \supseteq g$. The following definition and proposition give the reader some intuition for the meaning of the refinement ordering.

Definition 1 *A program P on space S is said to be correct with respect to specification R on S if and only if $\boxed{P} \sqsupseteq R$, where $\boxed{P}$ is the function defined by program P.*

Proposition 1 *Specification R refines specification R' if and only if any program correct with respect to R is correct with respect to R'.*

In [4], we have derived two propositions pertaining to the lattice properties of the refinement ordering. We present them here without proof, but with some discussion of their intuitive meaning.

Proposition 2 *Two relations R and R' have a least upper bound (also called the* join*) with respect to the refinement ordering if and only if they satisfy the condition (called the* consistency condition*):*

$$RL \cap R'L = (R \cap R')L.$$

When they do satisfy this condition, their join is denoted by $(R \sqcup R')$ and is defined by

$$R \sqcup R' = R \cap \overline{R'L} \cup R' \cap \overline{RL} \cup (R \cap R').$$

The consistency condition means that R and R' can be satisfied (refined) simultaneously. As for the expression of the join, suffice it to say that $(R \sqcup R')$ represents the specification that captures all the functional features of R (upper bound of R) and all the functional features of R' (upper bound of R') and nothing more (*least* upper bound). A crucial property of joins, for our purposes, is that an element A refines $R \sqcup R'$ if and only if it refines simultaneously R and R'.

In addition to discussing least upper bounds (joins), we also discuss greatest lower bounds (meets), which are introduced in the following proposition.

Proposition 3 *Any two relations R and R' have a greatest lower bound (also called the* meet*), which is denoted by $(R \sqcap R')$ and defined by*

$$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$

The meet of R and R' is a specification that is refined by R (lower bound of R), refined by R' (lower bound of R'), and is maximal (*greatest* lower bound): in other words, it captures all the functional features that are common to R and R'.

It is noteworthy that this lattice does have a universal lower bound, which is the empty specification, but does not have a universal upper bound. All relations that are deterministic and total are maximal in the refinement ordering.

3 A Computational Model for On-Line Learning Systems

As it evolves through learning, a neural net defines an evolving function from its set of inputs to its set of outputs. As such, it is a primary candidate for the kind of functional mathematics advocated by Linger et al in [18]. Yet we choose not to capture the semantics of a neural net by the function it computes because this function typically involves a significant arbitrary component (such as the arbitrary choice of initial weights in MLP, or the arbitrary choice of radial functions in RBF). The purpose of our computational model is to shift the focus away from the actual function computed by the neural net, and to concentrate instead on the *functional envelope* of the neural net.

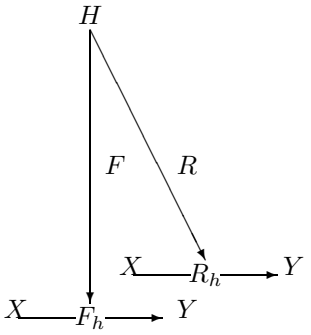

Figure 1: Abstract Computational Model

The functional envelope, which we define subsequently, abstracts away the random aspects in the function of a neural net, to capture only relevant aspects pertaining to the learning algorithm and the learning history.

Before we discuss the specifics the verification methods we propose, we first introduce an abstract computational model for adaptive systems and their evolution through learning. Figure 1 depicts the abstract model we have of an online adaptive system; this model is purposefully generic, to support a wide range of possible implementations (RBF, DCS, MLP), and to enable us to focus on relevant computational features (as opposed to being distracted by implementation specific details). Our model includes the following features:

- Set X represents the set of inputs that may be submitted to the adaptive system.
- Set Y represents the set of outputs that the adaptive system may return as output.
- Set H represents the set of learning data histories that are submitted to the adaptive system for learning; typically, this set is a set of sequences of the form (x, y), where $x \in X$ and $y \in Y$. We let ϵ represent the empty sequence (as an element of H).
- Function F is the function which, to each learning history h in H associates a function F_h from X to Y that captures the behavior of the adaptive system after receiving learning data h. According to this definition, the initial behavior of the adaptive system before any learning history is received is F_ϵ.
- Function R is the function which, to each learning history h in H associates a relation R_h from X to Y that captures the *learned behavior* of data h, and nothing else. Whereas F_h may include behavior that stems from its initialization, or stems from extrapolations, or stems from default options, R_h remains undefined or under-defined until learning data intervenes.

In order to elucidate the meaning of relation R_h, for history h, we consider the following development scenario for adaptive systems. An adaptive system is defined by some learning rule, which maps a learning history h into a function F_h; the learning algorithm is also defined by means of implementation-specific parameters, including randomly chosen parameters. For the sake of abstraction, we denote the vector of implementation-specific parameters by a variable, say λ, and we let Λ be the set of possible values for λ. To fix our ideas, we can think of Λ as representing a family of possible implementations of the learning algorithm, and of λ as a specific implementation within the selected family; also, we denote by F_h^λ the function that captures the behavior of the adaptive system whose parameters vector is λ, upon receiving learning data h. With this background in mind, we let R_h be defined as follows:

$$R_h = \bigsqcap_{\lambda \in \Lambda} F_h^\lambda.$$

By virtue of the definition of meet, $\bigsqcap_{\lambda \in \Lambda} F_h^\lambda$ can be interpreted to represent the functional information that is common to all possible implementations of the learning algorithm, for all possible values of λ. While F_h^λ is dependent on λ, R_h is dependent on Λ.

4 Verification of on-Line Learning Systems

Given that we have derived the *functional envelope* of an on-line learning system (as relation R_h), we discuss now how we can infer functional properties of the system. We discuss two methods in turn: *Monotonic Learning* and *Safe Learning*. Though there are significant differences, it is possible to view monotonic learning as the equivalent of testing for traditional deterministic programs; and to view safe learning as the equivalent of program proving. We will discuss subsequently in what sense and to what extent this analogy is valid.

4.1 Monotonic Learning

The idea of *monotonic learning* is to ensure that the adaptive system learns in a monotonic fashion, so that whatever claims we can make about the behavior of the system prior to its deployment are upheld while the system evolves through learning. Of course, we can hardly expect F_h^λ to be monotonic with respect to h, since there is no way to discriminate between information of F_h^λ that stems from learning and information that stems from arbitrary choices. In addition, because F_h^λ is total and deterministic, it is in fact maximal in the refinement ordering, hence cannot be further refined. We can, however, expect R_h to be monotonic, in the following sense.

Definition 2 *An adaptive system is said to exhibit* monotonic learning *if and only for all h in H, and for all (x, y) in $X \times Y$,*

$$R_{h.(x,y)} \sqsupseteq R_h$$

where $h.(x, y)$ is the sequence obtained by concatenating h with (x, y).

We may narrow down this definition to make it specific to a learning history (h) and to a learning pair (x, y). Traditional certification algorithms observe the behavior of a software product under test, and make probabilistic/ statistical inferences on the operational attributes of the product (reliability, availability, etc). The crucial hypothesis on which these probabilistic/ statistical arguments are based is that the software product will reproduce under field usage the behavior that it has exhibited under test. This hypothesis does not hold for adaptive neural nets, because they evolve their behavior (learn) as they practice their function.

The interest of monotonic learning is that whatever properties can be established by analyzing the adaptive system at any stage of its learning are sure to be preserved (in the sense of refinement) as the system learns. More significantly, any behavior that is exhibited at the testing phase is sure to be preserved (i.e. duplicated or refined) in field usage. It is in this sense that monotonic learning is similar to testing. But monotonic learning is different from traditional program testing in a significant manner: whereas one can check the behavior of a traditional deterministic program by testing it on selected test data, testing a neural net before delivery on selected test data does not do us any good for the purposes of monotonic learning because monotonic learning depends on establishing properties of R_h, rather than properties of F_h^λ.

In principle, to apply monotonic learning we need to derive a closed form expression of R_h, then we derive the condition provided in definition 2 and prove it. Because it is rather impractical to derive a closed form of R_h, this brute force approach is unrealistic. As a substitute, we submit sufficient conditions for monotonic learning, starting with the following proposition, due to [20].

Proposition 4 *If the following condition holds,*

$$\forall\lambda\exists\lambda' : F^\lambda_{h.(x,y)} \sqsupseteq F^{\lambda'}_h,$$

then the pair (x, y) provides monotonic learning with respect to history h.

If function F_h^λ is total for all h and all λ, we get the following sufficient condition.

Proposition 5 *If F_h^λ is total for all h and λ and the following condition holds,*

$$\forall\lambda\exists\lambda' : F^\lambda_{h.(x,y)} = F^{\lambda'}_h,$$

then the pair (x, y) provides monotonic learning with respect to history h.

In other words, the learning pair (x, y) produces monotonic learning if and only if appending it to learning history h produces the same outcome as starting with some other parameter λ' and applying the learning history h.

4.2 Safe Learning

The main idea of *safe learning* is to ensure that as the adaptive system evolves through learning, it maintains some minimal safety property S. In other words, in addition to maintaining the identity

$$\forall h\forall\lambda, F_h^\lambda \sqsupseteq R_h,$$

which stems from the modeling of the system, we also require that the system maintains the following property

$$\forall h, F_h \sqsupseteq S$$

for some specification S, to ensure the safe operation of the adaptive system as it evolves through learning. By virtue of the lattice-like structure of the refinement ordering, we infer that F must satisfy:

$$\forall h\forall\lambda, F_h^\lambda \sqsupseteq (R_h \sqcup S).$$

This can be satisfied if and only if R_h and S do indeed have a join, i.e. if and only if they satisfy the consistency condition. These conditions can be maintained by placing restrictions on the learning algorithms that can be deployed, or by controlling learning data that gets fed into the adaptive system, as per the following inductive argument:

1. As the basis of induction, these conditions hold for $h = \epsilon$, since R_ϵ is the minimal element of the lattice of refinement.
2. Given that they hold for h, we can ensure that they hold for $h.(x, y)$ by accepting entry (x, y) only if it does not violate these conditions.

This matter is currently under investigation, with a special focus on the inductive possibilities that it offers.

4.3 Inductive Alternatives

Most traditional program verification methods tackle the complexity of the task at hand by doing induction on some dimension of program structure (control structure, data structure, depth of recursion, etc). Likewise, while the two methods we present here appear attractive, we have no doubt that they are complex in practice, because they rely on an explicit formulation of the functional envelope of the system. Hence we are focusing our attention on means to use induction in such a way that we can apply these methods without having to derive R_h. The key to the inductive approach is the ability to derive inductive relationships between R_h and $R_{h.(x,y)}$.

5 Concrete Computational Models

To build on the foregoing discussion, we have considered three distinct neural network models, and analyzed how their verification may proceed using the results presented above; these models are MLP (Multi Layer Perceptron), RBF (Radial Basis Function), and DCS (Dynamic Cell Structure). Due to space limitations, we will present only the two first.

5.1 Multi Layer Perceptron

The back-propagation algorithm was first developed by Werbos in 1974 [30] and was later independently rediscovered by Parker [25] in 1982 and by Rumelhart, Hinton and Williams [28] in 1986. The version we present below, taken from [13], is due to [28].

- **Weight Initialization**. Set all weights and node thresholds to small random numbers.
- **Calculation of Activation**.
  1. The activation level of an input unit is determined by the instance presented to the network.
  2. The activation level O_j of a hidden and output unit is determined by
  $$O_j = F(\sum W_{ji}O_i - \theta_j),$$
  where W_{ji} is the weight from an input O_i, θ_j is the node threshold, and F is the sigmoid function:
  $$F(a) = \frac{1}{1+e^{-a}}.$$
- **Weight Training**.
  1. Start at the output units and work backward to the hidden layers recursively. Adjust weights by
  $$W_{ji}(t+1) = W_{ji}(t) + \Delta W_{ji},$$
  where $W_{ji}(t)$ is the weight from unit i to unit j at time t and ΔW_{ji} is the weight adjustment.
  2. The weight change is computed by
  $$\Delta W_{ji} = \eta \delta_j O_i,$$
  where η is a trial-independent learning rate ($0 < \eta < 1$) and δ_j is the error gradient at unit j.
  3. The error gradient is given by:
     - For the output units:
     $$\delta_j = O_j(1-O_j)(T_j - O_j),$$
     where T_j is the desired (target) output activation and O_j is the actual output activation at output unit j.
     - For the hidden units:
     $$\delta_j = O_j(1-O_j)\sum_k \delta_k W_{kj},$$
     where δ_k is the error gradient at unit k to which a connection points from hidden unit j.
  4. Repeat iterations until convergence in terms of the selected error criterion.

We interpret this algorithm as defining function F_h (see section 3) by induction on the complexity (length) of h. If we recognize that F_h is not entirely determined by h but is also dependent on the arbitrary initial parameters (and their subsequent manipulations) then we rewrite this function as F_h^λ, where λ is the vector of weights

$$\lambda = \begin{pmatrix} .. \\ .. \\ W_{ji} \\ .. \\ .. \end{pmatrix}.$$

Also, we recognize that the range of values that weights can take evolves as the algorithm proceeds, hence the term Λ in the equations of section 3 should, in fact, be indexed with h; to acknowledge this, we write it as Λ_h. Consequently, we find:

- Λ_ϵ, the initial set of possible weights, is defined by the *Weight Initialization* step in the back-propagation algorithm.
- $\Lambda_{h.(x,y)}$ is obtained from Λ_h by applying the function detailed in the *Weight Training* step of the back-propagation algorithm. Specifically, if we let WT be the function detailed in this step, which has the form

$$\begin{pmatrix} W_{ji}(t+1) \\ .. \\ \delta_j \end{pmatrix} = WT \begin{pmatrix} W_{ji}(t) \\ .. \\ \delta_j \\ O_j \end{pmatrix},$$

then $\Lambda_{h.(x,y)}$ can be defined as follows:

$$\Lambda_{h.(x,y)} = \{ \begin{pmatrix} .. \\ W_{ji}(t+1) \\ .. \end{pmatrix} \mid \begin{pmatrix} W_{ji}(t+1) \\ .. \\ \delta_j \end{pmatrix}$$

$$= WT \begin{pmatrix} W_{ji}(t) \\ .. \\ \delta_j \\ O_j \end{pmatrix} \wedge \begin{pmatrix} .. \\ W_{ji}(t) \\ .. \end{pmatrix} \in \Lambda_h \}.$$

In light of this, we rewrite the characterization of R_h as follows:

$$R_h = \bigsqcap_{\lambda \in \Lambda_h} F_h^\lambda.$$

Initial Weights	Input	Iteration Times with Output						
		10	**20**	**50**	**100**	**500**	**2000**	**Converge**
W_0 =1.0								**7285****
	(1,1)	0.95718	0.88763	0.64715	0.57213	0.50373	0.12861	0.04999388
	(1,0)	0.88929	0.75727	0.49946	0.48526	0.51578	0.88881	0.95669980
	(0,1)	0.88985	0.76131	0.50934	0.48949	0.51579	0.88868	0.95675480
	(0,0)	0.74329	0.58170	0.41602	0.45580	0.50102	0.09960	0.03909299
W_0=0.5								**7560****
	(1,1)	0.70029	0.58756	0.53496	0.52377	0.51087	0.14863	0.04999296
	(1,0)	0.60074	0.51103	0.48290	0.48596	0.49385	0.87160	0.95670090
	(0,1)	0.60987	0.52137	0.48944	0.48869	0.49424	0.87141	0.95675580
	(0,0)	0.55051	0.49504	0.48686	0.49964	0.51760	0.11487	0.03909090
W_0 =0.0								**8926****
	(1,1)	0.50565	0.50740	0.50880	0.50976	0.51116	0.50880	0.04999402
	(1,0)	0.48353	0.48525	0.48714	0.48836	0.48878	0.49985	0.95669870
	(0,1)	0.49364	0.49367	0.49203	0.49037	0.48888	0.50006	0.95675415
	(0,0)	0.51421	0.51434	0.51342	0.51227	0.51134	0.51613	0.03909125
W_0*								**8942****
	(1,1)	0.51080	0.51098	0.51101	0.51096	0.51118	0.50909	0.04999226
	(1,0)	0.48304	0.48416	0.48627	0.48794	0.48876	0.49888	0.95670134
	(0,1)	0.49480	0.49397	0.49197	0.49027	0.48885	0.49911	0.95675653
	(0,0)	0.51010	0.51027	0.51051	0.51073	0.51137	0.51662	0.03908928

Figure 2: One Hidden Layer MLP NN for XOR Problem Trained by BP Algorithm with Different Initial Weights W0

Note that while F_ϵ^λ reflects the arbitrary choice of an initial weighting, R_ϵ does not; it only reflects the learning algorithm and the specific network architecture.

In order to assess the variability of the system function with respect to the choice of initial weights, we have run an experiment on a simple back-propagation neural network with one hidden layer, and have submitted to it learning data about the *exclusive or* function. Also, we have selected the initial weights, and have observed how these affect the function F_h^λ for various values of h. The column labeled "10" in figure 2 represents the learning sequence h made up of ten epochs. By abuse of notation, we can represent h by the number of epochs in h. We can make the following observations: the initial weights have a large impact on the evolution of F_h^λ; this impact lasts well into the future, and does not completely disappear even after several thousand epochs. For the purposes of our study, this means that R_h remains distinct from F_h^λ even for a long learning sequence h. In order to apply the method of monotonic learning, we explore/ investigate sufficient conditions of monotonicity on this MLP model. We briefly present three sufficient conditions of monotonicity, which are discussed in [20].

- *The first learning pair produces monotonicity*.
- *Duplication produces monotonicity*.
- *Convergence produces monotonicity*.

All these conditions are very trivial, and work is under way to derive more useful sufficient conditions. We are also exploring inductive alternatives. In terms of the MLP neural net under consideration, we know the relation between successive weights (as defined by the *Weight Training* function, WT), the relation between a set of weights (λ) and the corresponding system function (F_h^λ) and we know how the functional envelope R_h is derived from system functions (by taking the meet for all values of λ). We must infer from this the relation between R_h and $R_{h.(x,y)}$. See figure 3.

5.2 The Radial Basis Function Network

Our discussion of RBF neural networks is based on [24], to which the interested reader is referred to for further details. A *radial basis function* is a real-valued function of n real-valued arguments whose value decreases (or increases) monotonically with the distance from a central point (in the linear space R^n). The most general formula for any radial basis function is:

$$b(x) = \phi((x-c)^T N^{-1}(x-c)),$$

where ϕ is the function used (Gaussian, multi-quadratic, etc), c is the center and N is the metric used to measure distance in R^n. The term $(x-c)^T N^{-1}(x-c)$ is the distance between x and c in the linear space R^n, measured

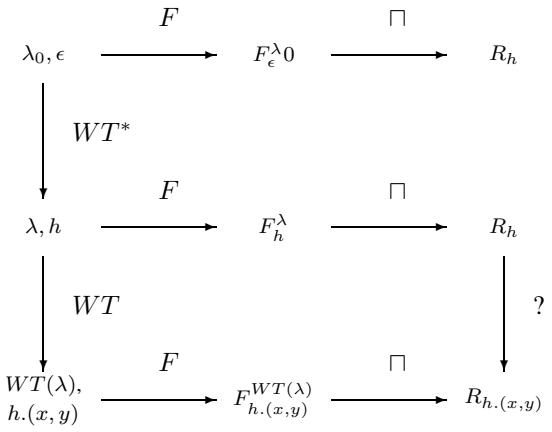

Figure 3: Inductive Structure

by norm N. The most commonly used norm is the Euclidian norm, defined by $N = r^2 I$, where I is the identity matrix of size n. As for the choice of function ϕ, common choices (among monotonic functions) include:

- Gaussian, $\phi(z) = e^{-z}$.
- Cauchy, $\phi(z) = (1+z)^{-1}$.
- Multi-quadratic, $\phi(z) = (1+z)^{\frac{1}{2}}$.
- Inverse multi-quadratic, $\phi(z) = (1+z)^{-\frac{1}{2}}$.

If we let n be 1 (i.e. we are interested in a function whose arguments are scalar), and let c and r be the center and radius of interest, then we obtain the following radial basis function:

$$b(x) = \phi(\frac{(x-c)^2}{r^2}).$$

Using radial basis functions, we derive a radial basis network as a combination of these functions, and optimize the multiplicative coefficients (called weights) of each function to provide a best fit for the learning data. In this paper, we limit our study to single layer networks where the radial basis functions are fixed (are not altered by the learning data). The function computed by a single layer RBF network is a linear combination of the radial basis functions, and (because the functions are fixed) the optimization of the weights amounts to solving a set of m linear equations in m unknowns, where m is the number of functions in the (single) layer of the network [24].

Given a neural network with n nodes (in a single layer), the decisions we must make to build a radial basis functions network are the following:

- Choosing the radius, r, within R^+.
- Choosing the center, c, within R^n.
- Choosing the monotonic function, from (say) the set discussed above: Φ = {Gauss, Cauchy, multi-quadratic, inverse multi-quadratic}.

We let Λ be the set of all possible choices of these parameters, i.e.

$$\Lambda = R^+ \times R^n \times \Phi^m,$$

and we designate by λ an arbitrary element of Λ. We let F_h^λ be the function computed by the neural network defined by the set of parameters λ after it has received history h. According to the abstract model discussed above, the *functional envelope* of the network is defined by:

$$R_h = \bigsqcap_{\lambda \in \Lambda} F_h^\lambda.$$

The crux of our approach is that for the purpose of correctness proofs, we choose to reason on R_h rather than on F_h^λ.

Given that the function of the RBF network is a linear combination of the radial basis functions, and given that these functions do not change with learning, the condition of proposition 5 can be satisfied if we ensure the identity of the linear coefficients; whence the following proposition.

Corollary 1 *We consider a single layer RBF network with radial basis functions $(b_1, b_2, ...b_n)$, and we let h be the learning history*

$$h = (x_1, y_1).(x_2, y_2)....(x_p, y_p)).$$

We let B be the $p \times n$ matrix defined by

$$B = \begin{pmatrix} b_1(x_1) & b_2(x_1) & ... & b_m(x_1) \\ b_1(x_2) & b_2(x_2) & ... & b_m(x_2) \\ ... & ... & ... & ... \\ b_1(x_p) & b_2(x_p) & ... & b_m(x_p) \end{pmatrix},$$

and we let A be defined by

$$A = B^T B,$$

and y be the vector of outputs (y_i), with $1 \leq i \leq p$. We consider a new learning pair (x, y) and we claim that (x, y) produces monotonic learning if

$$A^{-1} B^T y = A'^{-1} B'^T y',$$

where B', A' and y' are defined analogously to B, A and y, by replacing b_i by b_i' and adding a new pair (x, y) (hence a new row to B and a new entry to y).

To get some intuition for this corollary, we consider a very simple example, and analyze the sufficient condition of monotonicity. Specifically, we consider an RBF network with one input node (hence $n = 1$), two internal nodes in a single layer ($m = 2$) and one output node. Further, we let functions b_1 and b_2 be defined as follows:

$$b_1(x) = exp(-\frac{(x-c_1)^2}{r_1^2}),$$
$$b_2(x) = exp(-\frac{((x-c_2)^2}{r_2^2}),$$

for some constants c_1, r_1, c_2, and r_2. We let the target function be

$$f(x) = sin(2x) + 2sin(x)$$

and we let h be a learning history, i.e. a sequence of pairs of the form (x_i, y_i) where $y_i = f(x_i)$. We consider a new learning pair (x_{p+1}, y_{p+1}) and we analyze under what condition does the pair (x_{p+1}, y_{p+1}) produce monotonic learning with respect to history h. Corollary 1 provides that in order for (x_{p+1}, y_{p+1}) to produce monotonicity, coming after history h, the following condition must be satisfied:

$$A^{-1}B^T y = A'^{-1}B'^T y'.$$

This equation spells out the condition under which the new pair (which appears on the right hand side: x_{p+1} appears in B' and A'; y_{p+1} appears in y') provides monotonicity after h (which appears on both sides of the equation).

6 Conclusion

On-line learning systems in general, and their neural net implementations in particular are gaining increasing acceptance in control applications, which are often characterized by complexity and criticality. A significant obstacle to their acceptance and usefulness/ usability is the lack of adequate verification/ certification methods and techniques, as all traditional methods and techniques are inapplicable. In this paper we are presenting a tentative computational model for on-line learning systems and we use this model to sketch verification methods. Among the main contributions of our work, we cite:

- An abstract computational model that captures the functional properties of an evolving adaptive system by abstracting away random factors in the function of the system, to focus exclusively on details that are relevant to the learning algorithm and the learning history.
- The integration of this computational model into a refinement logic, which establishes functional properties of adaptive systems using refinement-based reasoning.
- The introduction of two venues for verifying adaptive systems: one based on *monotonic learning* (the *adaptive* equivalent of testing), and one based on *safe learning* (the *adaptive* equivalent of proving).
- An application of this reasoning framework to radial basis function neural networks, and the exploration of some sufficient conditions that allows us to make provable claims about the behavior of the neural network.

While this work is still in its infancy, we feel that it has introduced some meaningful concepts and has opened original venues for further exploration, by taking a refinement-based approach. We envisage the following extensions to this work:

- Experiment, be it on small examples, with the derivation of the functional envelope (R_h) of the system, and analyze what the conditions of monotonic learning and safe learning mean in practice. While it is easy to compute relation R_h extensionally, by listing some of its pairs (as we have done in figure **??**), it is not trivial to derive a closed form expression of it.
- Investigate means to obviate the need to derive an explicit closed form expression for R_h, by exploring inductive arguments that allow us to ensure monotonic learning and safe learning without computing the functional envelope.
- Fine-tune the proposed computational model for RBF neural networks, and analyze how this model helps us to make provable claims about such networks.
- Explore means to establish properties of the functional envelope of an RBF neural network at delivery time, by a combination of analytical and empirical methods.
- Derive tighter sufficient conditions for monotonicity for RBF networks, and further analyze the condition of safe learning.

This research is currently under way.

References

[1] J.R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] H. Ammar, B. Cukic, C. Fuhrman, and Mili. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. *Annals of Software Engineering*, 10, 2000.

[3] A. Avizienis. The n-version approach to fault tolerant software. *IEEE Trans. on Software Engineering*, 11(12), December 1985.

[4] N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.

[5] Ch. Brink, W. Kahl, and G. Schmidt. *Relational Methods in Computer Science*. Springer Verlag, New York, NY and Heidelberg, Germany, 1997.

[6] M. Caudill. Driving solo. *AI Expert*, pages 26–30, September 1991.

[7] C. H. Dagli, S. Lammers, and M. Vellanki. Intelligent scheduling in manufacturing using neural networks. *Journal of Neural Network Computing*, pages 4–10, 1991.

[8] J. Dean. Timing the testing of cots software products. In *First International ICSE Workshop on Testing Distributed Component Based Systems*, Los Angeles, CA, May 1999.

[9] Jules Desharnais, Ali Mili, and Thanh Tung Nguyen. Refinement and demonic semantics. In Brink et al. [5], chapter 11, pages 166–183.

[10] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[11] D. Bernard et al. Final report on the remote agent experiment. In *NMP DS-1 Technology Validation Symposium*, Pasadena, CA, February 2000.

[12] B. Fritzke. Growing self-organizing networks - why. In *European Symposium on Artificial Neural Networks*, pages 61–72, Brussels, Belgium, 1996.

[13] LiMin Fu. *Neural Networks in Computer Intelligence*. McGraw Hill, 1994.

[14] D. Gries. *The Science of programming*. Springer Verlag, 1981.

[15] H. Hecht, M. Hecht, and D. Wallace. Toward more effective testing for high assurance systems. In *Proceedings of the 2nd IEEE High Assurance Systems Engineering Workshop*, Washington, D.C., August 1997.

[16] Internet. Program verification system. Technical report, SRI International Computer Science Laboratory, 1997.

[17] R.C. Linger and P.A. Hausler. Cleanroom software engineering. In *Proceedings, 25th Hawaii International Conference on System Sciences*, Kauai, Hawaii, January 1992.

[18] R.C. Linger, H.D. Mills, and B.I. Witt. *Structured Programming*. Addison Wesley, 1979.

[19] M. Lowry, M. Boyd, and D. Kulkarni. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, 13th IEEE International Conference on Automated Software Engineering*, pages 322–331, Honolulu, HI, October 1998. IEEE Computer Society.

[20] A. Mili, B. Cukic, Y. Liu, and R. Ben Ayed. Towards the verification and validation of online learning adaptive systems. In Taghi Khoshgoftaar, editor, *Computational Methods in Software Engineering*. Kluwer Scientific Publishing, 2003.

[21] H.D. Mills, V.R. Basili, J.D. Gannon, and D.R. Hamlet. *Structured Programming: A Mathematical Approach*. Allyn and Bacon, Boston, Ma, 1986.

[22] C.C. Morgan. *Programming from Specifications*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.

[23] M. Napolitano, G. Molinaro, M. Innocenti, and D. Martinelli. A complete hardware package for a fault tolerant flight control system using on-line learning neural networks. *IEEE Control Systems Technology*, January 1998.

[24] Mark J. L. Orr. Introduction to radial basis function networks. Technical report, University of Edinburgh, www.anc.ed.ac.uk/mjo/papers/intro.ps, 1996.

[25] D.B. Parker. Learning logic. Technical Report S81-64, Stanford University, 1982.

[26] D. K. Pradhan. *Fault Tolerant Computing: Theory and Practice*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[27] Orna Raz. Validation of online artificial neural networks —an informal classification of related approaches. Technical report, NASA Ames Research Center, Moffet Field, CA, 2000.

[28] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume I: Foundations*. MIT Press, Cambridge, MA, 1986.

[29] Boeing Staff. Intelligent flight control: Advanced concept program. Technical report, The Boeing Company, 1999.

[30] P.J. Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. Technical report, Harvard University, 1974.