Backwards Differentiation in AD and Neural Nets: Past Links and New Opportunities[⋆]

Paul J. Werbos

National Science Foundation, Arlington, VA, USA
pwerbos@nsf.gov

Summary. Backwards calculation of derivatives – sometimes called the reverse mode, the full adjoint method, or backpropagation – has been developed and applied in many fields. This paper reviews several strands of history, advanced capabilities and types of application – particularly those which are crucial to the development of brain-like capabilities in intelligent control and artificial intelligence.

Key words: Reverse mode, backpropagation, intelligent control, reinforcement learning, neural networks, MLP, recurrent networks, approximate dynamic programming, adjoint, implicit systems

1 Introduction and Summary

Backwards differentiation or "the reverse accumulation of derivatives" has been used in many different fields, under different names, for different purposes. This paper will review that part of the history and concepts which I experienced directly. More importantly, it will describe how reverse differentiation could have more impact across a much wider range of applications.

Backwards differentiation has been used in four main ways known to me:

1. In automatic differentiation (AD), a field well covered by the rest of this book. In AD, reverse differentiation is usually called the "reverse method" or the "adjoint method." However, the term "adjoint method" has actually been used to describe two different generations of methods. Only the newer generation, which Griewank has called "the true adjoint method," captures the full power of the method.
2. In neural networks, where it is normally called "backpropagation" [532, 541, 544]. Surveys have shown that backpropagation is used in a majority

⋆ The views herein are those of the author, not the official views of NSF. However – as work done by a government employee on government time, it is in the open government domain.

of the real-world applications of artificial neural networks (ANNs). This is the stream of work that I know best, and may even claim to have originated.
3. In hand-coded "adjoint" or "dual" subroutines developed for specific models and applications, e.g., [534, 535, 539, 540].
4. In circuit design. Because the calculations of the reverse method are all local, it is possible to insert circuits onto a chip which calculate derivatives backwards physically on the same chip which calculates the quantit(ies) being differentiated. Professor Robert Newcomb at the University of Maryland, College Park, is one of the people who has implemented such "adjoint circuits." Some of us believe that local calculations of this kind must exist in the brain, because the computational capabilities of the brain require some use of derivatives and because mechanisms have been found in the brain which fit this idea.

These four strands of research could benefit greatly from greater collaboration. For example – the AD community may well have the deepest understanding of how to actually calculate derivatives and to build robust dual subroutines, but the neural network community has worked hard to find many ways of *using* backpropagation in a wide variety of applications.

The gap between the AD community and the neural network community reminds me of a split I once saw between some people making aircraft engines and people making aircraft bodies. When the engine people work on their own, without integrating their work with the airframes, they will find only limited markets for their product. The same goes for airframe people working alone. Only when the engine and the airframe are combined together, into an integrated product, can we obtain a real airplane – a product of great power and general interest.

In the same way, research from the AD stream and from the neural network stream could be combined together to yield a new kind of modular, integrated software package which would *integrate* commands to develop dual subroutines *together with* new more general-purpose systems or structures making use of these dual subroutines.

At the AD2004 conference, some people asked why AD is not used more in areas like economics or control engineering, where fast closed-form derivatives are widely needed. One reason is that the proven and powerful tools in AD today mainly focus on differentiating C or Fortran programs, but good economists only rarely write their models in C or in Fortran. They generally use packages such as Troll or TSP or SPSS or SAS, which make it easy to perform statistical analysis on their models. Engineering students tend to use MatLab. Many engineers are willing to try out very complex designs requiring fast derivatives, when using neural networks but not when using other kinds of nonlinear models, simply because backpropagation for neural networks is available "off the shelf" with no work required on their part. A more general kind of integrated software system, allowing a wide variety of user-specified modeling modules, and compiling dual subroutines for each module type and collections of modules, could overcome these barriers. It would not be necessary to work hard to wring out the last 20 percent reduction in run time, or even to cope with strange kinds of spaghetti code written by users; rather, it would be enough to provide this service for users who are willing to live with natural and easy requirements to use structured code in specifying econometric or engineering models, etc. Various types of neural networks and elastic fuzzy logic [542] should be available as choices, along with user-specified models. Methods for combining lower-level modules into larger systems should be part of the general-purpose software package.

The remainder of this paper will expand these points and – more importantly – provide references to technical details. Section 2 will discuss the motivation and early stages of my own strand of the history. Section 3 will summarize backwards differentiation capabilities we have developed and used.

For the AD community, the most important benefit of this paper may be the new ways of *using* the derivatives in various applications. However, for reasons of space, I will weave the discussion of those applications into Sects. 2 and 3 and provide citations and URLs to more information.

This paper does not represent the official views of NSF. However, many parts of NSF would be happy to receive more proposals to strengthen this important emerging area of research. For example, consider the programs listed at `www.eng.nsf.gov/ecs`. Success rates all across the relevant parts of NSF were cut to about 10% in fiscal year 2004, but more proposals in this area would still make it possible to fund more work in it.

2 Motivations and Early History

My personal interest in backwards differentiation started in the 1960s, as an outcome of my desire to better understand how intelligence works in the human brain.

This goal still remains with me today. NSF has encouraged me to explain more clearly the same goals which motivated me in the 1960s! Even though I am in the Engineering Directorate of NSF, I ask my panelists to evaluate each proposal I receive in the program for Controls, Networks, and Computational Intelligence (CNCI) by considering (among other things) how much it would contribute to our ability to someday understand and replicate the kind of intelligence we see in the higher levels of the brains of all mammals.

More precisely, I ask my panelists to treat the ranking of proposals as a kind of strategic investment decision. I urge them to be as tough and as complete about focusing on the bottom line as any industry investor would be, except that the bottom line, the objective function, is not dollars. The bottom line is the *sum* of the potential benefits to fundamental scientific understanding, plus the potential broader benefits to humanity. The emphasis is on *potential* – the risk of losing something really big if we do *not* fund a particular proposal. The questions “What is mind? What is intelligence? How can we replicate and understand it as a whole system?” are at the top of my list of what to look for in CNCI. But we are also looking for a wide spectrum of technology applications of strategic importance to the future of humanity. See my chapter in [479] for more details and examples.

Before we can reverse-engineer brain-like intelligence as a kind of computing system, we need to have some idea of what it is trying to compute. Figure 1 illustrates what that is:

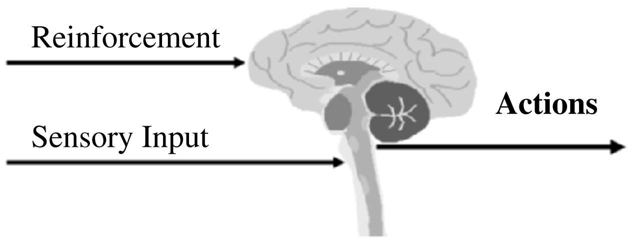

Fig. 1. The brain as a whole system is an intelligent controller.

Figure 1 reminds us of simple, trivial things that we all knew years ago, but sometimes it pays to think about simple things in order to make sure that we understand all of their implications. To begin with, Fig. 1 reminds us that the entire output of the brain is a set of nerve impulses that control *actions*, sometimes called “squeezing and squirting” by neuroscientists. The entire brain is an information processing or computing device. The purpose of any computing device is to compute its outputs. Thus the function of the brain *as a whole system* is to learn to compute the actions which best serve the interests of the organism over time. The standard neuroanatomy textbook by Nauta [404] stresses that we cannot really say *which* parts of the brain are involved in computing actions, since *all* parts of the brain feed into that computation. The brain has many interesting capabilities for memory and pattern recognition, but these are all *subsystems* or even *emergent dynamics* within the larger system. They are all subservient to the goal of the overall system – the goal of computing effective actions, ever more effective as the organism learns. Thus the design of the brain as a whole, as a computational system, is within the scope of what we call “intelligent control” in engineering. When we ask how the brain works, as a functioning engineering system, we are asking how a system made up of neurons is capable of performing learning-based intelligent control. This is the species of mathematics that we have been working to develop – along with the subsystems and tools that we need to make it work as an integrated, general-purpose system.

Many people read these words, look at Fig. 1, and immediately worry that this approach may be a challenge to their religion. Am I claiming that all human consciousness is nothing but a collection of neurons working like a conventional computer? Am I assuming that there is nothing more to the human mind – no "soul?" In fact, this approach does not require that one agree or disagree with such statements. We need only agree that mammal brains actually do exist, and do have interesting and important computational capabilities. People working in this area have a great diversity of views on the issue of "consciousness." Because we do not need to agree on that complex issue, in order to advance this mathematics, I will not say more about my own opinions here. Those who are interested in those opinions may look at [532, 548, 549], and at the more detailed technical papers which they cite.

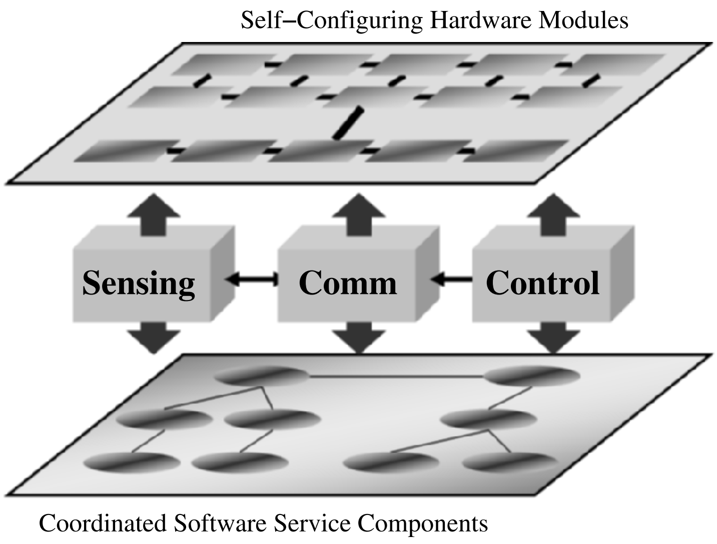
Self–Configuring Hardware Modules
Sensing
Comm
Control
Coordinated Software Service Components

Fig. 2. Cyberinfrastructure: the entire web from sensors to decisions/action/control designed to self-heal, adapt and learn to maximize overall system performance.

Figure 2 depicts another important goal which has emerged in research at NSF and at other agencies such as the Defense Advanced Projects Agency (DARPA) and the Department of Homeland Security (DHS) Critical Infrastructure Protection. More and more, people are interested in the question of how to design a new kind of "cyberinfrastructure" which has the ability to integrate the entire web of information flows from sensors to actuators, in a vast distributed web of computations, which is capable over time to learn to optimize the performance of the actual physical infrastructure which the cyberinfrastructure controls. DARPA has used the expression "end-to-end learning" to describe this. Yet this is *precisely the same design task* we have been addressing all along, motivated by Fig. 1! Perhaps we need to replace the word "reinforcement" by the phrase "current performance evaluation" or the like, but the actual mathematical task is the same.

Many of the specific computing applications that we might be interested in working on can best be seen as *part* of a larger computational task, such as the tasks depicted in Figs. 1 or 2. These tasks can provide a kind of *integrating framework* for a general purpose software package – or even for a hybrid system composed of hardware and software together. See `www.eng.nsf.gov/ecs/` for a link to recent NSF discussions of cyberinfrastructure.

Figure 3 summarizes the origins of backpropagation and of Artificial Neural Networks (ANNs). The figure is simplified, but even so, one could write an entire book to explain fully what is here.

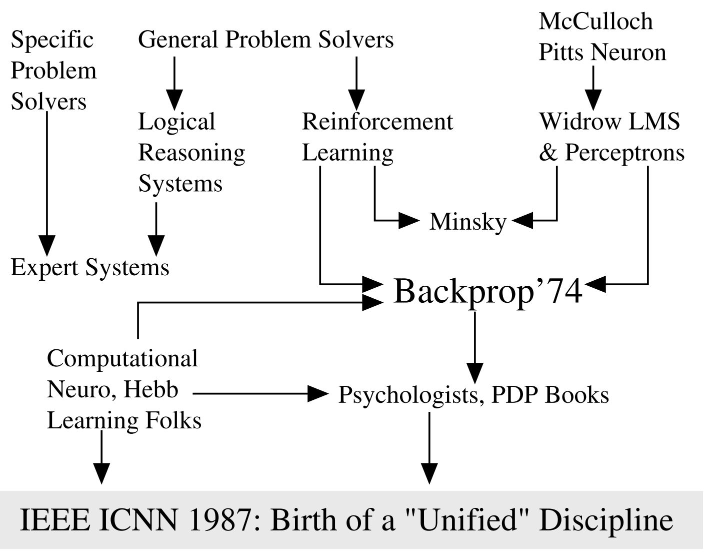

Fig. 3. Where did ANNs and backpropagation come from?

Within the ANN field proper, it is generally well-known that backpropagation was first spelled out explicitly (and implemented) in my 1974 Harvard Ph.D. thesis [532]. (For example, the IEEE Neural Network Society cited this in granting me their Pioneer Award in 1994.)

Many people assume that I developed backpropagation as an answer to Marvin Minsky's classic book *Perceptrons* [370]. In that book, Minsky addressed the challenge of how to train a specific type of ANN – the Multilayer Perceptron (MLP) – to perform a task which we now call Supervised Learning, illustrated in Fig. 4.

In supervised learning, we try to learn the nonlinear mapping from an input vector $\underline{X}$ to an output vector $\underline{Y}$, when given *examples* $\underline{X}(t)$, $\underline{Y}(t)$, $t = 1$ to T of the relationship. There are many varieties of supervised learning, and it remains a large and complex area of ANN research to this day, with links to statistics, machine learning, data mining, and so on.

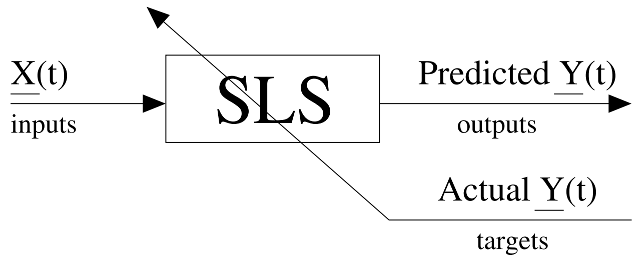

Fig. 4. What a Supervised Learning System (SLS) does.

Minsky's book was best known for arguing that 1) we need to use an MLP with a hidden layer even to represent simple nonlinear functions such as the XOR mapping; and 2) no one on earth had found a viable way to *train* MLPs with hidden layers good enough even to learn such simple functions. Minsky's book convinced most of the world that neural networks were a discredited dead-end – the worst kind of heresy. Widrow has stressed that this pessimism, which squashed the early "perceptron" school of AI, should not really be blamed on Minsky. Minsky was merely summarizing the experience of hundreds of sincere researchers who had tried to find good ways to train MLPs, to no avail. There had been islands of hope, such as the algorithm which Rosenblatt called "backpropagation" (not at all the same as what we now call backpropagation!), and Amari's brief suggestion that we might consider least squares as a way to train neural networks (without a discussion of how to get the derivatives, and with a warning that he did not expect much from the approach). But the pessimism at that time became terminal.

In the early 1970s, I visited Minsky at MIT and proposed a joint paper showing that MLPs can overcome the earlier problems if 1) the neuron model is slightly modified [534] to be differentiable; and 2) the training is done in a way that uses the reverse method, which we now call backpropagation [532, 544] in the ANN field. But Minsky was not interested [8]. In fact, no one at MIT or Harvard or any place else I could find was interested at the time.

There were people at Harvard and MIT then who had used, in control theory, a method very similar to the *first-generation* adjoint method, where calculations are carried out backwards from time T to $T-1$ to $T-2$ and so on, but where derivative calculations *at any time* are based on classical forwards methods. (In [532], I discussed first-generation work by Jacobsen and Mayne [282], by Bryson and Ho [75], and by Kashyap, which was particularly relevant to my larger goals.) Some later debunkers have argued that backpropagation was essentially a trivial and obvious extension of that earlier work. But in fact, some of the people doing that work actually controlled computer resources at Harvard and MIT at that time, and would not allow those resources to be used to test the ability of true backpropagation to train ANNs for supervised learning; they did believe there was enough evidence in 1971 that true backpropagation could possibly work.

In actuality, the challenge of supervised learning was not what really brought me to develop backpropagation. That was a later development. My initial goal was to develop a kind of universal neural network learning device to perform a kind of "Reinforcement Learning" (RL) illustrated in Fig. 5.

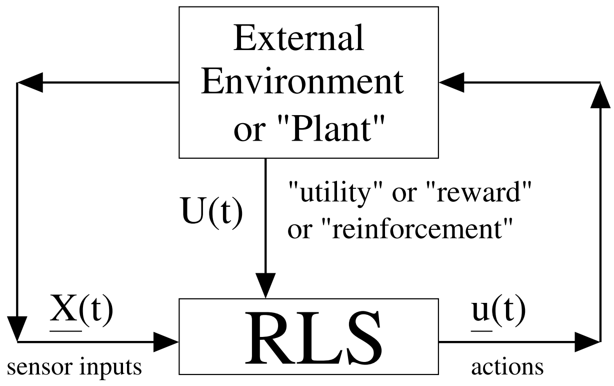
External Environment or "Plant"
U(t)
"utility" or "reward" or "reinforcement"
X(t)
sensor inputs
RLS
u(t)
actions

Fig. 5. A Concept of Reinforcement Learning. The environment and the RLS are both assumed to have memory at time t of the previous time $t-1$. The goal of the RLS is to learn how to maximize the sum of expected U ($\langle U \rangle$) over all future time.

Ironically, my efforts here were inspired in part by an earlier paper of Minsky [172], where he proposed reinforcement learning as a pathway to true general-purpose AI. Early efforts to build general-purpose RL systems were no more successful than early efforts to train MLPs for supervised learning, but in 1968 [531] I proposed what was then a new approach to reinforcement learning. Because the goal of RL is to maximize the sum of $\langle U \rangle$ over future time, I proposed that we build systems explicitly designed to learn an approximation to *dynamic programming*, the only exact and efficient method to solve such an optimization problem in the general case. The key concepts of classical dynamic programming are shown in Fig. 6.

In classical dynamic programming, the user supplies the utility function to be maximized (this time as a function of the state $\underline{x}(t)$!), and a stochastic model of the environment used to compute the expectation values indicated by angle brackets in the equation. The mathematician then finds the function J which solves the equation shown in Fig. 6, a form of the Bellman equation. The key theorem is that (under the right conditions) any system which chooses $\underline{u}(t)$ to solve the simple, static maximization problem within that equation will automatically provide the optimal strategy over time to solve the difficult problem in optimization over infinite time. See [479,546,553] for more complete discussions, including discussion of key concepts and notation in Figs. 6 and 7.

My key idea was to use a universal function approximator – like a neural network – to *approximate* the function J or something very similar to it, in

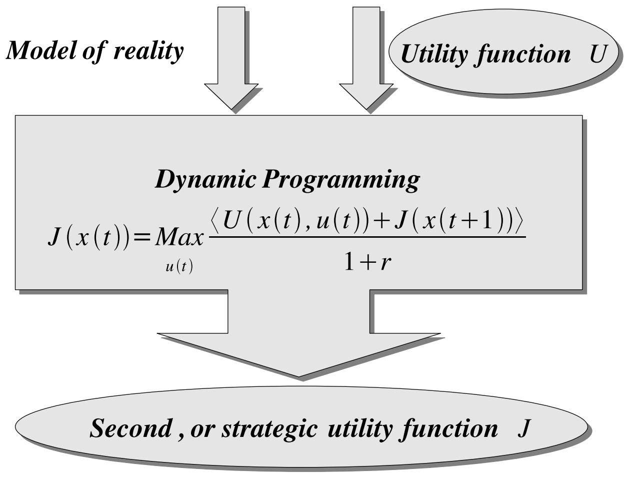
Model of reality
Utility function U
Dynamic Programming
$J(x(t)) = \underset{u(t)}{Max} \frac{\langle U(x(t),u(t)) + J(x(t+1)) \rangle}{1+r}$
Second, or strategic utility function J

Fig. 6. The key concepts in classical dynamic programming.

order to overcome the curse of dimensionality which keeps classical dynamic programming from being useful on large problems.

In 1968, I proposed that we somehow imitate Freud's concept of a backwards flow of credit assignment, flowing back from neuron to neuron, to implement this idea. I did not provide a practical way to do this, but in my thesis proposal to Harvard in 1972, I proposed the following design, including the flow chart (with less modern labels) and the specific equations for how to use the reverse method to calculate the required derivatives indicated by the dashed lines.

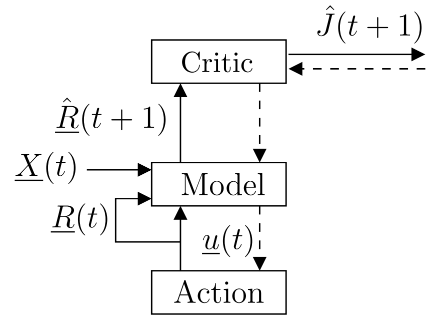

Fig. 7. RLS design proposed to Harvard in my 1972 thesis proposal.

I explained the reverse calculations using a combination of intuition and examples and the ordinary chain rule, though it was almost exactly a translation into mathematics of things that Freud had previously proposed in his theory of psychodynamics! Because of my difficulties in finding support for this kind of work, I printed up many copies of this thesis proposal and distributed them very widely.

In Fig. 7, all three boxes were assumed to be filled in with ANNs – with ordered computational systems containing parameters or weights that would be adapted to approximate the behavior called for by the Bellman equation. For example, to make the actions $\underline{u}(t)$ actually perform the maximization which appears in the Bellman equation, we needed to know the derivatives of J with respect to every action variable (actually, every parameter in the action network). The derivatives would provide a kind of specific feedback to each parameter, to signal whether the parameter should be increased or decreased. For this reason, I called the reverse method "dynamic feedback" in [532]. The reverse method was needed to compute all the derivatives of J with respect to all of the parameters of the action network in just one sweep through the system. At that time, I focused on the case where the utility function U depends only on the state $\underline{x}$, and not on the current actions $\underline{u}$. I discussed how the reverse calculations could be implemented in a local way, in a distributed system of computing hardware like the brain.

Harvard responded as follows to this proposal and to later discussions. First, they would not allow ANNs as such to be a major part of the thesis, since I had not found anyone willing to act as a mentor for that part. (I put a few words into Chapter 5 to specify essential ideas, but no more.) Second, they said that backwards differentiation was important enough by itself for a Ph.D. thesis, and that I should postpone the reinforcement learning concepts for research after the Ph.D. Third, they had some skepticism about reverse differentiation itself, and they wanted a really solid, clear, rigorous proof of its validity in the general case. Fourth, they agreed that this would be enough to qualify for a Ph.D. if, in addition, I could show that the use of the reverse method would allow me to use more sophisticated time-series prediction methods which, in turn, would lead to the first successful implementation of Karl Deutsch's model of nationalism and social communications [146]. All of this happened [532], and is a natural lead-in to the next section.

The computer work in [532] was funded by the Harvard-MIT Cambridge Project, funded by DARPA. The specific multivariate statistical tool described in [532], made possible by backpropagation, was included as a general command in the MIT version of the TSP package in 1973-74 and, of course, described in the MIT documentation. The TSP system also included a kind of small compiler to convert user-specified formulas into Polish form for use in nonlinear regression. By mid-1974 we had almost finished coding a new set of commands (almost exactly paralleling [532]) which: (1) would allow a TSP user to specify a "model" as a set of user-specified formulas; (2) would consolidate all the Polish forms into a single compact structure; (3) would provide the obvious kinds of capabilities for testing a whole model, similar to capabilities in Troll; and (4) would automatically create a reverse code for use in prediction and optimization over time. The complete system in Fortran was almost ready for testing in mid-1974, but there was a complete reorganization of the Cambridge Project that summer, reflecting new inputs from DOD and important improvements in coding standards based on PL/1. As I result, I graduated and moved on before the code could be moved into the new system.

3 Types of Differentiation Capability We Have Developed

3.1 Initial (1974) Version of the Reverse Method

My thesis showed how to calculate all the derivatives of a *single* computed quantity Y with respect to *all* of the inputs and parameters which fed into that computation in *just one sweep* backwards (see Fig. 8).

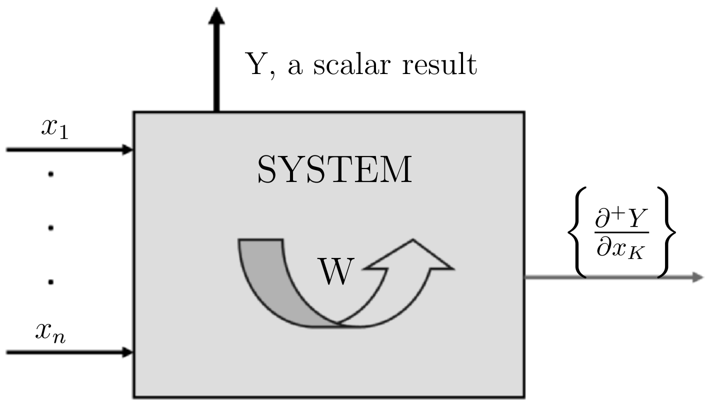

Fig. 8. Concept of the reverse method.

The first version of the reverse method required that the computational system be what I called an "ordered system." My definition of an "ordered system" in Chapter 2 of [532] was almost identical to the definition of an explicit computational algorithm given by Louis Rall in his chapter in this book [454]. At each time when we compute the scalar result Y, we need to be able to specify a sequence of intermediate computations f_1 through f_N which lead up to $Y = f_{N+1}$, where each computation is specified as a differentiable (and hopefully simple) function of what preceded it. In practice, these computations may form a kind of lattice of computations performed in parallel. However, that is just a useful and important special case of the general mathematics.

In order to specify and prove the validity of the reverse method, in the general case, I needed to define the concept of an *ordered derivative.* As shown in Fig. 8, the reverse method calculates the *entire set* of ordered derivatives of Y with respect to the set of inputs x_1 through x_n.

Many people at AD2004 asked how the reverse method could be better taught in schools. I would propose that *the very first course in calculus* that teaches partial derivatives should teach that there are at least three different *types* of partial derivative. The three different types make different assumptions, and need to be treated as distinct cases with distinct rules, to avoid confusion in the practical use of partial derivatives. I have seen enough confusion about partial derivatives in the study of complex systems, all across social sciences and basic science and engineering, that I believe it would save a lot of time in the end to be clear about these distinctions from the first.

The three basic concepts are: 1) the *algebraic* partial derivative, whose value (as an algebraic expression) depends on the *explicit algebraic expression* for the quantity being differentiated; 2) the *field* or functional partial derivative, whose value is well-defined only for a specific *set* of coordinate variables or input vector; and 3) the *ordered* derivative, which represents the *total* change in a *later* quantity which results when the value of an *earlier* quantity is changed, in an ordered system. Ordered derivatives occur in practice across all fields of science, but a confusing multitude of ad hoc terms and partial methods have been developed to deal with them. Again, it would save time to deal with the concept in a more unified and general way in basic calculus courses.

$$\begin{tikzcd}
z_3 \\
z_2 \arrow[u, "1"'] \\
z_1 \arrow[u, "3"'] \arrow[uu, bend right=60, "4"']
\end{tikzcd}$$

$$\frac{\partial z_3}{\partial z_2} = 1 \qquad \frac{\partial z_3}{\partial z_1} = 4$$

$$\frac{\partial^+ z_3}{\partial z_1} = \frac{\partial z_3}{\partial z_1} + \frac{\partial^+ z_3}{\partial z_2}\frac{\partial z_2}{\partial z_1} = 7$$

$$\frac{\partial^+ z_n}{\partial z_i} = \sum_{j=i+1}^{n} \frac{\partial^+ z_n}{\partial z_j}\frac{\partial z_j}{\partial z_i}$$

Fig. 9. The chain rule for ordered derivatives.

Figure 9 illustrates the relation between direct or algebraic partial derivatives and ordered derivatives, and gives the chain rule for ordered derivatives. In my view, the chain rule for ordered derivatives should be taught in second-year calculus classes. The proof of the chain rule in Chapter 2 of [532] (reprinted in [540]) is the proof of the validity of the reverse method. The reverse method *is* the use of this chain rule for the case of ordered systems. Notice that the direct or algebraic derivative of z_3 with respect to z_1 is only 4, because that is the direct impact along the outer arrow. However, the total or ordered derivative is 7.

For a system with n inputs as in Fig. 8, the reverse method allows one to compute all the required derivatives exactly in one pass, instead of the n passes needed with older methods. Thus it reduces costs by a factor of n. The person funding my work in the late 1970s argued that reductions in computational cost were growing less and less important, as computer costs fell. I replied that greater computing capacity is properly leading us to build ever larger models and modeling systems and control systems; thus as n grows larger and larger, the cost reduction becomes more and more important. For systems as large as the brain, the cost reduction is indispensable.

3.2 Extensions of the Reverse Method (1974-86)

During 1974-1986, I developed three kinds of extension to the reverse method: 1) extensions to calculate derivatives through "recurrent" or "implicit" systems; 2) extensions to calculate selected higher-order derivatives or even derivatives of eigenvalues or eigenvectors; and 3) extensions to manage block structured or modular computer systems. I used and published the method in several specialized areas – but interest became much broader after a detailed 1980 DOE/EIA Validation Report summarizing their capabilities and a condensed summary [534], which we distributed very widely.

Recurrent or Implicit Systems

The neural network community talks a lot about "feedforward networks," which sound identical at first to "ordered systems." A feedforward network would contain N elementary processing elements or "neurons," like the functions f_k above. At each time, the network would take n inputs (as in Fig. 8) and work forward step by step to compute its outputs. There may be more than one output, but still it is an ordered system. Neural network people often picture such a network as a kind of computational graph made up of circles and arrows (for example, see [534] or `www.nd.com`). Each circle represents the calculation of an intermediate variable, and the arrows flowing into any circle show us which earlier results are directly used in that calculation.

A "recurrent network," in neural network language, is a network which cannot be ordered, because the graph contains arrows "pointing backwards" (or looping back to the same level they start in.) The idea of recurrent or recursive neural networks was known back in Minsky's time [370]. The commonest form of recurrence is a loop *from* neuron number k *back to itself.*

The literature on recurrent networks has become very confused and often inaccurate, in part because there are different interpretations of what it means when people insert a backwards arrow into the computational graph. There are three common versions of what a backwards loop might mean: (1) a time-lagged flow of information – for example, when the calculation of neuron k at time t depends on the previous output at time $t-1$ of the same neuron; (2) an instantaneous flow of information, such that the network must be interpreted as an *implicit system*, as a system of nonlinear simultaneous equations such that the output of the system is defined as the result of solving those equations; or (3) a flow of information in continuous time, governed by ordinary differential equations (ODE).

I have defined a Time-Lagged Recurrent Network (TLRN) as a feedforward system augmented by the first kind of recurrence. I have defined a Simultaneous Recurrent Network (SRN) as a feedforward system augmented by the second kind of recurrence. The most general case, for systems based on discrete time, is a *hybrid* TLRN/SRN, where both kinds of recurrence are present. I have worked at times with the ODE versions [539], but this usually causes more trouble than it is worth (except in certain stability proofs in control [546]). The current lack of reliable software to handle TLRN/SRN hybrids effectively is a major barrier to progress in making better use of ANNs, in my view. Time-lagged recurrence and simultaneous recurrence each provide fundamentally different kinds of modeling or computational capability. For maximum (brain-like) overall capability, it is essential to be able to combine these two capabilities without blurring the distinction between them.

TLRNs are still ordered systems, if one considers the entire web of calculations across time. Later, I defined the term "backpropagation through time"(BPTT) [541] to refer to the use of backpropagation across an ordered space-time system. Of course, the cost of a complete and exact backwards sweep to get all the derivatives is still of the same order as the cost of a forwards sweep. BPTT was implemented in [532], and numerous examples were given of ways to use it. TLRNs trained using BPTT, along with sophisticated ways of *using* the derivatives, are the core of some of the most powerful applications of ANNs today. For example, the work by Feldkamp, Prokhorov, and others at Ford Research contains many examples of the effective use of TLRNs.

True implicit systems are a more difficult case. Perhaps the easiest way to think about implicit systems is to use the definition of SRN given in Sect. 3.2.4 of [553], with minor revision. An SRN may be defined as a vector-valued mapping $\underline{F}$:

$$\underline{Y} = \underline{F}(\underline{X}, W) \tag{1}$$

defined as the result of applying a "read-out function" $\underline{g}$:

$$\underline{Y} = \underline{g}(\underline{y}^{(\infty)}, \underline{X}, W) \tag{2}$$

to the converged value $\underline{y}^{(\infty)}$ of a vector $\underline{y}$, which we update by an iteration rule

$$\underline{y}^{(n+1)} = \underline{f}(\underline{y}^{(n)}, \underline{X}, W) \ , \tag{3}$$

where $\underline{f}$ is a feedforward system, and W is a set of weights or parameters, together with some procedure for determining the initial iterate $\underline{y}^{(0)}$. I sometimes call $\underline{f}$ the "feedforward core" of the SRN.

In 1980, soon after starting work for the Office of Energy Information Validation at the Energy Information Administration (EIA) of the Department of Energy, I encountered two examples of such implicit systems: (1) an econometric model of the natural gas industry [539], which included time-lagged effects but was defined as a simultaneous-equation system, like most standard econometric models; and (2) the Long-Term Energy Analysis Package (LEAP) [535], which was a large simultaneous-equation system operating forwards and backwards through time. I had responsibility for managing two large contracts which included sensitivity analysis of such models, one at MIT [31] (for econometric models) and one at Oak Ridge National Laboratories (ORNL) evaluating LEAP.

The ORNL group had studied the best current literature on the first-generation adjoint sensitivity methods, some of which they forwarded to me. Extending that approach, they calculated "sensitivity coefficients" (ordered derivatives) for LEAP, by calculating the Jacobian of $\underline{f}$, in effect, and iterating over the gradient of (3).

Looking at this work, I realized immediately that I could *combine* their approach to addressing the simultaneous equations aspect, *together with* the use of the reverse method applied to the feedforward core in order to avoid Jacobian calculations, and together with BPTT to handle the time-lagged effects in a normal econometric model. I implemented this new unified method as follows. First, I translated the current EIA model of natural gas markets and natural gas regulation from FORTRAN into a model in the Troll system. (This took some time, but was much appreciated by EIA management, because it made it much easier for them to know precisely what was assumed inside this model.) Then I hand-coded the dual or adjoint code to go with the model, as another "model" in Troll, so that I could quickly compute the sensitivity of any model result to *all* of the many inputs and parameters of the system. The results were written up in an EIA report "published" as an energy validation report, distributed within DOE and ORNL and a few other places, and theoretically distributed to the general public. The resulting journal article [539] was delayed due to the (verified) finding that the predicted residential gas price could vary by \$1 or more, in response to changes of only 0.001 in one of the elasticity parameters. The group which I managed at ORNL soon after became a primary source for the second-generation adjoint sensitivity methods. The person I exchanged papers with the most in this group retired after making a large amount of money on the stock market using neural network methods to guide his investments.

The method described in the final section of [539] is very close to the "white box method for implicit systems" as now used in the AD community. The presentation of the method in Chapters 3 and 10 of [553] may be somewhat easier to work with than [539].

SRNs are not widely used yet in ANN technology, even though many important applications will require them for real success. Part of the problem is a lack of suitable software and a need for research into how to speed up the learning. (Kozma, Ilin, and I have recently had preliminary results cutting learning time by a factor of ten, compared with [426, 551], using the simplest partial version of some new approaches.)

Another part of the problem is a widespread lack of understanding of what SRNs can offer, if properly trained.

Most ANN users know that simple MLPs are "universal approximators." Andrew Barron of Yale [20] has proven theorems about *how many parameters* are needed to achieve a given level of approximation accuracy, when approximating smooth functions. In essence, he has proven that the required complexity grows exponentially with the number of inputs for linear basis function approximators (such as lookup tables, Taylor series, or radial basis function approximators). However, it grows much more slowly for the simple MLP. Many neural network people conclude: if MLPs are so effective in approximating any input-output mapping, why bother with the extra complexity of an SRN?

However, many tasks critical to intelligent systems require that we approximate nonsmooth functions or functions with high computational complexity. Minsky's "connectedness" function [370] is one example. Evaluating a position in a game like Go is another example. The SRN provides a kind of Turing-like capability [553] that ensures it has the most general kind of representation ability we need in practice, and we often do need it.

To try to demonstrate this, Pang and I [426, 551] showed how a simple SRN could learn to solve a kind of "generalized maze navigation" task, where MLPs and the Simple Recurrent Networks later proposed by psychologists both failed very badly. In [551] I showed how an SRN trained on six easy mazes could steadily improve its performance on six hard mazes on which it was never trained. In [426], we exhaustively discussed the five major approaches to computing the derivatives needed to train an SRN structure (four applicable to TLRNs as well). We actually used the "black box" approach in this early demonstration. The black box approach – treating the iterations of (3) as if they were time points, and using BPTT – takes more memory than the "white box" approach, but it was simple and exact, and did not lead into the tricky pitfalls of the white box method discussed at AD 2004.

In the work of [426, 551], we used a special kind of SRN, a "cellular SRN," suitable for situations where the inputs come from a kind of two-dimensional grid with translational symmetry. More recently I have developed and patented a more general special case of SRN, called an ObjectNet, for situations where the inputs may come from a more general class of relational networks (such as the state of electric power grids). ObjectNets can be used to train a single network to learn from a training set consisting of data from different power grids with different topologies and different numbers of state variables – and yet they are still an inherently distributed computational structure, like the cellular SRN.

One may also ask: how could the *brain* calculate the derivatives it needs to train time-lagged recurrences, since it cannot memorize its entire life history as one time series, and it does not seem to sweep back through time in the same way that BPTT does? In [426,553], we describe an approach to approximating the required derivatives in forwards time, called the "Error Critic." Also, when systems need to learn over very long time intervals T, there may be a close relation between the kinds of multi-scale time representation we need for intelligent control [545] and the kind we need for memory management or "checkpointing." As a simple example, if $T = 2^n$, we could live with only n "memory records" in an exact BPTT, by allocating one record to hold $T/2$, one to hold $3T/4$ initially but later $T/4$, and so on, in a scheme similar to binary search. The brain may not be so limited in its memory capacity, but the organization of its information may lead to some interesting parallels.

Modular Structure, Higher-Order Derivatives and Eigenvalues

Up to now, I discussed how to calculate the ordered first derivatives of a scalar quantity of interest Y as depicted in Fig. 8, for systems which are ordered (Sect. 3.1) or recurrent (Sect. 3.2). But what if the system of interest has more than one output of interest? What if we need higher-order derivatives? For reasons of space, I cannot present all the extensions I have worked with, but I can give some examples and citations.

First, consider the example of Fig. 4. A supervised learning system usually has several outputs Y_1 through Y_n, forming a vector $\underline{Y}$. How can we apply the capability shown in Fig. 8 where there is only a single output of interest?

The SLS shown in Fig. 8 may be a neural network *or any other* system which may be represented by a vector-valued function $\underline{F}$:

$$\underline{\hat{Y}}(t) = \underline{F}(\underline{X}(t), W) \ , \tag{4}$$

where W represents the *set* $\{W_\alpha\}$ of weights or parameters to be adapted. There are many ways to adapt such systems. In "vanilla backpropagation" or in "basic backpropagation" [541, 544] in real-time, we adapt the weights W_α to reduce the square error of the prediction of $\underline{Y}(t)$ at the current time

$$E(t) = \sum_{i=1}^{n} \frac{1}{2}\left(\hat{Y}_i(t) - Y_i(t)\right)^2 \ . \tag{5}$$

For truly brain-like capability, it is very important to modify this approach by adding penalty terms (e.g., those of Phatak) and by accounting for the related issues addressed by various authors involving loss functions [448], robustness [31], empirical risk [518] or dynamic robustness [532, 552, 553] and by allowing for a kind of interplay between learning from current experience and learning from memory as in what I have called syncretism [553], which is related to Atkeson's memory-based learning.

In basic backpropagation, the scalar quantity of interest is $E(t)$. The system to be differentiated is not the SLS itself but the *combination* of the SLS and (5). Thus we can apply the reverse method directly. As a practical matter, it is important to write clean modular code here. Modular code becomes ever more important as we work our way up to more complex applications.

In writing modular code, we would like to use a name for each variable as close as possible to the label we use in the mathematical papers that describe the system, but computer languages will not let us use "$\partial^+ E(t)/\partial W_\alpha$", for example, as a variable name. Thus I have used the shorthand notation "F_W_α", for example, to represent the $\underline{f}$eedback to the quantity W_α, the ordered derivative of the current quantity of interest with respect to W_α. (In the AD community, people might use "ad_W_α" instead.)

For developing a modular version of basic backpropagation [541], I have shown how the reverse calculations can be split up into two parts. The first part, in the main program, calculates:

$$F\_\hat{Y}_i \equiv \frac{\partial^+ E(t)}{\partial \hat{Y}_i(t)} = \frac{\partial E(t)}{\partial \hat{Y}_i(t)} = \hat{Y}_i - Y_i \ , \tag{6}$$

for $i = 1$ to n. Then a dual subroutine, a dual or adjoint to the SLS, works back the implied ordered derivatives with respect to all of the inputs or weights. (To minimize run costs, we may sometimes code two versions or entries to the dual subroutine, one of which only calculates feedback to the weights, for cases where that is all we need.) The dual subroutine inputs the entire set of F_Y variables, but it implements a single reverse calculation aimed at a single scalar quantity of interest further downstream.

Only about 12 people have fully implemented structures like Fig. 7 so far, because it requires us to keep track of three main scalar quantities of interest, the specific error measure used in training the Critic, the error measure used in training the Model, and the estimate of J itself as used to train the Action network. People who use off-the-shelf neural network software without really understanding the reverse method find it difficult to keep track of the complexity. Often I explain the system by discussing how to adapt the three parts in three separate sections, so that I discuss only one error measure in each section. Many other explanations, examples, and applications appear in [479]. In my parts of [553], I give the more general case more explicitly, by using dual subroutines in the specification of algorithms, so that a user can select any mixture of neural networks, elastic fuzzy logic, or user-specified systems of equations for any of the components. I have sometimes wondered whether I should use notation like FJ_W_α, FEJ_W_α, and FEX_W_α to explicitly describe how systems like Fig. 7 require us to consider three quantities of interest at the same time. In 1986 [536], I described some early ideas for how one might implement capabilities like this as a user-friendly systems of commands in the SAS system. There are tutorial slides with text which progress from simple pattern recognition and data mining methods, through to diagnostics and time-series issues, through to many generations and types of methods for decision and control [547].

Chapter 10 of [553] discusses the issues which arise when we try to train "Models" (as in Fig. 7) which can predict partially observed systems over time, and chapter 13 discusses a stochastic extension (the "Stochastic Encoder/Decoder/Predictor") which may be thought of as a kind of nonlinear maximum likelihood factor analysis system in the special case where the Predictor is set to zero. Backwards differentiation is essential to making these kinds of complex capabilities workable in realistic computing systems or in realistic chip-level or distributed hardware implementations. Jose Principe at the University of Florida (`www.ece.ufl.edu/facultystaff/principe.html`) also has interesting ideas. Feldkamp and Prokhorov of Ford recently did benchmark studies where even simple TLRNs performed state estimation as well as more expensive "particle filter" methods and better than Extended Kalman Filters. Still, more work is needed to unify the pieces needed when unobserved variables are partly continuous and partly discrete.

As a further example, people sometimes use supervised learning to learn nonlinear relationships in a statistical way from real observed data, but supervised learning can also be useful as a way to develop a kind of reduced order model of a more complex model. For example, one may use it to *approximate* a large model running on a supercomputer by a neural network model which could fit on the $1-10 neural chip designed and tested by Raoul Tawel of the Jet Propulsion Laboratory and Mosaix LLC (funded by an NSF SBIR grant with encouragement from Ford). In such cases, however, you can get better results by developing an adjoint for the large model as well, so that the training set includes $\underline{X}(t)$, $\underline{Y}(t)$, and the Jacobian of $\underline{Y}(t)$ with respect to $\underline{X}(t)$ for each example t. One can minimize the augmented error function:

$$E(t) = \sum_{i=1}^{n} \frac{1}{2}\Big(\hat{Y}_i(t) - Y_i(t)\Big)^2 + \frac{1}{2}\sum_{j=1}^{m} C_j \sum_{i=1}^{n} \left(\frac{\partial^+ \hat{Y}_i}{\partial X_j} - \frac{\partial^+ Y_i}{\partial X_j}\right)^2 , \tag{7}$$

where the input vector $\underline{X}$ has m components, and the nonnegative weights C_j can be chosen in various ways.

I have called this method "Gradient Assisted Learning" (GAL). To minimize this error function, one must in effect calculate *its* derivatives, which involve ordered second derivatives. Chapter 10 of [553] discusses how to do so in some detail. (See also [538].) The easiest general approach is simply to note that the calculation of $E(t)$ is itself an ordered system, even though some of its intermediate calculations are *motivated* by derivative calculation. One can apply the reverse method directly to *that* ordered system. Similar issues arise in implementing a control method which I call Globalized Dual Heuristic Programming (GDHP), where I discussed (less clearly) how to get the second derivatives [533, 534, 537]. GDHP now seems most important as a way to handle decision or control problems where the actions $\underline{u}(t)$ include both discrete and continuous choices [479]. See [546] for some discussion of stability theory and links to control theory.

Sensitivity analysis and convergence of large models was a major application when I was at EIA, calling for many kinds of derivatives for many uses. See [534] and [535] for examples. For example, combining the reverse method with the Fadeev formulas for derivatives of eigenvalues and eigenvectors yielded interesting information. All of the methods here can provide important practical information in global modeling packages, such as one would use for long-range strategic planning where it is important to know the impact of current decisions or policies on long-term global outcomes, and to generate value measures or “shadow prices” to guide optimal decisions by tactical or distributed agents.