Good morning. I would like to thank you all for coming here this morning. I would especially like to thank David Parker and Bernie Widrow for inviting me here this morning, and for providing me with moral support and logistic support without which none of this would be possible. Before I get started, I would also like to emphasize that I am speaking this morning as a private individual, not as a representative of DOE or NSF. In fact, my paper was written before I even came to work at NSF. Tomorrow night I will speak as an NSF program director, but not today.

BACKPROPAGATION: PAST AND FUTURE

- CAN WE DESIGN/UNDERSTAND INTELLIGENCE?

  — HISTORY AND STRATEGY

  — 3 NEW TOOL BOXES

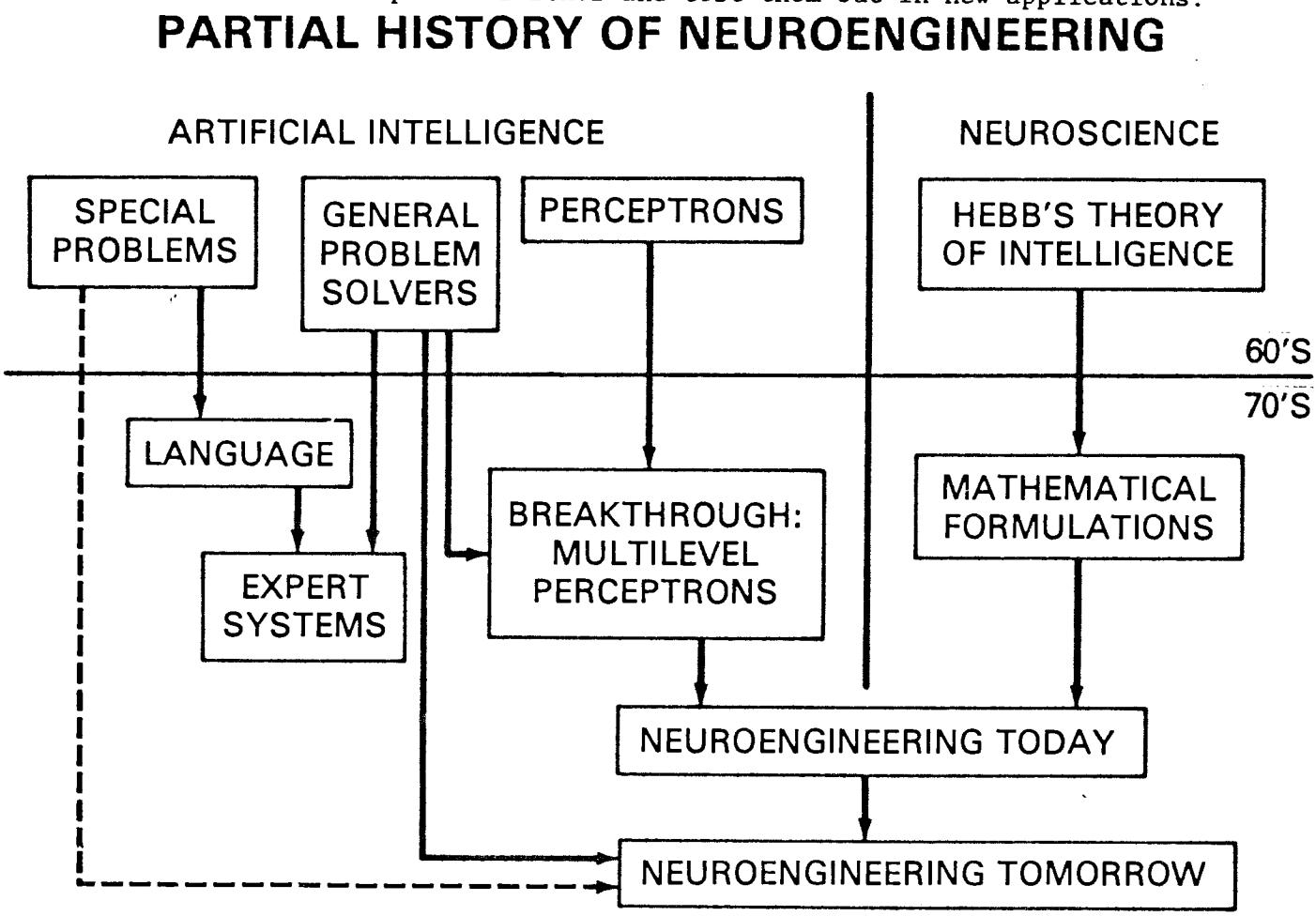
Slide 1:

My topic this morning is "Backpropagation: Past and Future", with an emphasis on the future. The main question I will focus on this morning is this one: "How can we use backpropagation and its extensions to build intelligent systems, to build artificial systems which demonstrate the same general kind of intelligence that we see in the human brain?"

Now to be perfectly honest, my personal interest here is not so much in building artificial systems (though I'm glad that there are some applications out there to help justify this work) but in developing the kind of mathematics that we need in order to improve our understanding of the human mind. Like some of the biologists here, I became interested in this field more than twenty years ago, after reading D.O Hebb's classic book on the brain. Hebb's book was really quite interesting,

and it contains a whole lot more than just a learning rule. After reading that book, I concluded immediately that Hebb's original learning rule could never reproduce the complex kinds of behavior that Hebb was talking about, so I resolved to look for algorithms which really could, on a neuronal kind of basis. And that's what got me started on all this.

So this morning, I'm going to start out (point) by discussing the kinds of strategy I've been using to approach this question -- a strategy which has really begun to work out in the past few years. And then I'm going to talk about three specific boxes of tools which are needed to take us up from what we're doing today with backpropagation all the way through to the construction of brain-like intelligent systems. These tools already exist in rudimentary form, and many of them have already been proven to work in realistic applications, but a lot more work is needed to develop them further and test them out in new applications.



Slide 2:

Now... the next slide shows a brief history of where neuroengineering came from. A lot of people have complained about all the hype that neuroengineering has received in the last year or two, but twenty-five years ago there was ten times as much hype about another new, emerging technology -- artificial intelligence. Back in those days, people were asking exactly the same question I just referred to -- "How can we build artificial systems with the same kind of intelligence that the

human brain possesses?"

There were three different ways of trying to answer this question, which led to three different strands of research. On the one hand, some people tried to concentrate on solving specific problems which are generally considered to require intelligence. For example, they developed algorithms and computer programs to play better and better games of chess. Some of the early work on these lines -- like Samuels' famous checker-playing program -- developed some very important general insights, but after awhile the field began to get cluttered with hundred-page assembly-language programs specialized to narrow applications.

On the other hand, a second school of thought -- the general problem-solving school -- argued that there is no intelligence in these kinds of specific algorithms. They argued that intelligent systems produce algorithms in much the same way that a clam grows a clamshell. The life is in the clam, not in the shell. Likewise, the intelligence is in the system which learns and develops the algorithm, not in the algorithm itself.

The theory of this approach was very good, but in practice it ran into a very large obstacle. How do you design a generalized machine to do anything, without specifying anything at all about what it is supposed to do? So as a practical matter, the people working in this school also had to specify specific mathematical problems, problems which they hoped would be generic enough in character to approximate what the brain does. They focused on two such problems: first, the problem of generalized symbolic reasoning -- a problem which people are still working on today; and second, a problem which people are now calling reinforcement learning.

In reinforcement learning, we ask a system to control some kinds of overt actions -- like physical movements -- over time, so as to maximize some measure of performance or reinforcement. All of these words could be discussed at great length, but let me focus on two of them for now: "over time." There is a lot of work in robotics which focuses on the connection between actions at any time and results at the same time -- the kind of connection which you worry about in posture control or hand-eye coordination, for example; however, the hard part of this problem -- the core of the design problem -- is in accounting for the effect of actions at one time on results at a later time.

Some psychologists believe that human intelligence really is a matter of reinforcement learning, almost entirely. Others -- like myself -- believe that the human mind does possess other attributes which go well beyond reinforcement learning in higher brain centers. In some of my papers, I have even tried to discuss some of these attributes. Nevertheless, reinforcement learning clearly is part of what the human brain does, and it is a worthwhile starting point in trying to understand the overall structure of the brain.

A third school of thought -- the perceptron school -- tried to copy over what was known about neurons in the 1950's and 1960's. In those days, people thought that they knew how neurons worked, more or less -- following the classic model of McCulloch and Pitts. People strung together networks of McCulloch-and-Pitts neurons, and tried to develop learning rules to make these networks perform useful functions.

None of these schools of thought were able to live up to all of the early hype about AI. People discovered very quickly that you can't build an intelligent system by mindless hacking or by using a few quick tricks. Nevertheless, progress did continue in this field. For example, many of you have heard -- from David Parker or Bart Kocko or Stephen Grossberg -- that I myself developed the backpropagation method back in the 1970's. This didn't happen as a result of my staring at the existing perceptron designs, and trying to come up with a better learning rule; a lot of people had already tried that in the past. It happened because I was looking for a way to do reinforcement learning, and I realized very quickly from the mathematics that derivatives are essential to efficient maximization and minimization. I knew that McCulloch and Pitts neurons were not differentiable, but I also knew enough about neurophysiology to know that the McCulloch-Pitts model was already becoming obsolete.

This point about McCulloch and Pitts is very important. Yesterday, Marvin Minsky said that he himself tried to do reinforcement learning in the 1960's, and that he thought about using derivatives. But he listened to the neuron modellers of his time, who said that the McCulloch-Pitts model was right. On the other hand, I has the advantage of actually looking at firing data from cortical neurons, from the hard-core neuroscience literature. It didn't look anything like a string of digital pulses. Instead, it looked like a regular sequence of "bursts" or "volleys" of a set of spikes all scrunched together, which varied continuously in intensity. And that's how I knew to question the McCulloch-Pitts model.

Beyond that, I knew enough about the credit-assignment problem and related ideas from control theory and Freudian psychology to translate all of this directly into a method for adapting neural networks. Believe it or not, there is a close connection between backpropagation and Freud's ideas about "psychic energy," which I mentioned in my 1968 Cybernetica paper.

KEY HISTORY CITES (ICNN PAPER):

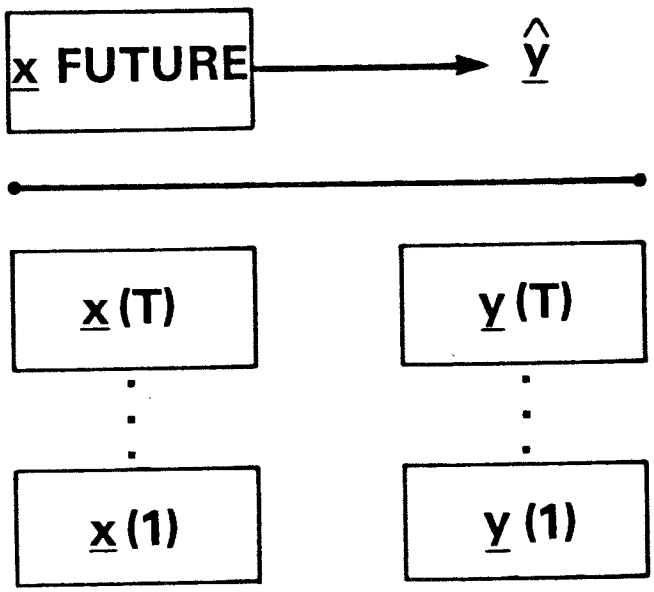
- 1968 CONCEPTS, CYBERNETICA
- 1971/72 RHW-LINE BUT REINFORCEMENT
- 1974 THESIS: GENERAL NETWORKS
- 1981 IFIP: OPTIMIZATION IN NEURAL NET
- 1988 DOE: GAS MODEL DOCUMENTATION

Slide 3: Now...

The next slide shows a few highlights of my work on backpropagation. I'm not going to go into detail about who shot John, and that sort of thing; those who are interested in that kind of history can tree back from the references in my paper, especially including the DOE gas model documentation.

In 1971 to '72, I developed a formulation of backpropagation nearly identical to the Rumelhart, Hinton and Williams version, except for the notation. One of the reviewers, at Harvard, stated that this method would be very interesting for a seminar paper, but was not important enough to warrant a Harvard Ph.D. In response to his comments, I then generalized backpropagation to apply to any network of differentiable functions, and applied it to a realistic problem in political forecasting, using what people now call a recurrent structure. Finally, in 1981, I discussed the neural network applications of this method in some detail, at an international conference as large and as important as this one, cited in my paper. And there were other publications.

SUPERVISED LEARNING = REGRESSION



Slide 4:

The next slide illustrates what people are now doing with backpropagation. They are using it mainly to solve the supervised learning problem, which is set up exactly like the classical problem of nonlinear least squares, which statisticians have known about for decades.

In supervised learning, you start out with a lot of trial patterns (point). Each of these patterns consists of an input vector, which I call "x", and a target vector, which I call "y". This is standard statistical notation. The object of the exercise is to adjust the synapse weights, or parameters, of the model network, to make its outputs as close as possible to the target vectors over these patterns. The goal is to do this in such a way that good outputs will be produced in the future, when the network is applied to new input vectors and the target vectors are unknown (point). Again, this is simply a special case of a well-known problem in statistics.

$$E_p = \tfrac{1}{2} \Sigma (t_{pi} - s(net_{ip}))^2 \quad i>N$$

$$net_{ip} = W_{i0} + \Sigma W_{ij}\, s(net_{jp}) \quad j>m$$

$$\Delta W_{ij} = \alpha \frac{\partial E_p}{\partial W_{ij}} + \text{MOMENTUM}$$

Slide 5:

The next slide shows how Rumelhart and all set up the supervised learning problem for backpropagation. The first equation defines the error function that they try to minimize. They try to minimize the sum of squared error, where error is defined as the difference between target values -- which they call "t" -- and the outputs of the final layer of neurons. In their notation, the array called "net" -- n, e, t -- represents the activation level of the neuron, and "s of net" -- where s is the sigmoid function -- represents its actual output.

The second equation on this slide gives the rule for calculating "net" for all the neurons in the network (except for the input neurons). The final equation shows how the weights in the network are adjusted over time. Except for the momentum term -- whose importance has been debated -- the last equation is simply the well-known steepest descent rule for minimizing functions.

What I've left off of this slide are the equations to calculate the derivatives of error with respect to the weights. Those are actually the key equations here; the use of square error and the use of steepest descent had been known and tested decades before.

Back when I tried to generalize this approach to an arbitrary network, I asked the following question: What if the error function, or loss function, were any arbitrary differentiable function of the inputs and the weights, instead of square error? What if the elementary units, or neurons, could each independently represent any differentiable function of their inputs or weights? What if inputs from earlier time periods were allowed? Still assuming a feedforward network, how could we calculate all the derivatives in a single pass? The next slide shows the answer.

$$L(x_1, \ldots, x_N)$$

$$x_i = f_i(x_1, \ldots, x_{i-1})$$

$$\frac{\partial^+ L}{\partial x_i} = \frac{\partial L}{\partial x_i} + \sum_{j>i} \frac{\partial^+ L}{\partial x_j} \cdot \frac{\partial f_j}{\partial x_i}$$

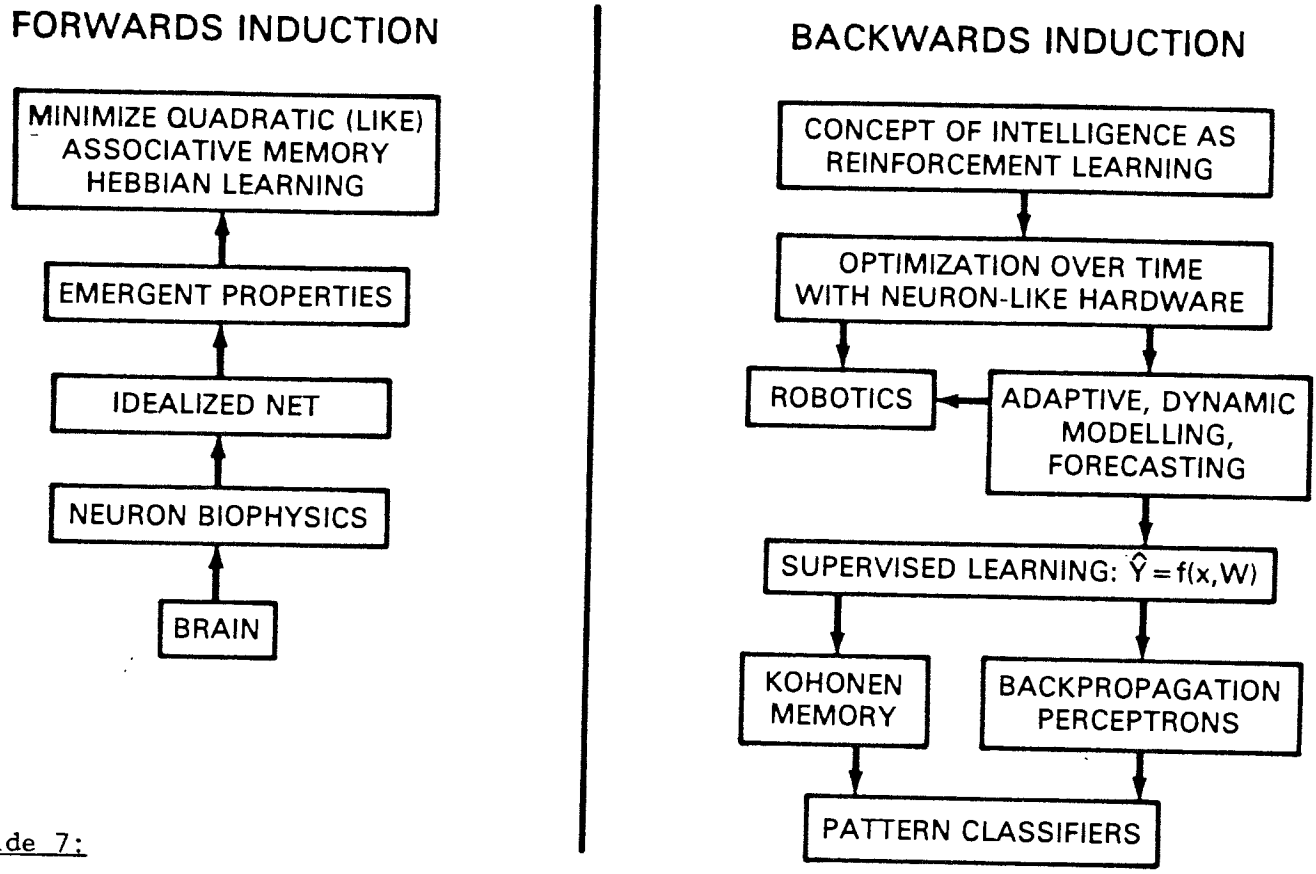
Slide 6:

Suppose that L is any differentiable loss function or error function, or even just a function that we are trying to maximize. Suppose that the set of quantities x-sub-i includes all of the outputs of all of the units in the system, at all times, as well as all of the inputs, including the weights of the network. Then the third equation here -- the chain rule for ordered derivatives -- is the proper recurrence equation to use in calculating all of the derivatives, with respect to all of the weights, in a single pass.

Note that there are two kinds of derivative here -- ordered derivatives, which have little plus signs on them, and conventional partial derivatives. This is not the ordinary chain rule from first-year calculus. Some people have argued that Rumelhart's version of backpropagation falls out directly from applying the chain rule; that is true, if you apply this chain rule, but the ordinary chain rule will not do the job in a rigorous way, unless you do an awful lot of fancy footwork. As long as you are looking at simple systems, you can get by without rigor, by using your intuition; however, when things get complex, one's intuition can be confused, and a lack of rigor can lead to unnecessary difficulty and error.

This chain rule is useful when working with econometric models or engineering problems, where people have some knowledge about the kinds of functions they want to use. But more importantly, it also makes it very easy to work with recurrent networks, second-order derivatives, and other things I talk about in my paper. My paper works out some of the equations for backpropagation for a few of these structures, but you would frankly be much better off using this chain rule yourselves, directly. If you need some reassurance about how to use it, my paper cites three other papers, all describing natural gas applications, which provide varying levels of tutorial on this equation.

NEUROENGINEERING



Slide 7:

With this slide, I will end my discussion of history and strategy, and start my discussion of specific tools.

This slide is borrowed from NSF, and we can ignore the left-hand side for now.

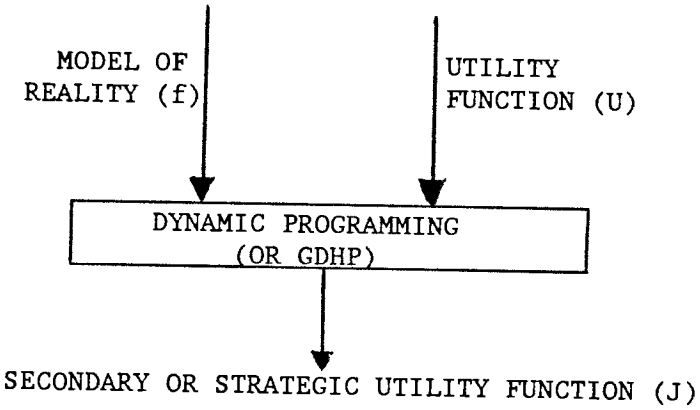
The right-hand side basically shows the logical path which I encountered, in the 1960's and 70's, working my way down from the goal of reinforcement learning over time, through to the subgoal of neural nets for causal modelling, which led to the further subgoal of supervised learning, and which finally led to backpropagation. Note that pattern classification was always a kind of afterthought, a way of using supervised learning, but very much a special case.

In order to work our way back up to reinforcement learning over time, we need to take three major steps, concurrently. First, we need to develop better learning rules for supervised learning, learning rules which account for the insights of Kohonen and learning rules which learn a lot faster. Second, we need to face up to the time-series problem directly, and exploit the long experience of statisticians and people like myself in working with models or networks to predict things over time. Note that this kind of learning could be viewed as a kind of unsupervised learning, where a system is learning how its environment actually works, without being rewarded or punished or told what to do. In addition to

myself, Shastri and Watrous have also done work on this level, using backpropagation and accounting for recurrent effects across time; their success in speech recognition experiments has shown very graphically how important this work is to real-world applications. And finally, we need to build networks which use time-series models, and learn to control actions over time so as to maximize some kind of utility function. In addition to myself, this area has been looked at seriously by Barto, Sutton and Anderson, and by Uno of Kawato et al. As noted on this slide, it has important implications for areas like robotics and factory automation.

So these are the three bags of tools. This morning, I will start out by talking about optimization over time, mainly because it sets up the context and because you will want me to be brief on that this morning. Next, I will talk about improvements in supervised learning. And finally, I will talk about neural networks for causal modelling or prediction, and squeeze in a few words about real brains if there is time.

So: my next slide has to do with optimization over time.

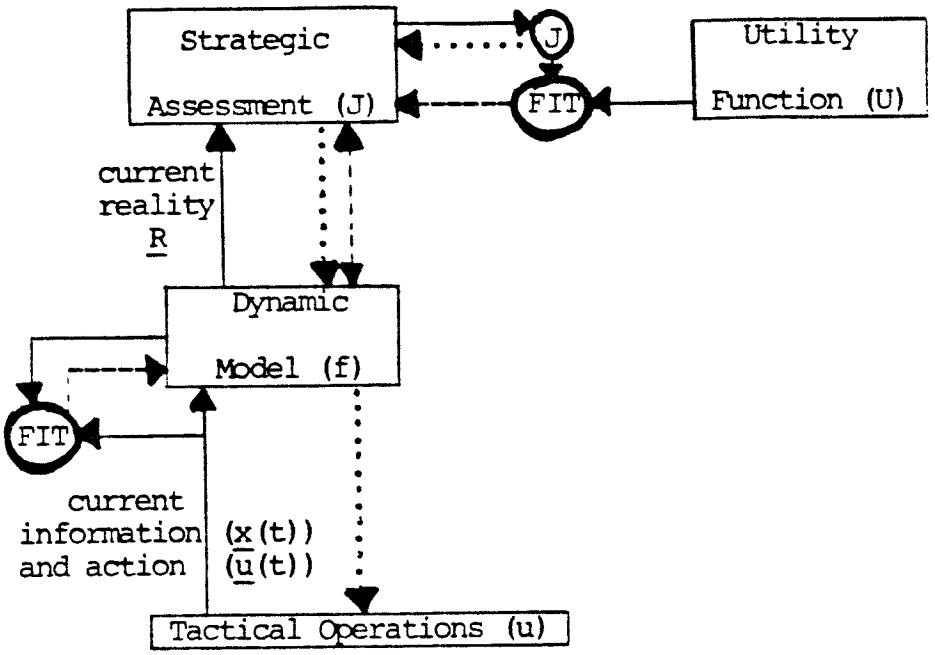


Slide 8:

This slide illustrates the basic trick used in dynamic programming, which is one of the fundamental tools used in control theory. Dynamic programming is the only exact and efficient method available to control actions or movements over time, so as to maximize a utility function in a noisy, nonlinear environment, without making highly specialized assumptions about the nature of that environment.

Dynamic programming requires as its input a utility function U and a model of the external environment, which I denote as little "f". Dynamic programming produces, as its major output, another function, J, which I like to call a secondary or strategic utility function. The key insight in dynamic programming is that you can maximize the function U, in the long-term, over time, simply by maximizing this function J in the immediate future. Whenever you know the function J, and the model f, it is a simple problem in function maximization to pick the actions which maximize J.

Unfortunately, it is impossible to apply dynamic programming exactly for complicated problems, because the calculations become hopelessly complex. However, it is possible to approximate these calculations, by using a model or a network to estimate the J function or its derivatives. We can find an error function for this network which can only be zero exactly when the output of the network satisfies the equations of dynamic programming, and we can adapt the network so as to minimize that error function. I have developed three different families of method to do this, all of them defined for general networks, which would include neural networks as a special case. The simplest family of methods -- which I call heuristic dynamic programming -- is described in my paper, in neural sorts of language. A more complex and powerful method, which I call GDHP, is described in an earlier paper which I cite, and it requires the calculation of second derivatives through backpropagation. To apply GDHP to neural networks, I would recommend going back to the earlier paper for the theory and the loss function, but using the differentiation tricks in my current paper to set up the neural networks. GDHP can be a whole lot faster than heuristic dynamic programming for complicated problems where cause-and-effect relations are understood to some degree.



Slide 9:

At any rate, the next slide shows the kind of architecture which all of these methods lead to, in the general case. To build an intelligent system based on these methods, we need to build three subnetworks, each adapted by something like backpropagation, though it is possible to allow some overlap between the networks. One network has to represent a model of the external environment. One has to estimate the J function. A third has to exercise direct control over movement, using other variables as input. My earlier paper spelled out the gory details of how to adapt the top and bottom networks, with GDHP, but it leaves open the problem of how to adapt the middle network -- the model of the environment. Furthermore, the use of backpropagation with the top and bottom networks leads to many of the same issues as you encounter with supervised learning.

This approach to reinforcement learning over time is very similar to the "adaptive critic" approach used by Barto, Sutton and Anderson. My "J network" corresponds to their "adaptive critic." Given that Barto, Sutton and Anderson have had great success with this approach, in simple tests to date, there is every reason to expect that the general forms of these methods can work out very well in practical applications. However, even though I have applied backpropagation to real-world optimization problems, in a production environment, I have yet to work on the stochastic kinds of examples which really require these general methods.

By the way, Ron Williams yesterday raised the question of how to do backpropagation through a stochastic unit, which arises in some forms of reinforcement learning. There is a simple way to handle this problem, buried in my SMC Transactions paper. The basic idea is to treat random numbers from fixed distributions as an _input to the network (in the formal mathematics); no other random numbers are needed, to generate any distribution you like for the output of the network.

SUPERVISED LEARNING TOOLS:

— IMPROVED CONVERGENCE

— MERGE KOHONEN INSIGHT

Slide 10:

Now I would like to move on to the question of supervised learning. As I said before, there are two main issues here -- the speed of learning, and the use of insights due to Kohonen, insights which do have some relation to some of the earlier work of Grossberg.

When I talk about Kohonen, please bear in mind that I am not talking about the rapid convergence of Kohonen's learning rules, in their simplest approximate form. Rather, I am talking about the validity of the results, the statistical accuracy of the weights which these methods are converging to. The issues of convergence speed and accuracy are often mixed up in this area, and it is very important to test such ideas against appropriate criteria, in a diversity of situations.

CONVERGENCE STRATEGIES:

— BATCH LEARNING

— PATTERN LEARNING

— REAL-TIME LEARNING

Slide 11:

The next slide relates to the issue of convergence strategy and convergence speed.

There are three general ways that people have tried to use the derivatives that come out of backpropagation, in order to adapt networks.

The first way is what Jeff Hinton has called "batch learning." In batch learning, you go through all the patterns in your training set, and calculate the total derivatives of error across all the patterns with respect to the weights. You then adjust the weights, and go back through all of the patterns all over again before you adjust the weights a second time. And so on. Batch learning is exactly what statisticians have been using for decades, and there are very sophisticated methods for doing it. Furthermore, when Shastri and Watrous and I have adapted recurrent networks, we have generally used batch learning instead of the other alternatives, because of its greater speed in our applications.

The second alternative is something I like to call pattern learning. In pattern learning, you go through the patterns in your training set one by one. You calculate the derivatives of error in matching each pattern, and then adjust the weights immediately before moving on to the next pattern. You cycle through all of the patterns in the training set, over and over, in whatever order you like, until you converge.

In principle, it seems as if pattern learning really ought to be faster than batch learning. When steepest descent is used -- as in Rumelhart's formulation -- it usually is. When there is low information density in your training set -- such that a pattern could be thrown out without changing the results a whole lot -- then pattern learning will sometimes do better. But with high information density, especially, the use of batch learning lets you use sophisticated

numerical methods which perform far better than today's forms of pattern learning. Personally, I like to believe that this situation will change, after we do more research, and learn how to apply the sophisticated concepts of numerical analysis to the situation of pattern learning.

Real-time learning is yet another alternative. Real-time learning is just like pattern learning, except that you only get to see each pattern one time, and you always have to cycle through in the forwards time-direction. Almost no one is using real-time learning with backpropagation, and that is a measure of how far we have to go before we can fully understand organic intelligence.

BATCH LEARNING TOOLS:

- NONSQUARE ERROR, TARGETS
- "GENETICS", "METABOLISM"
- WEIGHTING SCHEMES
- SHANNO CONJUGATE GRAD
- FULL NEWTON
- ATTACHED MEMORY

Slide 12:

The next slide mentions some of the methods I have used or written about, in trying to speed up the convergence of systems I have worked with, usually in batch mode. It is certainly not an exhaustive list, but other people will be talking more about this subject this morning in any case.

On the whole, I have not devoted such a large share of my time to convergence as many of the people in this room have, simply because the problem has never affected me as severely as it has in some of the other work reported in this literature. There are at least two major reasons for this. First of all, I have

worked a lot with systems that are not based on the sigmoid function. The sigmoid function has unique properties, related to concavity and to saturation, which do not occur so often with other functions. Second of all, my colleagues and I have often found ways to apply different learning rates to different weights or parameters, on an ad hoc basis, which allowed very rapid convergence even as the inputs to the systems changed dramatically. For example, the Department of Energy used to use a long-range model, which was not based on backpropagation but faced essentially the same kind of convergence problem; even though it had about a hundred thousand highly nonlinear equations to satisfy, with major changes in input assumptions, it routinely converged in about twenty iterations, using this approach. I have also used this approach, with success, with a natural gas model based on backpropagation. Beyond this, I have also used some of the standard methods from numerical analysis on occasion.

This experience is reflected in two of the items on my list. First of all, by moving away from square error as measured for the output of the neurons, one can move away from a lot of the sigmoid saturation problem. After all, the use of square error tends to assume normally distributed noise, and normally distributed noise is a logical impossibility for variables that go between zero and one. Using a consistent noise model, one can improve the situation here. Perhaps the idea of target levels for the activation levels of cells might be extended even further, as a different kind of approximation to Newton's method.

Second of all, the list does mention the idea of using different learning rates for different weights, in effect. Unfortunately, no one has really demonstrated the right way to automate what my colleagues and I have done here, so there is lots of room for further research, as discussed in my paper.

Some people have even talked about "genetic algorithms" as a possibility here. Certainly they may be relevant to making and breaking connections and experiments within large, sparse networks, which I have written about in the past.

Going further down the list, there are two more methods -- Shanno's conjugate gradient method (which is not the same as BFGS) and the full Newton's method -- which come directly from numerical analysis. Shanno's method operates in O(n) storage and O(n) cost per iteration, exactly like steepest descent, but it's a whole lot faster; unlike most other conjugate gradient methods, it doesn't require perfect line searches, which can be a real problem when working with neural networks.

Contrary to popular opinion, Newton's method itself -- and not just the approximation due to Parker -- can also be implemented in O(n) storage, using tricks described in my paper. Whatever the ultimate value of this method, it should be useful in helping us to understand these convergence problems a little better.

And finally, in order to approach the real-time convergence abilities of the human brain, we may need to use a secondary kind of associative memory, to give our systems something like the ability to relive past experience even after the original input vectors are lost. Bear in mind here that the requirements for such a true memory system are very different from those of a supervised learning system used to predict new situations.

PICK WEIGHTS W_{ij} TO MAXIMIZE:

$$\Pr(\underline{W}/DATA) = \frac{\Pr(DATA/\underline{W})\ \Pr(\underline{W})}{\Pr(DATA)}$$

$$\text{e.g. } \Pr(\underline{W}) = N(0, w)$$

Slide 13:

The next slide, believe it or not, has to do with Kohonen's insights into supervised learning.

The supporters of backpropagation have often argued that minimizing square error -- however long it takes to converge -- will always converge to the right thing, according to statistical theory. After all, the least squares solution is the same as the maximum likelihood estimate, if we assume a normal distribution for the noise in the system. Leaving aside the issue of normality, this argument would be right, if maximum likelihood were always the right way to go.

On this slide, I am showing you the basic formula which underlies the theory of maximum likelihood estimation. In maximum likelihood estimation, we are trying to find the set of weights which has the highest probability of being right, or true, after a certain amount of experience. In other words, we are trying to maximize the probability of the weights conditional upon that experience as represented by the data in the training set. By Bayes' Law -- a basic theorem in probability theory -- this conditional probability equals the ratio on the right-hand side of the equation.

The denominator of this ratio really doesn't matter, in comparing different sets of weights, because it's not a function of the weights. The first term in the numerator is called the likelihood term -- the probability that we would have observed what we did if these were the true weights. The other term is the prior probability term, the probability that a set of weights might be true before we have any empirical knowledge available to us. If we assume that every possible set of weights is equally likely apriori, then we need only maximize the likelihood term, and that's what leads us to minimize square error.

However, economists and statisticians have discovered that this approach can break

down very badly in practice, especially when the number of patterns is less than the number of inputs. In fact, philosophers have argued that no kind of learning would be possible at all in the real world if it weren't for Occam's Razor -- a fundamental principle which depends on the idea that some sets of weights and structures are more likely to work than others, apriori. In the simple problems studied by statisticians, it has been discovered that a simple normal distribution can be assumed for the weights instead of equal probabilities, with good results. This approach, called "ridge regression," has been tested over and over again in practical applications, and found to work quite well.

PRIOR PROBABILITIES:

$$\text{RIDGE: } E' = E + k \sum b_i^2$$

$$\Delta b_i = \frac{\partial E}{\partial b_i} - c b_i$$

- EQUIVALENT TO KOHONEN
- ALTERNATIVES CITED

Slide 14:

This slide says just a little more about ridge regression. Ridge regression, when generalized to neural networks, yields the modified error function shown on this slide, which in turn leads to the well-known "decay term" often used with backpropagation. But bear in mind that we are looking at an accuracy issue here, not a convergence issue. When the parameter "k" goes to infinity here, the result we are converging to will get closer and closer to Kohonen's estimate, to within a scalar factor, assuming the use of Kohonen's simplest method (the approximate pseudo-inverse method). For many problems, the greatest level of accuracy will come for values of k somewhere between zero and infinity. My paper talks just a little bit about the empirical results I have had with this approach, and ideas for exploiting this situation to get faster convergence. Those ideas, in turn, are probably just the tip of a large iceberg, accounting for more advanced forms of associative memory.

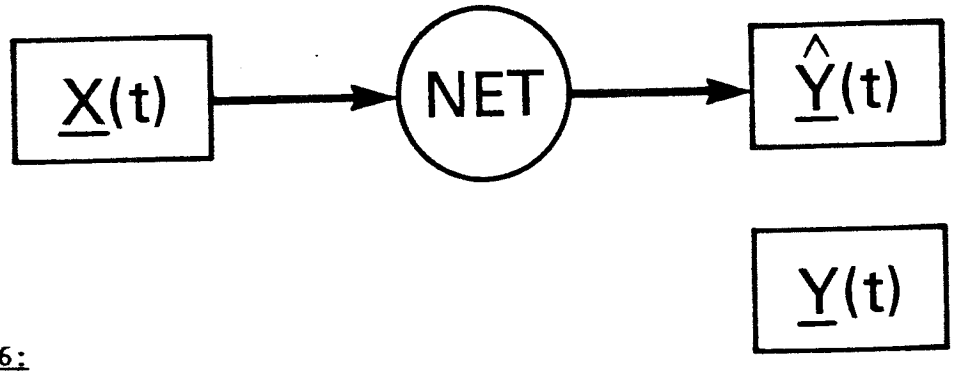
NETS FOR DYNAMIC MODELLING:

- SMC PROC THEORY, ICNN DETAILS
- SUPERVISED LEARNING, 1&2 NETS
- UNSUPERVISED, 1, 2 AND 3 NETS

Slide 15:

And now, finally, I will move on to the area where I've done the most work -- the development of networks for use in causal modelling or forecasting. Everything on the last 4 slides could still be used here but, for the sake of simplicity, I will go back to assuming a conventional least-squares version of backpropagation. I will display a series of five possible network architectures -- two of them supervised and three of them unsupervised -- building up to a "3-net" architecture which I would recommend as the canonical design for coping with these problems. My paper today says a lot about how to implement these architectures in a basic kind of way, but to understand the full range of theory and testing behind them, I would recommend that you go back to my SMC Proceedings paper from last year, cited in the footnotes.

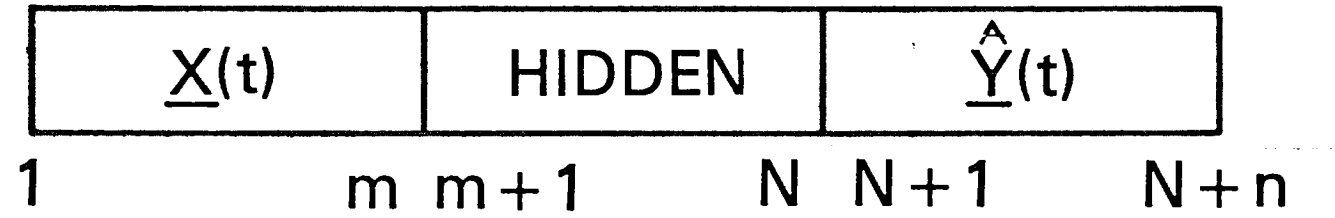
1-NET SUPERVISED LEARNING



Slide 16:

This slide illustrates the conventional, single-net architecture described by Rumelhart and all for supervised learning. The network inputs a vector "x", and outputs a vector "y-hat", which is supposed to approximate the target vector "y".

ARRANGEMENT OF NET$_i$ (t) NEURONS IN 1-NET

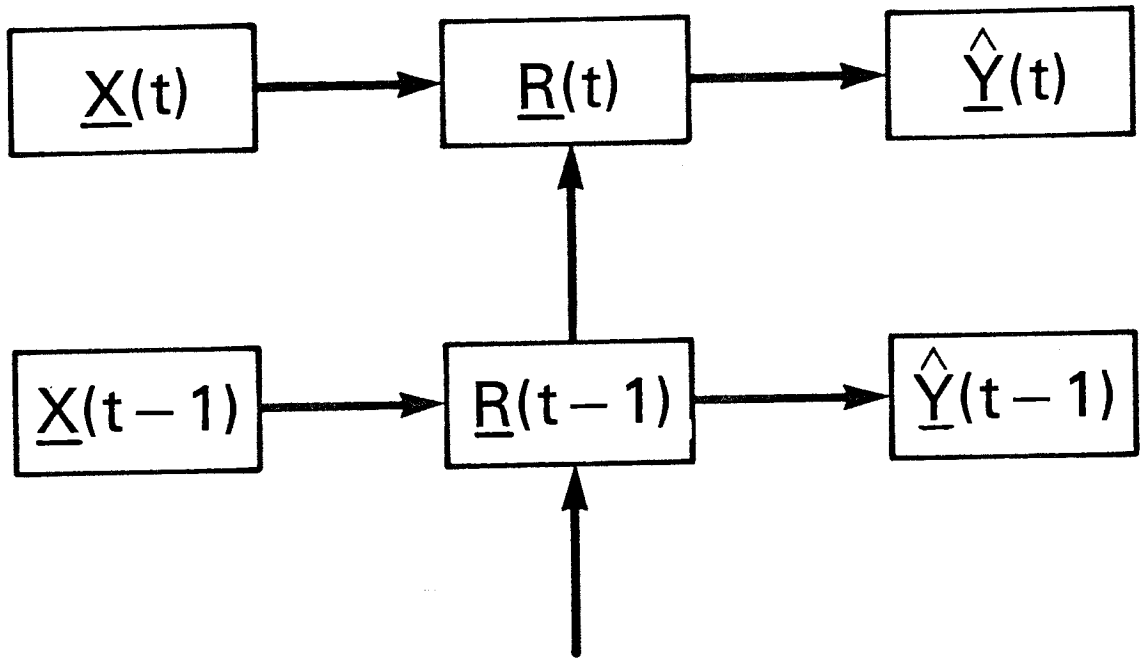


Slide 17:

The next slide illustrates how you can actually program that kind of network in practice, on a sequential computer. (The last time I wrote this up, I assumed a parallel computer, and the notation drove everyone up the wall.) You can define an array, "net," which corresponds to the activation levels of all the cells. The first m cells correspond to inputs, and the last little-n cells correspond to output cells. Everything in-between is a hidden cell. To calculate the activation levels, you simply set up a DO loop from cell number m+1 -- the first hidden cell -- through to cell number N+n -- the last output cell.

To calculate all the derivatives, you first calculate all the derivatives of error for all of the output cells. You then work your way backwards, from cell number capital-N -- the last hidden cell -- through to the far left and the weights. And Then you adjust the weights. You could use cell outputs instead of activation levels in your program, or a combination of the two, but the same general approach applies.

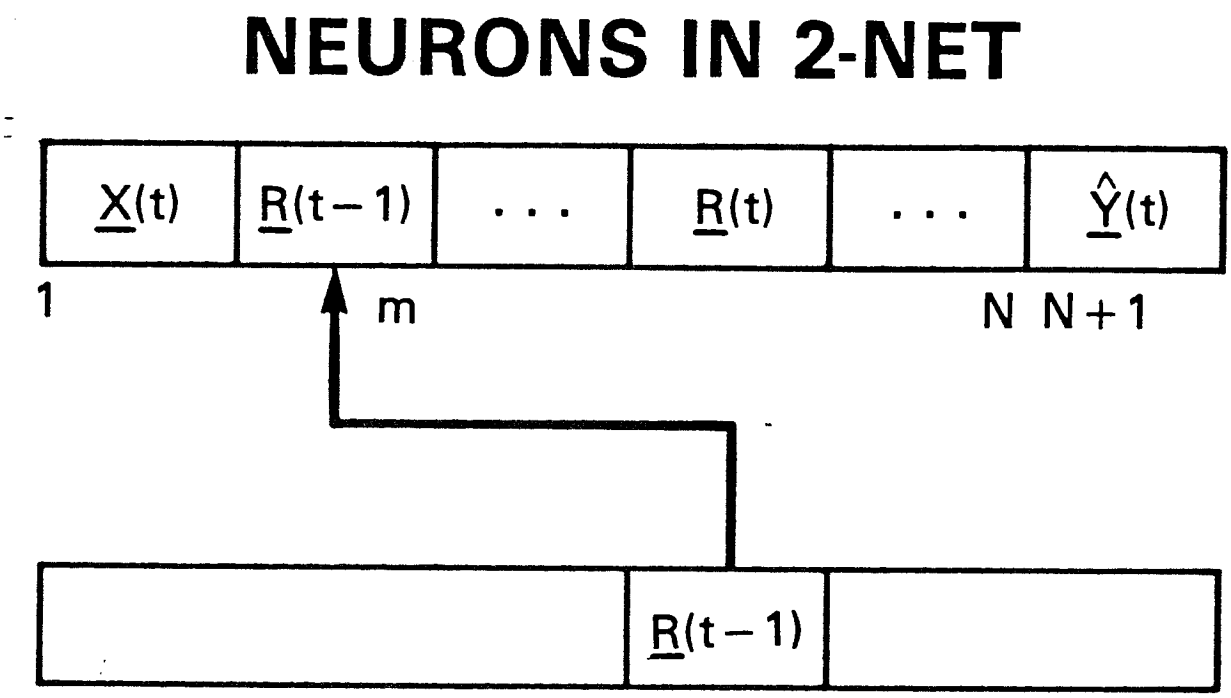
2-NET SUPERVISED LEARNING



Slide 18:

This slide shows a two-net architecture for supervised learning. This is a very slight generalization of the architecture used by Shastri and Watrous with so much success in the area of speech recognition. The vector "R" -- which I think of as "Reality" or "Recurrent" -- provides a kind of short-term memory of the dynamic system under study.

Just yesterday, Morris Hirsch told us that simple backpropagation networks could not possibly represent what goes on in the human brain, because the human brain has interesting dynamics while those networks do not. His comment is a valid criticism of 1-Net supervised learning networks. But here, the addition of R -- and the other features I will get to -- does lead to very interesting dynamics.



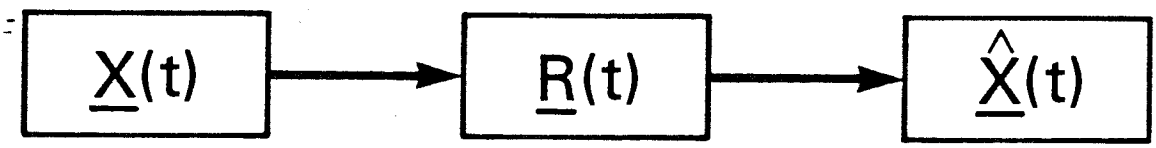
Slide 19:

The next slide shows how to implement that kind of architecture, using backpropagation. Again, we define a single array, called "net," which includes all of the cells. And again, we can use a DO loop to calculate the activation levels of all the cells from m+1 through to N+n. In this case, however, when we are done calculating all the activation levels for a given time, we must next copy over some of these activation levels into the input block for the next time period. We have to proceed forwards in time, from the beginning of a time sequence to the end, to calculate all of the activation levels.

Then, in order to calculate the derivatives, we have to work our way backwards in time, from the last pattern to the first. With the last pattern, we calculate the errors and the derivatives exactly as we would in simple backpropagation. But then, we must remember the derivatives of error with respect to R(T-1), as it appears in the input block for time T; when we go on to calculate the derivatives for that previous time period, we have to add in these derivatives to the derivatives of error with respect to R(T-1), at the point where R(T-1) is calculated. The formulas are given in my paper, but they are trivial to work out in any case from the chain rule for ordered derivatives.

This kind of architecture is more demanding than the 1-net architecture, but the importance of having a memory and the success already reported by Shastri and Watrous should make it clear that it is worth the trouble.

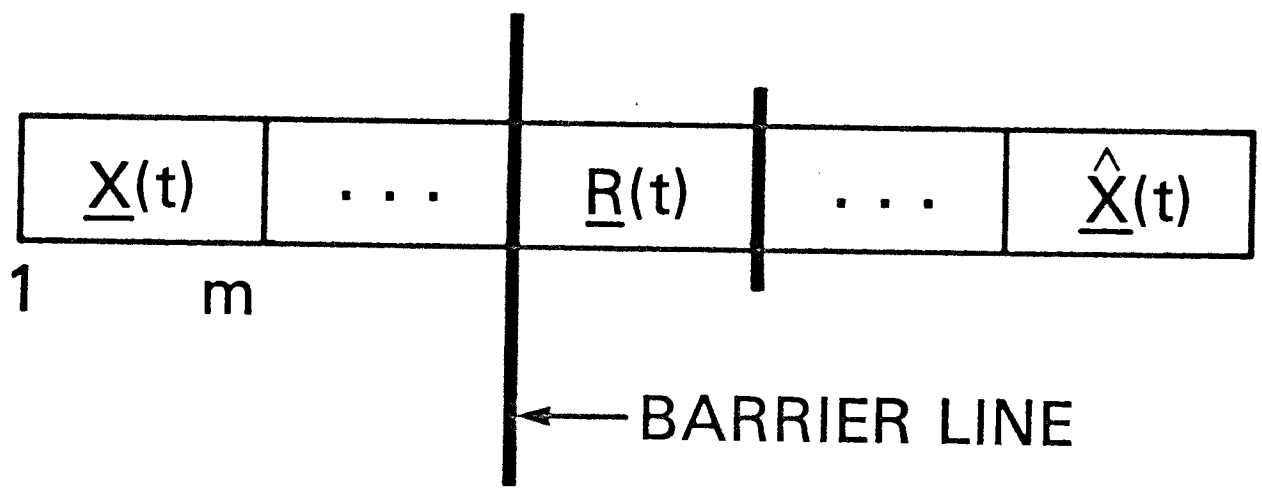
"1-NET" ENCODER/DECODER



Slide 20:

This slide illustrates a simple kind of unsupervised learning network, using backpropagation, which has been used by many people in the past. I'm not sure who first used this architecture, but I suspect it was Hinton. The idea here is to compress the original vector, x, into a smaller vector R, so as to minimize the decoding error when we go back again to x. Notice that this will only work if we force the vector R to be smaller than x, and insist that the decoding network not have access to the original vector x.

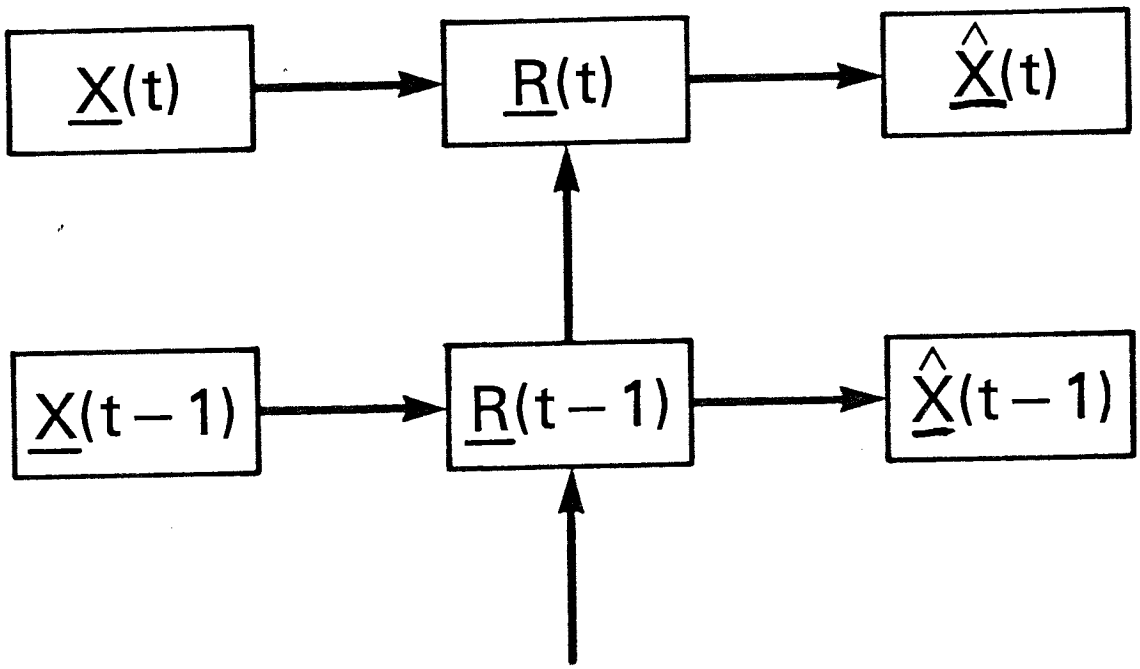
NEURONS IN ENCODER/DECODER



Slide 21:

This slide shows how to implement the simple encoder/decoder, with a very slight generalization. Again, note the importance of not allowing the later neurons to have access to any inputs from the left of the barrier line.

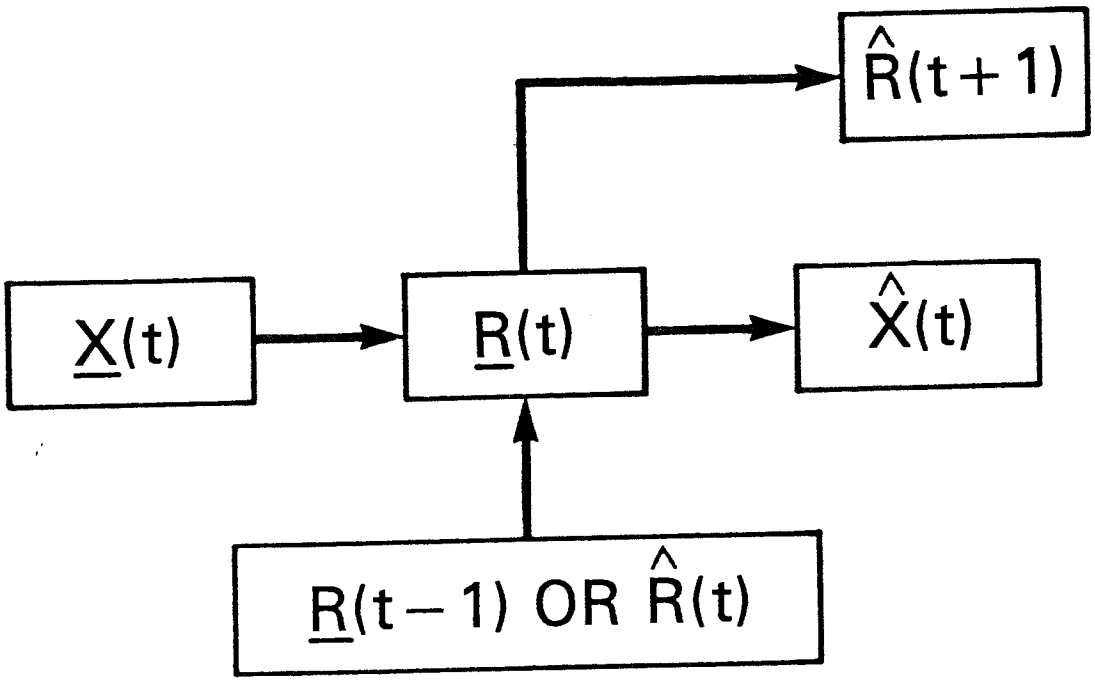
2-NET UNSUPERVISED LEARNING



Slide 22:

This slide illustrates a generalization of the simple encoder-decoder idea to a dynamic, 2-Net structure. Again, it is necessary that R be smaller than x. The implementation of this network would be nearly identical to that of the 2-Net supervised architecture. In practice, the addition of memory here may not be as useful as it was with supervised learning; however, the next slide will fix that problem.

MINIMAL EXAMPLE OF 3-NET



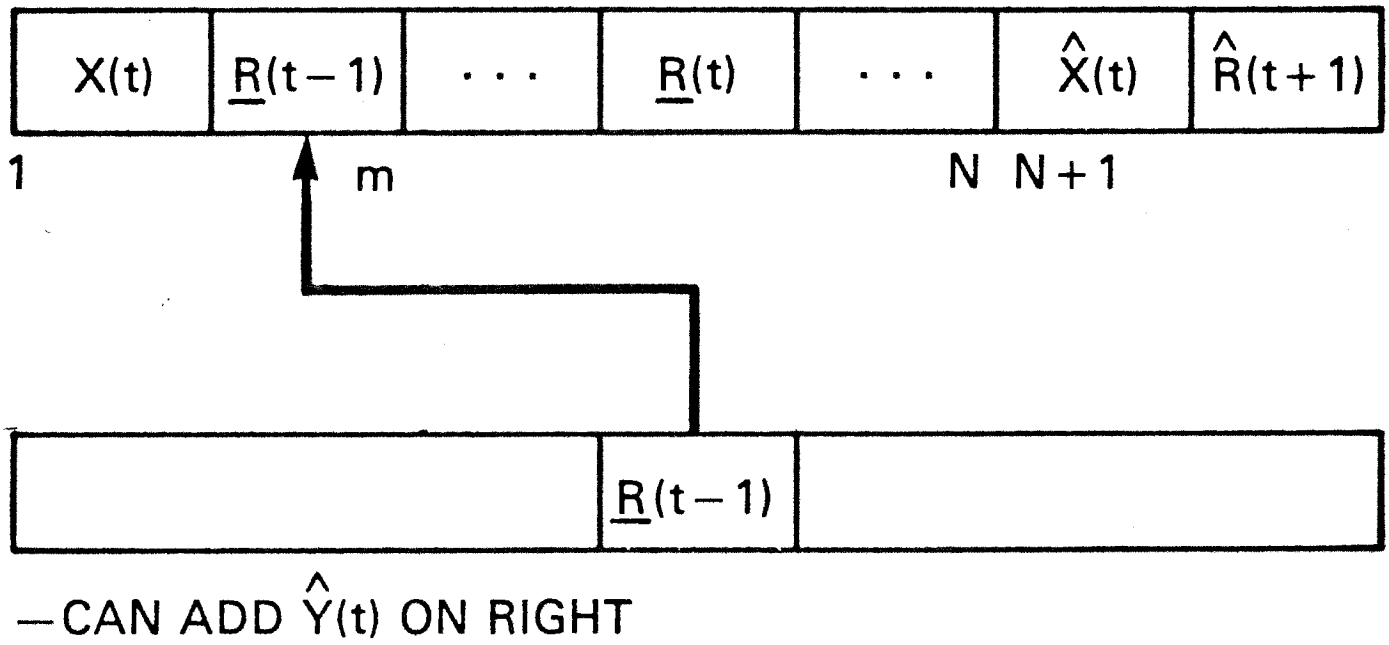
Slide 23:

And finally, this slide illustrates a basic form of the 3-Net architecture, which in my view is the only plausible picture we now have for the kind of modelling going on in mammalian brains. Note that this architecture lets you forecast both x and R over time, so that it gives you the full-blown kind of causal model that you would need for reinforcement learning over time.

Also note that R no longer has to be smaller than x. This is important, since x represents raw inputs, like what we get from the retina, and R represents the kind of thing we would expect to find in the cerebral cortex. It would be hard to believe a model which requires the cerebral cortex to be smaller than the retina.

Finally, the 3-Net architecture -- unlike the others -- is capable of representing a generalized stochastic model, with error correlations of all kinds between different observed variables. These kinds of advantages are discussed further in my SMC Proceedings paper.

NEURONS IN 3-NET EXAMPLE



Slide 24:

The next slide illustrates the implementation of the 3-Net architecture, which is not much harder at this level than the implementation of the 2-Net supervised architecture. Some connections have to be disallowed, as with the other unsupervised architectures, but this is no problem. Note, however, that it could get tricky if you also had to weight the errors for different target variables with different weights. This is a subject discussed in my SMC Proceedings paper which requires further research. As with the encoder/decoder structure, it is possible to begin with a smaller R vector, and then add extra cells in an incremental way. Also, for certain practical applications, like speech recognition, one can add a y-hat output to the far right, so that a single network of this type can do both feature extraction and word labelling at the same time.

$$R_i(t) = (1 - w_i)\hat{x}_i(t) + w_i x_i(t)$$

$$\underline{\hat{x}}(t) = \underline{f}(\underline{R}(t-1))$$

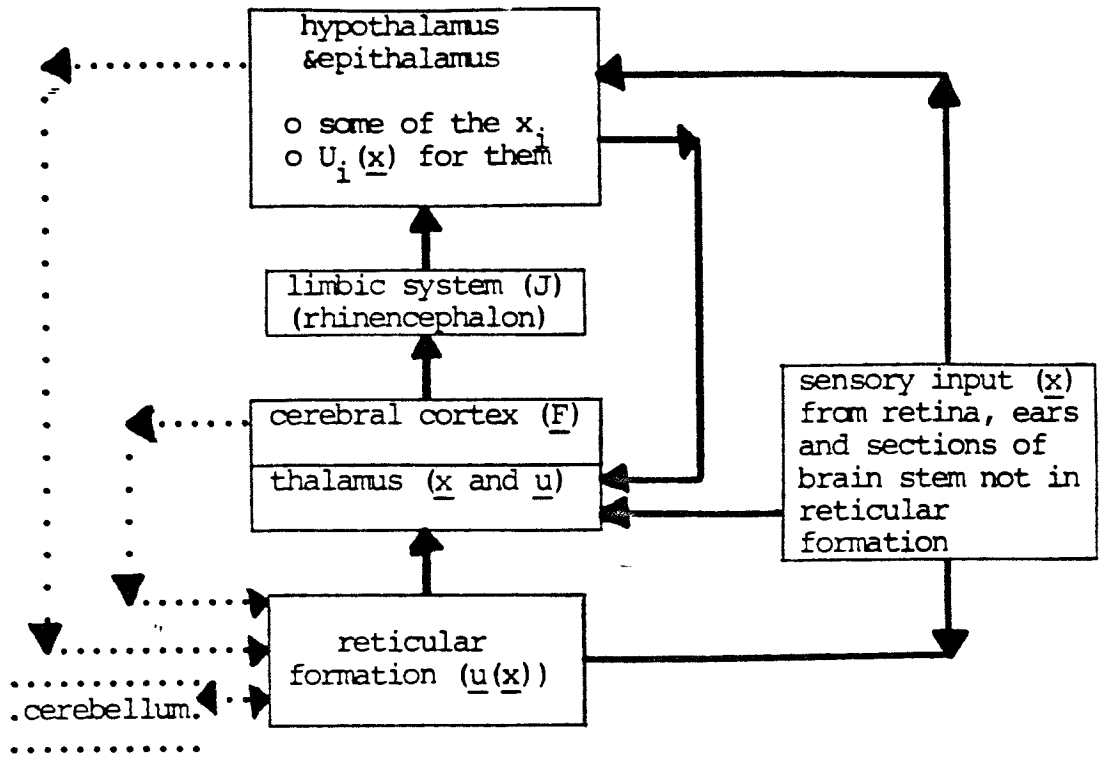
$$\text{MINIMIZE } E(w_i, \Sigma(x_i - \hat{x}_i)^2)$$

A ROBUST TIME-SERIES METHOD
(SMC 9/78; DOE/EIA-0420/3)

Slide 25:

The next slide quotes a few equations from my SMC Proceedings paper, equations which describe a family of structures which I tested out on time-series data from social science. This kind of structure -- which can be represented within the 3-Net architecture -- led to much smaller prediction errors than conventional time-series methods, in a number of experiments, for technical reasons to be discussed in my October paper in Neural Networks. The intuition here is similar to that of Sutton's ideas for time-stepped forecasting, but the method used is quite different.

Comparison with the Human Brain



Slide 26:

This is my final slide, and it's really just a place-holder for a subject I've looked at in great detail, which there's no time for here this morning. This slide summarizes some of my efforts to go back from these complex, mathematical designs to what we actually observe in the human brain. Surprisingly enough, there are some very close fits between the kinds of architecture I have talked about and some of the gross, structural features and behavior of the brain which micro-level theories tend to neglect. The details relating to this chart are given in my SMC Transactions paper from last year, while details relating to the 3-Net architecture are touched on in my SMC Proceedings paper.

Before I close, let me talk about one more point on these lines which has come to be a great source of confusion in this field, just as the McCulloch-Pitts model was in the 1960's.

A conventional wisdom has grown up which says that Hebbian learning is realistic, from a biological point of view, while backpropagation is not. In actuality, I just received some papers from Leon Cooper, a Noble Prioze winner at the Brown University Center for Neurosciences, who points out that Hebbian learning and backpropagation both require backwards flows of information. For example, if we assume that the giant pyramid cells in the cerebral cortex are capable of adaptation, then both theories would require a flow of information going all the

way back from the axon hillocks of these cells through to the tippy-top of their apical dendrites. No such flows are known to exist, and the very idea of such a flow tends to contradict what is known about the flow of electricity in the memranes of nerve cells. If you look at all the complex models of membranes which have been developed in the last few years, you will see that none of them implies such a flow.

On the other hand, nerve cells are not just membranes. If you want to study the forwards, learned, electrical properties of neural networks, then membranes are mostly all you have to know about. But plasticity or learning is something quite different. The hard-core neuroscientists will tell you that the cytoskeletons of neurons are almost certainly crucial to learning, even though we don't know all the details as yet. If the cytoskeleton of an amoeba can carry information rapidly, in electromechanical form, inside of a cell, then why can't the cytoskeleton of a human nerve cell, if such information is crucial from a computational point of view?

Another live possibility for carrying such information is the system of glial cells in the brain. For example, the axons of the giant pyramid cells are lined with a system of coupled glial cells (myelin sheath) which might be capable of carrying information backwards. Backwards flows of information along axons are required by backpropagation, but do not occur in Hebbian learning. In any event, the discussion above suggests that there may be a mix of different kinds of cells in the brain, some of them adapted by backpropagation, some by Hebbian learning, and some by a kind of combination.