## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

**Order Number 9405598**

# Neural programming, computational scheme and applications

Weiss, Jacob, Ph.D.

City University of New York, 1993

**Copyright ©1993 by Weiss, Jacob. All rights reserved.**

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

# Neural Programming,
## Computational Scheme and Applications

by

**Jacob Weiss**

A dissertation submitted to the Graduate Faculty in Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy, The City University of New York

1993

© 1993

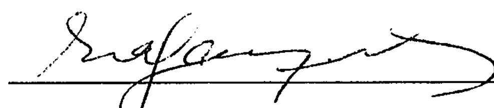
Jacob Weiss

All Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

May 26, 1993

Date



Professor Eralp A. Akkoyunlu

Chair of Examining Comittee

May 26, 1993

Date

Professor Stanley Habib

Executive Officer

Professor Frank Beckman

Doctor Neil Gerr

Professor Kenneth McAloon

Professor Stathis Zachos

Supervisory Commitee

THE CITY UNIVERSITY OF NEW YORK

# Abstract

## Neural Programming, Computational Scheme and Applications

by

**Jacob Weiss**

Adviser: **Professor Eralp Akkoyunlu**

An attempt is made to solve difficult AI problems by introducing a new scheme, the net program. The net program is a programmable network of cooperative neural nets and procedural functions. Its goal is to bridge the gap between connectionist-subsymbolic and symbolic processing in AI. A set of subroutines has been devised for "neural programming" as an enhancement to the procedural language C. The resulting language, which we call $C^{+NET}$, enables and controls modular structuring, training and activation of nets and the interaction among nets and procedural functions. We applied $C^{+NET}$ on a model for natural language comprehension as a cardinal unsolved AI problem of an intricacy that would strain any conceivable computational scheme and on a model for pattern recognition as a demonstrative example of the advantages of the approach we are presenting. We also present a design of an application to financial forecasting utilizing our net program scheme.

## Acknowledgments

My deep thanks to the people who encouraged and helped me throughout this work; Prof. Eralp Akkoyunlu, Prof. Frank Beckman, Prof. Kenneth McAloon, Prof. Zachos, Prof. Habib, Prof. Anshel and Drs. Hsu & Gerr, who showed me the way in various junctions of my academic venture, to the lifeguards and dwellers of the beaches of Rio where I started this work, and Tel Aviv where I finished it, who made sure it could go on, and to my parents to whom this is dedicated.

# Contents

1 Introduction and Motivation....1
1.1 About the Appeal of Neural Nets....2
1.2 Some Limitations of Neural Nets....3
2. An Introduction to Net Programs....8
3. The Net Program Scheme....9
3.1 Overview....9
3.2 Details....10
3.2.1 Neural Net Nodes of the Net Program....10
3.2.2 Connecting Neural Net Nodes....12
3.2.3 Activation of the Net - First Visit....14
3.2.3.1 Feedforward....14
3.2.3.2 Training....16
3.2.4 Functions....17
3.2.5 Activation of the Net - Second Visit....19
3.2.6 Short Term Memory....20
3.2.7 Activation of the Net - Final Visit....20
4. The Net Program Language - C+NET....21
5. A Natural Language Comprehension Model Based Upon and Implemented by the Net Program Scheme....27
5.1 Introduction....27
5.2 The Input....31
5.3 The Language Comprehension Net....38
5.3.1 Subnets, Connections and Usage....39
5.4 The Parser....51
5.5 A Rather Long Example of Parsing a Sentence....61
6. An Example of a C+NET Procedure in the Parser....70
7. An Outline of a Model for Financial (or Other) Forecasting....74
8. A Net Program for Pattern Recognition....82
8.1 Overview....82
8.2 Details of an Application to Digit Recognition....84
9. Conclusion....97
10. Future Research....97
Appendix - Some C Code of C+NET....99
References....136

# 1. Introduction and Motivation

The goal of this work is to present and demonstrate a fusion of classical programming and neural nets into a scheme capable of solving hard AI problems.

The scheme we are presenting, and the programming language implementing it which we are providing is intended to make use of the best of both procedural programming and connectionist methods, yielding benefits not present in either alone. It equips neural nets with capacities that they are now missing: logic through deductive reasoning, symbolic processing, reuse of skills, efficiency, modularity, accessibility and communicability. It inherits from neural nets the advantages that they can provide and enables their usage within the rest of the programming world. We intend to help turn neural nets from the magic boxes as which they are often treated into problem solving tools. The way we chose to do it is by treating neural nets as software tools embedded in a procedural environment. To achieve this, we modularized the structure of nets as well as their activation. By the presented methodology, nets and subnets can be created, trained, connected and accessed as modules. They can interact with each other and with conventional programs. They can be controlled by programs and interact with logical modules. They can be called to do sub tasks within a broader algorithm, or they can use a procedural algorithm to do a sub task for them. They can be used recursively. They can be reused. or duplicated. All these operations are handled by a programming language which extends ANSI C to include a set of tools for neural programming.

We first discuss the motivation for the new scheme, then present the scheme, which we call "net program", and its implementation, a programming language which we call $C^{+NET}$. We proceed to the application of the scheme on a key AI problem - natural language comprehension, on a financial forecasting model and on a pattern recognition solution.

## 1.1 About the Appeal of Neural Nets

Neural nets can do some very nice things that are difficult to achieve using other methods [1].

- They can generalize.
- They can approximate.
- They are versatile and can be used for many different problems.
- They can deal with imprecisely defined problems.
- They can solve problems for which there is no good algorithm.
- They can be implemented to function in a massively parallel way.

All of these are much sought after in the field of AI, and have made neural nets a very promising method for solving AI problems. Furthermore, neural nets are attractive for solving AI problems because from a cognitive point of view, they may have an advantage in being formulated after the biological computation mechanism [2, 3, 4, 5].

However, neural nets have not revolutionized the field of AI to the extent of retiring other methods, and they seem to still be a promising tool rather than a fully mature one. Substantial research has been done along the lines of improving their 'internals' by studying their mathematical support and in applying neural nets to a multitude of problems. This research has been improving the performance and applicability of neural nets nicely. Yet, most current applications look similar - a preprocessor and a rather basic decision net. It is not rare for the preprocessor to have the major role in the application and define the way it works, leaving little for the net to do. We believe that this, almost paradigmatic, usage of nets is of limited capacity. It does not exploit the full potential of neural nets and it does little to address the important limitations that neural nets do have.

## 1.2 Some Limitations of Neural Nets

- Neural nets need sufficient empirical evidence to solve a problem. This may mean a lot of sample data and many variables.

- Sufficient sample data is often not available, and even if it is, processing it may require unreasonable training time. Accommodating more than a few variables results in a need for unreasonable computer resources. Combinatorial explosion [6] can be reached quite easily.

- Neural nets do not use any knowledge other than what they directly induce from the data, unless they are structured and supervised.

- They are inefficient unless they are structured and supervised.

- They are difficult to structure and supervise. Their learning capability may be restricted by hardwiring and they would then lose their versatility and adaptability.

- They do not use any procedural knowledge, rules or instructions, which may be critical for solving a problem. In fact, it has been argued [6] that they perform no more than "hill climbing".

- They are not communicative - it is difficult to tell them what to do and to know what it is that they are doing.

- The solutions they find are statistical and their veracity is not demonstrated, nor provable.

- They cannot accept problems nor produce solutions in a symbolic form. Furthermore, they may require an impractical encoding scheme from symbols (e.g. in a large domain or a continuous one) into input neuron activation.

- They cannot reuse their acquired skills for different tasks.

One of the most restrictive characteristics of neural nets is also one of their strongest, and that is their relying on inductive reasoning. Inductive reasoning is very important in AI, but we believe it is insufficient or at least impractical and, more importantly, surmountable for complicated tasks.

Apparently, "natural intelligence" uses both inductive and deductive reasoning, as humans can do both. Moreover, it uses them in a complementary manner. Humans use experience, intuition and logical reasoning to solve problems and, introspectively, we can rarely tell exactly which one of them we use to reach a solution of a problem because we use them cooperatively. We need and use symbolic processing for at least the higher functions of natural intelligence. This may be a strong claim for a philosopher to post [7, 8, 9 and much cited 28], but a natural and almost obvious working assumption for a computer scientist.

In [28] Fodor and Pylyshyn maintain that minds are systematic. That is to say that there are structural relations between related states of mind. Generating syntactically correct sentences, without having the list of all possible sentences, requires applying combinatorial rules upon syntactic structures. This starts at the very basic level of combining words. **Brown** and **cow**, put together, produce **brown cow**, a syntactic structure which relates to both its semantic constituents. Similarly, "John and Mary came in" has logical relations to "John came in" and "Mary came in". Furthermore, anyone who can say in any language "John loves Mary" would not need any learning to be able to say "Mary loves John", which can be formed by a structural operation on "John loves Mary". But it's not just language, its the way mind works. Logical inferences, too, rely on structure. In any mental life we would expect that if $P\&Q\&R \Rightarrow P$ is valid then also $P\&Q \Rightarrow P$ would be valid. The understanding of relations is connected to structure. In any organism we would expect that a sensitivity to $aRb$ would imply sensitivity to $bRa$.

The "Orthodox Connectionist" approach does not facilitate any of the above. Fodor and Pylyshyn say that "The only system that is known to be able to produce pervasive systematicity is Classical architecture", where by Classical architecture they mean traditional Turing-machine-like symbolic processing. They go on to say that "Classical architecture is not compatible with Connectionism since it requires internally structured representations."

David Hume, in "A Treatise of Human Nature", called the mind "**a bundle of impressions**". In that, he was an associationist and perhaps would have been a connectionist. Fodor and Pylyshyn say in a footnote to [28] that "Hume in fact cheated: he allowed himself not just association but also 'imagination', which he takes to be an 'active' faculty that can produce new concepts out of old parts by a process of analysis and recombination." Hume, in this matter, did not adhere to the rules of the game of strict associationism, but we do not consider this cheating. He took the natural step which we believe should be taken beyond connectionism: the consolidation of symbolic deductive processing into the connectionist (associationist) model. Indeed, representations can be created by connectionist nets, but they can be processed and restructured in the classical symbolic way.

We believe that natural intelligence uses symbolic processing, but on the other hand it seems rather obvious that it uses some kind of neural nets. Denying, or dismissing the role of neural nets in natural intelligence is a fine dialectical feat, and an all decisive proof pro or con is not at hand, but we believe it is more constructive to accept that neural nets do operate in the brain in a manner that bears at least some resemblance to the understanding that we have of them, the understanding that inspired connectionist models.

We believe that it would be a good strategy to solve AI problems with similar tools to those of natural intelligence, using both symbolic and connectionist

inference. Using serial, and even parallel, current electronic computers we have a disadvantage compared to biological brains in the inductive arena, but we (probably) have an advantage in the symbolic processing arena. This is, perhaps, the reason that some faculties, the likes of formal logic, that we consider higher, are easier for computers than some faculties that humans take for granted. Deductive and symbolic processing may in some senses be the most "intelligent" part of natural intelligence, and cannot be dismissed in artificial intelligence. They are probably essential for key subjects of the mind. We believe that better AI solutions can be reached by improving the usage of inductive tools, and by incorporating them with deductive tools. We believe that the inductive tool to use for AI problems is neural nets because, when *structured and* trained properly, they can emulate the sustenance of innate and inductive knowledge. These types of knowledge are probably behind most of the processes that take place in the brain. Contemporary interesting AI problems involve, by nature of being interesting, elements of both "higher" and "lower" faculties of the mind, innate, inductive and deductive forms of knowledge and reasoning methods.

Neural nets can provide some indispensable power for massively parallel tasks which abound in AI (vision, pattern matching), and for associative memories. When coupled with symbolic processing they can turn into semantic nets [29] and enable implementation of important related models of mind and, further, of expert systems operating with fuzzy logic.

Mind has to be modular [2, 9, 10, 11, 12]. The problems that it solves are too complicated and too intricate for a single backpropagation [1] type net to be able to solve. The modularity is necessary for a few reasons. One is practicality. It is to our opinion unthinkable to magnify a (now classical) backpropagation net to the scale in which it can solve problems that humans do, even those that do not

involve logical reasoning or symbolic processing. Secondly, symbolic and deductive processing cannot take place without an interaction among agents. Perfectly distributed processing means perfectly distributed representations, and this would render symbolic processing both meaningless and impossible. Furthermore [2], any large scale computation will probably be too fragile to be performed in a non modular manner. Lastly, there is, by now, ample empirical data about the structure of the brain, its connectivity and patterns of arousal of neurons in it [11, 12, 13, 14, 15] to support, if not prove, the assumption that the brain is a composition of various networks of different function and expertise. From a philosophical point of view, this may not necessitate a modular structure of *mind,* as opposed to *brain,* and of a modular structure for a computational model of mind, but for a computer scientist it is not a huge logical leap from brain structure to mind structure, it is, rather, a natural course to follow in order to achieve solutions to problems similar to those that the brain, or mind, solves. Authors from disciplines other than philosophy proceed smoothly from brain structure to models of performing its tasks [2, 30]. Certainly, it should be natural for connectionists to draw conclusions from the brain structure, in which case they should have to accept changes to the strict multi-layer neural net model.

*The conclusion we make is that a consolidation of inductive and deductive reasoning capacities and symbolic and connectionist processing methods can be a strong instrument for solving difficult AI problems. It can be constructed by devising a programming scheme to structure, aggregate and master neural nets and their interaction, and imbed them in procedural language computer programs.*

We think that such a scheme will be capable of solving real AI problems. The work underlying this thesis is an attempt to bring one about.

## 2. An Introduction to Net Programs

The key idea of our consolidation of neural nets and procedural programs is to view simulated neural nets as programmable software modules to which programming methods are applied. This opens the way to a marriage of nets and programs, yielding the hybrid we call a net program. The goal is to endow the hybrid with the advantages of both neural nets and programs, as follows:

- The hybrid would use both procedural and connectionist means to solve a problem.

- Its input, output and, where needed, internal representations could carry symbolic meaning. It could process symbols.

- It could accept descriptions of problems and solutions as they are known to the programmer - using symbols, algorithms, rules and prior and axiomatic knowledge as well as evidence from sample data. This is the way humans reason, and this is what an AI problem solver should be able to do.

- It could permit a variable level of control of structure and learning. It could blur the distinction between procedural knowledge and the knowledge acquired from the data, by having structure and procedural assertions help reach a solution without dictating it, much like human reasoning. It could use strictly computer-like processes where accuracy is needed. The way it functions will be understandable to and manageable by its users.

- It could be structured, created, tested, refined and used like a high-level language, compounding modules by structure and functionality, distributing various tasks to various modules, and reusing and re-invoking modules. It could do so for both functions and neural nets.

- It could solve problems of meaningful scale and complexity.

# 3. The Net Program Scheme

## 3.1 Overview

The net program scheme was designed to achieve the goals stated above in section 2. Its basic components are procedural functions and neural nets. Each of the neural nets is (currently but not necessarily) a backpropagation net of one or more layers of one or more neurons. A net program is composed of these basic elements or of other net programs. There are no restrictions upon the connections among nets - nodes can be connected to any number of any other nodes, and cycles are permitted.

Neural net nodes and function nodes can be connected by way of invocation - a net node can invoke a function, a function can use a net to do a sub-task. Invocations are recursive - nets and functions can invoke and re-invoke each other.

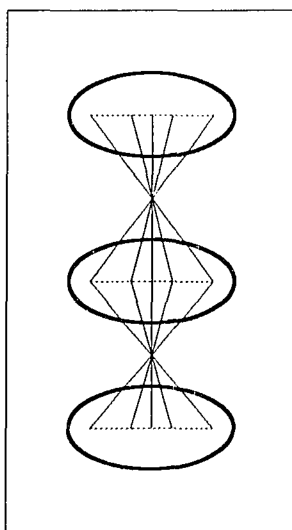
Neural nets can be connected to each other by arrays of connection weights. Connected nets, therefore, need not be of matching input and output layer size. The backpropagation mechanism is extended accordingly. The connection weights can be manipulated to set default correlation. Subnets can be used or trained separately or jointly. Connected nets may be pre-trained.

The net program is written in a high level language. The procedural statements within function nodes are regular programming language commands. Imbedded in them are statements that handle creation, input/output manipulation and training of neural net nodes. These statement activate various parts of an engine that drives the execution of the net program.

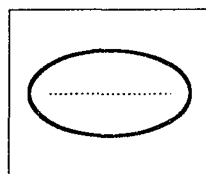
## 3.2 Details

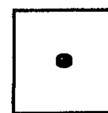
### 3.2.1 Neural Net Nodes of the Net Program

Neural net nodes of the net program can be multilayer nets,



or one layer nets,





or one neuron nets.

Each net node is a backpropagation neural net.

The nets being backpropagation nets is not essential to the concept; back propagation was chosen for convenience, and other similar algorithms, if proven better, would fit just as well. The one-layer and one neuron nets are meaningful within larger structures as described further on.

The backpropagation net is conventional. It follows [1] and [16]. There is an input layer, optional hidden layers of any number and an output layer. All neurons in each layer are connected to all neurons in the next layer by an array of weights. An extra weight is added for a threshold unit. The activation of neurons in layers other than the input layer is determined by a squashing function of the activation of the neurons in the previous layer, as follows:

$$O_{j,k} = \frac{1}{1+e^{-\sum_i W_{i,j,k} O_{i,k-1}}}$$

where *O(j,k)* is the output of neuron j in layer k, *W(i,j,k)* is the weight for the *i*th connection to it (the connection from the *i*th neuron in layer *k*-1). Neuron 0 is set to 1, to enable derivable thresholding [16].

The error is calculated for neuron *j* in the output layer *K* as:

$\delta_{j,K} = [P_j - O_{j,K}][O_{j,K}(1 - O_{j,K})]$, where $P_j$ is the output pattern.

The factor $O_{j,K}(1 - O_{j,K})$ is the derivative of the activation function above.

For neuron *j* in layer *k*, other than the output layer, the error is calculated as:

$$\delta_{j,k} = [\sum_i \delta_{i,k+1} W_{j,i,k+1}][O_{j,k}(1 - O_{j,k})]$$

The weight changes are propagated backward among layers by the formula:

$$W_{i,j,k} = W_{i,j,k} + \rho\delta_{j,k} O_{i,k-1} + \Theta Oldel_{i,j,k}$$

where $W_{i,j,k}$ is the weight from neuron *i* in layer *k-1* to neuron j in layer *k*,

$0 < \rho <= 1$ is a learning rate parameter,

$Oldel_{i,j,k}$ is the result of the expression $\rho\delta_{j,k} O_{i,k-1}$ for the previous cycle

and $\Theta$ is a smoothing factor.

The **learning rate** and **smoothing** factor are parameters of the net node. They are given, or set to default, when the node is created, and can be changed afterwards at any point of the execution of the net program.

The other learning parameters are:

**Tolerance** (0 to 1): - the maximum difference allowed between the desired and actual output of a neuron.

**Pattern Tolerance** (0 to 1): - what fraction of the number of output neurons must be within the above tolerance for each neuron.

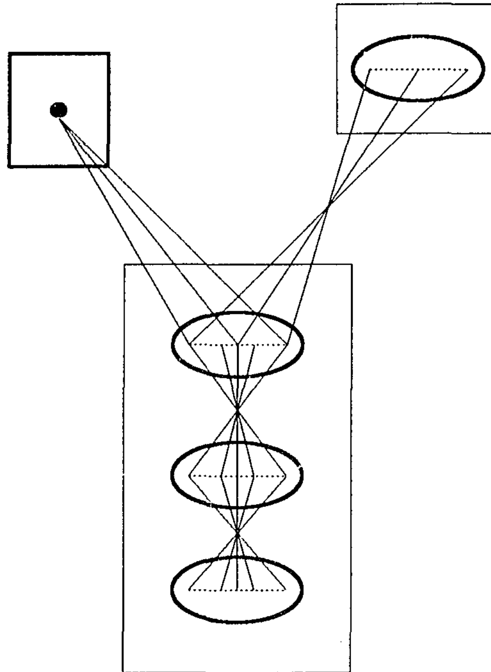
**Tenacity** (0 to 1): - A measure of a limit to the effort to be invested in reaching the desired accuracy of output. This has a stronger practical meaning than tolerance. When less than 1, it activates two control mechanisms:

1. Limiting the number of learning cycles.
2. Requiring an improvement in the proximity of the output to the desired patterns in a rate that corresponds to its value. If the learning in a node progresses slower than that, it is stopped.

**Feedprobability** (0 to 1): A probability for the node activation. Meaningful in the context of multiple node activation, and will be explained in that context.

### 3.2.2 Connecting Neural Net Nodes

Net nodes in the net program can be composed into larger structures. They can be connected to form a structured backpropagation net. The structured back propagation net is an extension of the layered one. The connection between two nodes is a connection between the output layer of one node and the input layer of the other, and this is an array of weights, as follows:



The feedforward formula for an input layer of a node that has other nodes connected to it now involves collecting the input from the output layers of the these nodes:

$$O_{j,k} = \frac{1}{1 + e^{-\sum_{c}\sum_{i} W_{i,j,k} O_{i,k-1}}}$$

where $c$ goes over the connections from active nodes.

The backpropagation formula for an output layer of a connected node would change similarly, to collect the errors from the nodes to which it is connected:

$$\delta_{j,k} = [\sum_{c}\sum_{i} \delta_{i,k+1} W_{j,i,k+1}][O_{j,k}(1 - O_{j,k})]$$

The actual formula is a bit more elaborate: As detailed in the context of subnet activation, the backpropagation of errors between nodes is optional. It can stop at the connection level to affect the connection weights only. This may be desirable when the input node is pre-trained, or is trained by other connections. In this case

we do not want to affect the weights of a node by some or any of its output nodes. A backpropagation probability (0 to 1) is assigned to each connection to control the collection of errors. Furthermore, a node can have its own output patterns to learn and be connected as input to other nodes simultaneously. The exact formula for calculating the error is therefore:

$$\delta_{j,k} = [\alpha(n)(P_{j,k} - O_{j,k}) + \sum_{c}\beta(c)\sum_{i}\delta_{i,k+1}W_{j,i,k+1}][O_{j,k}(1 - O_{j,k})]$$

where $\alpha(n)$ determines whether node n has output patterns to learn,

and $\beta(c)$ (0 or 1) determines whether to collect the error from connection $c$.

The learning parameters for weight changes are node unique - when the back propagation goes through a node, it uses its learning parameters for its internal and connection layers.

The role of one layer net nodes in the global feedforward and backpropagation mechanism is now clear. While helping to give structure and control to the net, they can only be used in connection to other nodes.

There are no restrictions on the morphology of the net. It may be a bush containing cycles. Nodes can be used independently or in subsets of the large structure or even contemporaneously in various configurations. These situations are kept "benign" by the invocation mechanism to be explained below.

### 3.2.3 Activation of the Net - First Visit

#### 3.2.3.1 Feedforward

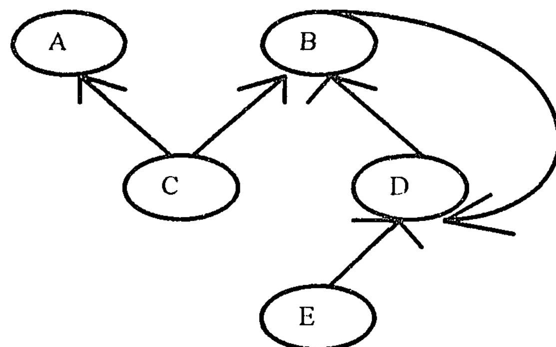
The objects related to by the activation engine are **net lists**. A net list specifies a subnet to be acted upon. Any subnet can be run independently, provided that it has inputs.

At this stage we can describe the activation of a structured backpropagation net in its most basic gyration. The objects related to by the activation engine are **net lists**. A net list specifies a subnet to be acted upon. Any subnet can be run independently, provided that it has inputs.

A list of net nodes is chosen to be run. For each node in the list, the list of nodes connected to it as input is visited first, if it exists. If the node then has inputs, either from the connected nodes or from an external source, the input is collected and fed forward through the layers of the node.

A first elaboration on the above version is the possibility of skipping a node that is chosen to be activated. There are a few reasons to make some nodes ineligible to run. A node can be put in "ignore" state, in order to avoid using a branch of the net. It can also be put in "freeze" state which enables using its activation as is, without running it. The node is also skipped if it has no inputs or the nodes connected to it have no inputs. A node can also be non eligible to run for the purpose of the following elaboration that handles non trivial net morphology.

A node is not activated more than once in one "run", even if it has more than one path to higher-level nodes that have been chosen. This policy takes care of multiple connections and of cycles. It results in the actual invocation starting from nodes that do have external input sources. In case of cycles, the result is usage of activation from a previous run cycle as input, in the same way that it happens in time delayed neural nets.




In the net drawn to the right, assuming that the net list to be run is "A B", that C is connected as input to A, that C and D are connected as input to B, that B and E are connected as

input to D, and that C and E have external input sources, then the order of feedforward will be:C→A→E→D→B, and the activation of B will be used by D in the next run cycle.

Another reason for skipping a node is that of run probability. Nodes in a run list can be assigned a run probability. The run probability can be specified with the node name in a net list that is to be run (as "A .5 B"), or it can be preset as a parameter of a node, or it can be specified when connecting nodes. In the latter case, as the nets connected to a node form a list of nets to be invoked by it, the feedprobability will be kept in that net list. The default run probability is 1. When a node is eligible to run it is determined whether to run it or not by the given probability. This can be important for both activation and training as it can express a different interdependence among nets than the one expressed by weights: the relative importance of the node or the contingency of the solution upon it, rather than its numeric contribution. It may mean that the system can spend less time training less important nodes and be able to reach a (possibly inferior) solution without them.

#### 3.2.3.2 Training

As with feedforward, any subnet can be trained directly, as long as it has inputs and output patterns. Trained nodes can be duplicated, connected and used in various contexts.

A subnet is trained by repeating feedforward and backpropagation cycles. A training cycle for a net list is performed by first executing the algorithm described above for feedforward, and then backpropagating the errors throughout the nodes in reverse order to the feedforward stage. The backpropagation is performed by the formulas given above in section 3.2.2.

Subnets can be trained separately or jointly, errors for a given run list can be backpropagated to connected nodes and on(back)ward, or the backpropagation can stop at the connection layers of the nodes at the top level (i.e. the nodes specified in the run list). This is decided by a parameter to the learning facility. Backpropagating errors between nodes can also be controlled by the backprobability parameter that can be assigned to a connection. This is a value between 0 and 1, which determines whether to collect the errors from that connection for the purpose of correcting the weights within the input node and the nodes connected to it. If the weights of the input node are to stay totally or partially independent of the output node, this option can be useful.

A facility is supplied to set default correlation between connected nets. It can be used when there is known correspondence between neurons of the output and input layers of the input and output nodes. It sets default weights and a better than random starting point for training.

As mentioned before, the learning parameters of each node are prevalent when going through the node. This is correct for the tenacity parameter as well, but only as far as requiring a certain improvement rate in each node. As for limiting the number of cycles, there is a general tenacity parameter.

### 3.2.4 Functions

So far we have discussed the net part of the net program. The net program, however, is not just a modular backpropagation net, it is a modular construction of both nets and procedural functions. As this work is done under the aegis of the computer science department, "functions" will herein mean procedural subroutines rather than a special kind of mapping.

The nets described in the previous sections exist and are operated upon in the environment of a procedural program, rather than as a closed process. They are created by a facility that is invoked in a statement-like manner from within a program and are used in the same manner. This means that a program can utilize structured nets for particular sub tasks.

But symbiosis between nets and programs in the net program is even closer, as net nodes are not only employed by functions, they can also invoke functions for their purposes. Functions, as well as nets, can be considered, graphically speaking, nodes of the net program. They are invoked by net nodes by an extension to the previously described feedforward mechanism, and they can in turn invoke other (or same) nodes or functions. That is what makes the net program a hybrid of nets and programs.

There are no restrictions on the invocations. Recursion is permissible. A function can re-invoke a subnet that has components that are currently in use by other functions in the recursion line; While weights are static, the context of net nodes is kept in a stack.

There are a few ways to connect functions to nodes:

**Input Functions:**

A net node can have an input function as its input source. As recalled from section 3.2.3, when a node is to be visited, its input nodes are visited first. Now, if there is a function connected to the node, the function is called before the node is activated.

A special data type is declared for input functions. They receive a few relevant parameters and return a structure which contains a directive (e.g. Proceed, Pass,

NoMore) and either activation for the input layer of the net node, output patterns for learning, or both.

For training, these functions can be called either once before the learning operation, to create a static set of inputs and patterns, or dynamically with each activation of the node.

These functions, as all others types of functions below, are free to do any sort of processing, including activation of other or same nets.

**Conversion Functions:**

For input and output. They can transform string input into a numerical value, and numerical output into a string.

**Numeric Manipulation Functions:**

Receive an input or pattern value just as it is about to be used, and return a replacement value. For example, they could produce random noise. They can also perform side effect operations that depend on input or pattern values.

**Sidekick Functions:**

Activated whenever the node is activated, before other types of functions, and before visiting its input nodes. These functions can perform any preparatory work for the activation of the node, including changing its input settings, and produce any side effects that are to take place with the activation of the node.

### 3.2.5 Activation of the Net - Second Visit

A list of net nodes is chosen to be run. For each node in the list, if it is eligible to run, if it has a sidekick function, the function is called. If the node now has its input from an external source, it is taken from there, if it has an input function, the input function is run and the input is taken from it, and if it has input nodes, they

are visited first, and the inputs are collected from the outputs of the connected nodes.

Backpropagation is performed for the nodes that were actually run, in reverse to the order in which they were invoked. Errors are collected from output nodes, if their backprobability is larger that 0, and in accordance with that probability.

### 3.2.6 Short Term Memory

A short term memory (STM) facility is supplied. An STM can be defined for subnets specified by the list of their highest level output nodes and the list of their lowest level input nodes. Training a new pattern will always be done together with the last N-1 most used patterns, where N is the STM size. The subnet can then be presented with training patterns one at a time, but it will in fact remember the recent most used patterns, and be able to produce accurate outputs for them. The STM is meant to enhance the continuity of the net's memory.

### 3.2.7 Activation of the Net - Final Visit

When learning, the feedforward stage changes significantly ifthere is an STM defined for the subnet being run. If the subnet has an STM, then the list of lowest level nets in the subnet is visited first. The activation of these nets is checked against the sets of the nets' activation stored in the STM. If the activation set is not found in the STM within each node's tolerance then it is put there, displacing the least used set if necessary, along with the output patterns of the output nodes. If the activation of the input nets is found, then the patterns for the output nets are replaced with the new ones. The system then performs a learn operation for the subnet with the set of all input and pattern and learning parameters sets kept in the STM along with the new set, and then the feedforward proceeds as usual (with the new weights).

# 4. The Net Program Language - C+NET

The net program scheme has been implemented as an enhanced version of ANSI C, a programming language which we call C+NET. This involved the development of a collection of subroutines to be used as an extension to C. A current implementation runs on a PC using MS-DOS and MS-C (and/or MS-Windows).

C+NET supplies facilities that can be invoked in a statement-like manner from within C programs. The basic objects upon which C+NET operates are net lists: lists of one or more net (node) names. Most statements (facilities), therefore, accept as their main operand (argument) a list of net names, e.g. "A B C,". Where applicable, a net name can be preceded by a float type constant e.g. "A .5 B C."

The basic five facilities that are used to create a basic multinode net program, train it and run it are:

| | |
|---|---|
| **CreateNeuNets** | to create backpropagation nets and name them |
| **connets** | to create connections among the nets |
| **setinpat** | to set the input source for nodes that need external inputs and patterns and to connect input functions to net nodes |
| **learn** | to train one or more nodes with or without those connected to them |
| **run** | to activate a list of nodes and those connected to them |

An elaboration and a description of other facilities follows; the underlined functions are the ones that would be used more often.

**<u>CreateNeuNets</u>(***net names, learning parameters, array of layer sizes***)**

Create backpropagation neural networks with the given number and size of layers and the specified learning parameters. The array of layer sizes has the size of each layer; for example {3,3,2,3,0} specifies 4 layers of sizes 3,3,2 and 3, correspondingly (0 specifies no more layers).
The learning parameters are given in a structure containing tolerance for neuron, tolerance for pattern, learning rate, smoothing factor, tenacity and feedprobability. All parameters have default values. A net will be created and named after each of the names given in *net names*.

**<u>connets</u>(***list of* ***from nets****, list of* ***to nets*****)**

Connect the output layers of each of the nets of the first list to the input layers of each of the nets in the second list.
If a float constant appears before a net name in the 'from' list, then this will be the probability for it to be invoked by the 'to' node when the to node is activated.
If a floating number constant appears before a net name in the 'to' list, then this will be the probability for the errors from the output node to be propagated back from it.

**correspond(***from net, to net, from neuron, to neuron, K***)**

Set a default correspondence between neurons in the output and input layers of connected nodes, when one is estimated. While changeable in training, it improves learning time. K , the correspondence, is a number between -1 and 1.

**<u>setinpat func</u>** *(net name, input source, function name, function parameters)*

Get a set of inputs and/or matching patterns from a function. A special function type is defined . It is possible to have the input come from connections and the patterns from functions. The function can also return directives, such as "pass this cycle for this input."

**<u>setinpat func dynam</u>** (*net name,source, function name, function parameters*)

As above, but the function is called each time the node is activated rather than for a set of inputs. The training set need not be pre-fixed. The input function can employ C statements for any processing and side effects it wishes to have.

**setinpat_file(***net name, source, file name***)**

Read a set of inputs for the node from the file specified. This may relate to inputs and matching patterns, or to patterns only, while the input can come from other sources.

**setinpat(**net name, source)

Choose a source of inputs: function, file, connections or duplication of the output patterns.

**set_convinpi(***net name, function***); set_convinpo(***net name, function***)**

**set_convpati(***net name, function***); set_convpato(***net name, function***)**

Supply conversion routines for input and output of a node.

**setinpat_manip(***net name, function for input manipulation, function for patterns***)**

Supply functions to manipulate the numeric value of inputs and patterns just before use.

**set_side_func** (*net name, function name, function parameters*)

Supply a (sidekick) function to be called for side effects or preparatory processing whenever the node is about to be activated.

**CopyNet**(*from net, to net*)

Create a new backpropagation net (node) with the parameters and weights of an existing one. Enables duplicating a trained node.

**run**(*net names*)

Feedforward: Invoke the nets in the list and the functions and nets connected to them feedforward. Each of the names in the list can be preceded by a run probability factor. The same is true for the net lists in the learning commands below. The input source can be connected nodes, or input functions, or a file, as set by the various SET_X functions.

**learn**(*net names, learn mode*)

Train the subnet specified by the 'net names' list, with the inputs and patterns assigned to them by the various SET_x functions described above. The learning mode can be local - just the nodes specified, or global - trickle on to connected subnets. The function returns the number of training cycles or zero if the pattern is in STM.

**freeze**(*net names*); **defreeze**(*net names*)

Disable/enable the nets for running. Their activations will be used as they were when frozen.

**Ignore**(*net names*); **retain**(*net names*)

Disable/enable running the nodes specified. Their activation will not be used when in 'ignore' mode.

**getact**(*net name*)

Get an array with the activations of the neurons in the output layer of the net specified.

**stmize**(*net names, input nets, size, learn mode*)

Create a short term memory for the subnet defined by the two lists of nodes. Learning will be done in sets of the given size. Meant to be used when the training set is dynamic and patterns arrive one at a time, but can also be used with large training sets. A pattern that is to be learned will be kept in the STM together with the latest most used patterns to form a training set for a learn operation that the system spawns.

**refrestm**(*net list*); **destmize**(*netlist*); **copystm**(*from netlist, to netlist*)

Refresh the STM contents; discard the STM; use the contents of the STM for another subnet with some common nodes.

**pushenv**(*net names*); **popenv**(*net names*)

Hide and restore the nets' running environment for a different invocation.

**storenets**(*file name, net names*)

Store the nets specified in the file given. Can be repeated for same file.

**retrievenets(*file name, net names*)**

Retrieve the specified nets from the given file.

**storenet(*file name, netname, storeas*); retrievenet(*filename,netname,storeas*)**

Store and retrieve using a possibly different name.

**storeall(*file name*); retrieveall(*file name*)**

Store and retrieve all nets known to the system.

**pushdinfo(*strings*); popdinfo(); prindinfo()**

Push debug information into a stack (when entering a process); pop it; print the stack.

**li_** *net names ...* **_st**

Create a list of net names of partial lists by concatenation. The list is a string of names separated by blanks. It is kept in a string pool. It can substitute for 'net names' in the function calls above.

# 5. A Natural Language Comprehension Model Based Upon and Implemented by the Net Program Scheme

## 5.1 Introduction

Language comprehension is a natural candidate to be addressed with a net program. To handle it, it is necessary to deal and experiment with much structure of which we know little, so a flexible structure would be useful. There are a lot of clues that may come in many forms. There is a combination of structural, contextual, lexical, transformational and rational-logical processing and the exact border lines are blurred. A good way to approach such a domain seems to be with a scheme that enables a flexible interaction of procedural, lexical and "neural knowledge," where "neural knowledge" is both *structure and* experience. *Structure* and *experience* can perhaps be associated with *innate* and *learned* knowledge when modeling natural understanding of language. Symbolic processing can be associated with both innate mechanisms (perhaps inaccessible rules) and learned ones (perhaps conscious rules).

Jean Piaget and Noam Chomsky have debated [17] the manner of acquiring language. Piaget's view is, very generally, that the knowledge of language is constructed in necessary stages, consecutively based upon each other. They develop from each other by way of abstraction and generalization. Chomsky, contrarily, assumes an innate fixed nucleus for the language knowledge. The two views, in our opinion, are less contradictory than they appear to be. Both assume some initial mechanisms and some learning. Both agree that a very certain fixed nucleus of knowledge is achieved. They differ, mostly, in assuming what is innate (or learned by an evolutionary process) and what is learned. If we were to create a model for language acquisition that follows either one, or, maybe, tries to experiment with finding a compromise, we would want to be able to manipulate

structure easily, and enable the employment of rules upon it conveniently and flexibly. We would need to facilitate creation of and experimentation with structure, training and logical reasoning. For these purposes, the net program scheme might be appropriate.

Connectionist models have been applied to problems relating to language comprehension. However, current systems that do work limit either their domain, as in [18, 19] which have an inherently limited vocabulary, or their range, as in [20] that (successfully) associates a limited number of responses (actions) with unrestricted input.

We shall try to create a system that may be in actuality expandable to deal with real texts.

We want to contribute to devising integratable parts of a solution to a large problem, rather than solutions to reduced problems.

Natural language processing is indeed a prominent part of mind, as it is interconnected with other faculties and is closely related to abstract thinking. It may exemplify thinking in the parallelism that it may have to the "language of thought" [28].

It may well be that the natural language problem cannot be solved without joining all of the means that we have, connectionist and symbolic. Classical systems do not have the flexibility, adaptability, fuzziness, robustness, graceful degradation and associativity that are needed for the task. Connectionist models do not have the capacity to process representations of structures and their compositions, nor can they do acceptable logical reasoning [3, 28]. As there probably are not any serious advocates of strict behaviorism left, there should not be a serious reason for anyone to believe that a backpropagation type net can

somehow learn to behave as if it knows the grammar of a natural language, without appropriate representations and reasoning mechanisms.

We hope to contribute to natural language comprehension in doing what is doable by each of the tools available, and trying to avoid forcing upon any of the mechanisms what is not feasible by it.

We shall not aim at full understanding of meaning as [18] and [19] do; it may be impossible at this stage. We shall not train for every possible word and every possible structure as [18] may have to do, as it may be hopeless. We shall try to learn syntax and be aided by "light" semantics. We shall try to build a system that can learn to handle many forms of sentences by training on examples containing frequent building blocks, so that it can induce recurring and composable sentence structures.

We aim at parsing. Parsing has an important role as it masters syntax, and syntax is essential for logical reasoning. This is why identifying keywords and responding to them has little to do with understanding language. Understanding necessitates the ability to parse, represent and perform operations upon structure. Understanding language may also be thought of as understanding somebody else's state of mind. But it is more than that. Humans can "operate upon" states of minds. They can have their own, while understanding another's. They can also manipulate others' states of mind, in relation to their own. They can convey information but they can also deceive. The messages that they convey are not direct responses to their state of mind, they are the result of rather elaborate reasoning. Messages that humans receive do not directly alter their state of mind; they first pass a reasoning process. All of this is done using language. This means that language understating must involve logic and therefore structural representations and syntax.

There have been impressive results for parsing with some programs and some dedicated computational schemes [21, 22, 23]. We do not know of substantial success with neural nets. The problem, though, is far from solved. On one hand there appear to be universal syntactic rules [24, 25, 26, 27]; on the other, it is tantalizingly difficult to define and employ rules. The problem, which the human brain appears to solve so effortlessly, seems intractable when it comes to searching for an algorithm.

Our approach uses a neural net much like an oracle. The oracle takes care of the parts of the domain that do not have a good known tractable deductive algorithm, and interacts with a parser. It derives its "wisdom" from its own structure and the inductions and experience coming from the data presented to it. With the oracle's help, the parser's task is much simplified. It builds a parsing tree, presenting situations to the net and asking it for decisions as to what to do next in each situation (e.g. push or pop the element in consideration).

The structure of the net is such that it can receive an encoding of the partial parse tree, create reusable internal representations for its constituents, and output a decision. The parser keeps the codes created by the net for words and phrases, to re-present them to the net as parts of larger phrases. The parser, therefore, does the programmatic work, the nets do the decisions and encoding.

The system we are presenting here is a result of experimentation with, and elaboration of, the system we presented in [33] and is overviewed in [34].

## 5.2 The Input

The input to the system is continuous text with intermingled optional directives. The directives relate to meaning, i.e. lexical values, of words and to the strucure of phrases. The net learns to reproduce both meaning and structure. Lexical values are kept in a dictionary as well as taught to the net. The dictionary is managed according to word level directives. These are given in square brackets ([ ]). Phrase level directives drive the parsing and the definition of phrases; that is; they decide what a phrase includes and of what speech part and syntactic role it is. These are given in parentheses. Angle brackets (< >) enclose text that is to be processed without creating and training nets, so as to check the validity and correctness of the directives. Comments can be given in braces ({ }) and can be nested.

An example text is:

You [pronoun noun gend(anim) num(singular plural)] (start pivot,noun)
bake [verb tran form(finite) sem(food)] (start cont pivot)
a [article] (start push)
cake [noun gend(inanim) sem(food)] (cont pivot) . [punct] (fullstop)

The syntax of directives is given below:

**WORD LEVEL DIRECTIVE** := [DICACTION,LEXIVAL]
[LEXIVAL]

DICACTION :=
todict
update
combine
enhance
ignore
remove
pick

```
LEXIVAL :=
        SPEECHPARTS
        SPEECHPARTS QUALIFIERS
                SPEECHPARTS :=
                        SPEECHPART ...
                QUALIFIERS :=
                        QUALIFIER(VALUES) ...
                                QUALIFIER :=
                                        gend
                                        person
                                        num
                                        syn
                                        stem
                                        var
                                        form
                                        tense
                                        sem
                                        any additional qualifiers are permitted
                                VALUES:= VALUE ...
                                        VALUE:= string
```

**PHRASE LEVEL DIRECTIVE** := (PARSE INSTRUCTION,LEXIVAL,ROLES)

```
        PARSE INSTRUCTION := OPERATOR ...
                OPERATOR :=
                                start
                                push
                                pop
                                cont
                                adjoin
                                lookahead
                                enclose
                                push
                                record
```

Dictionary instructions are optional. When a word is followed by a dictionary instruction (or a few), its lexical value is decided by the instruction(s), and the system uses the value for its training. If there is no lexical instruction but there already is a lexical value kept for the word in the dictionary, then the value in the dictionary will be used; otherwise, the system will use whatever the network knows about the word.

The general dictionary instruction is in the form [DICACTION,LEXIVAL], but normally a dictionary directive would just be: [LEXIVAL], and the DICACTION would default to enhance which means: if the word has no dictionary entry, create one with LEXIVAL, otherwise, consolidate the new and old values by adding the new components in the new lexical value to the old value. The other instructions mean:

todict: create a dictionary entry

update: replace a dictionary entry

combine: combine the given LEXIVAL with the one in the dictionary, use the combined value but do not update the dictionary

enhance: combine the value in the dictionary and the new ones and update the dictionary entry

ignore do not use the dictionary for the purpose of the current context

remove delete the dictionary entry

pick use the given part of the dictionary entry

LEXIVAL, or lexical value, consists of speech part(s) and qualifiers. They are not pre-defined, but some basic ones are assumed. Speech parts are terms like noun, verb, article etc. Qualifiers are variables with values such as gend(male) sem(young). Multiple values are allowed and are entered by including more than one value within the parentheses that follow a qualifier, or by repeating the

qualifier. Values can be negated by being prefixed by a minus sign such as in num(singular -plural), which means singular and not plural, rather than both singular and plural as in num(singular plural). The qualifiers and values given in the input cause the creation of corresponding net nodes, as will be seen later. Any new speech parts, variables and values can be given. The coding is not done across a fixed set of micro features; it uses relevant, and possibly, introduced ones.

Some qualifiers that the system assumes are:

| | |
|---|---|
| gend | gender (male, female and perhaps others) |
| person | first, second, third |
| num | singular, plural |
| syn | synonymous to |
| stem | the basic form of the word, the lexical value will be enhanced by the lexical value of the stem word |
| var | variation of, the lexical value will be enhanced by the lexical value of the main variant |
| form | finite, irregular "ed" "ing", etc. |
| tense | tense name |
| sem | semantic values, e.g. good, young, positive |

A phrase level directive determines how to construct the parse tree and what is the function of the immediate word or phrase within its context. It relates to the current entity being considered, which may be a word received from the input or a previously constructed phrase or a phrase "under construction". The parse instruction in (PARSE INSTRUCTION, LEXIVAL, ROLES) tells the parser how to proceed with the current entity in relation to the partial parse tree.

The parse instruction consists of combinations of operators:

| | |
|---|---|
| start | Start a new phrase. |
| pop | Close the open phrase, recheck the entity in relation to the phrase one level up. |
| cont | The entity belongs to current phrase, put it in its continuation. |
| start cont | Start a new phrase at the same level of the open one, close the open one. |
| start push | Start a new phrase as part of the open one |
| start pop | Close the open phrase, start a new phrase, recheck the new phrase in relation to the phrase one level up. |
| start cont enclose | Close the open phrase, start a new one at the same level and enclose the two of them within a new higher level phrase. |
| start push enclose | Start a new phrase within the open one and enclose them within a new higher level phrase. |
| adjoin | The entity is part of a bigger one (an expression). |
| lookahead | Withhold decision. Read and encode the next few words, then redo the parsing. |
| pivot | The current entity serves as a head in its phrase, the phrase will get (the relevant parts of) its lexical value. |
| record | Learn to encode the last element, but ignore it as part of the phrase |
| fullstop | Finish parsing the sentence, display it. |

The LEXIVAL within the directives is optional. It assigns a lexical value to the phrase being constructed (the one to which the parse instruction relates), and its format is the same as LEXIVAL in the dictionary direction. The roles are also optional; they are the syntactic roles (subject object etc.).

An extension of the previous example input should clarify this a bit. Here are a few simple sentences. The lexical value of words is remembered once entered, the second sentence has only one new word. The last sentence is very similar to the first one, it is given without directives, so that the system will have to parse it 'all by itself.'

```
{noun phrase This is just a comment}
        You [pronoun noun gend(anim) num(singular plural)] (start pivot,noun)
{verb phrase}
        bake  [verb tran form(finite) sem(food)] (start cont pivot)
        {noun phrase}
                a [article] (start push)
                cake [noun gend(inanim) sem(food)] (cont pivot) . [punct] (fullstop)

{noun phrase}
        I  [pronoun noun gend(anim) num(singular)] (start pivot)
        {verb phrase}
        eat [verb tran form(finite) sem(food)] (start cont pivot, verb)
        {noun phrase}
                a (start push,noun)
                cake (cont pivot) . (fullstop)
```

{noun}

The [todict,article] (start,noun,subject)

boy [todict,noun gend(anim male person) num(singular) sem(young)]

(cont pivot)

{verb}

ate [verb tran stem(eat) form(irregular) tense(past)]

(start cont pivot,verb,predicate)

{noun}

a (start push,noun,indirect object)

cake (cont pivot) . (fullstop)

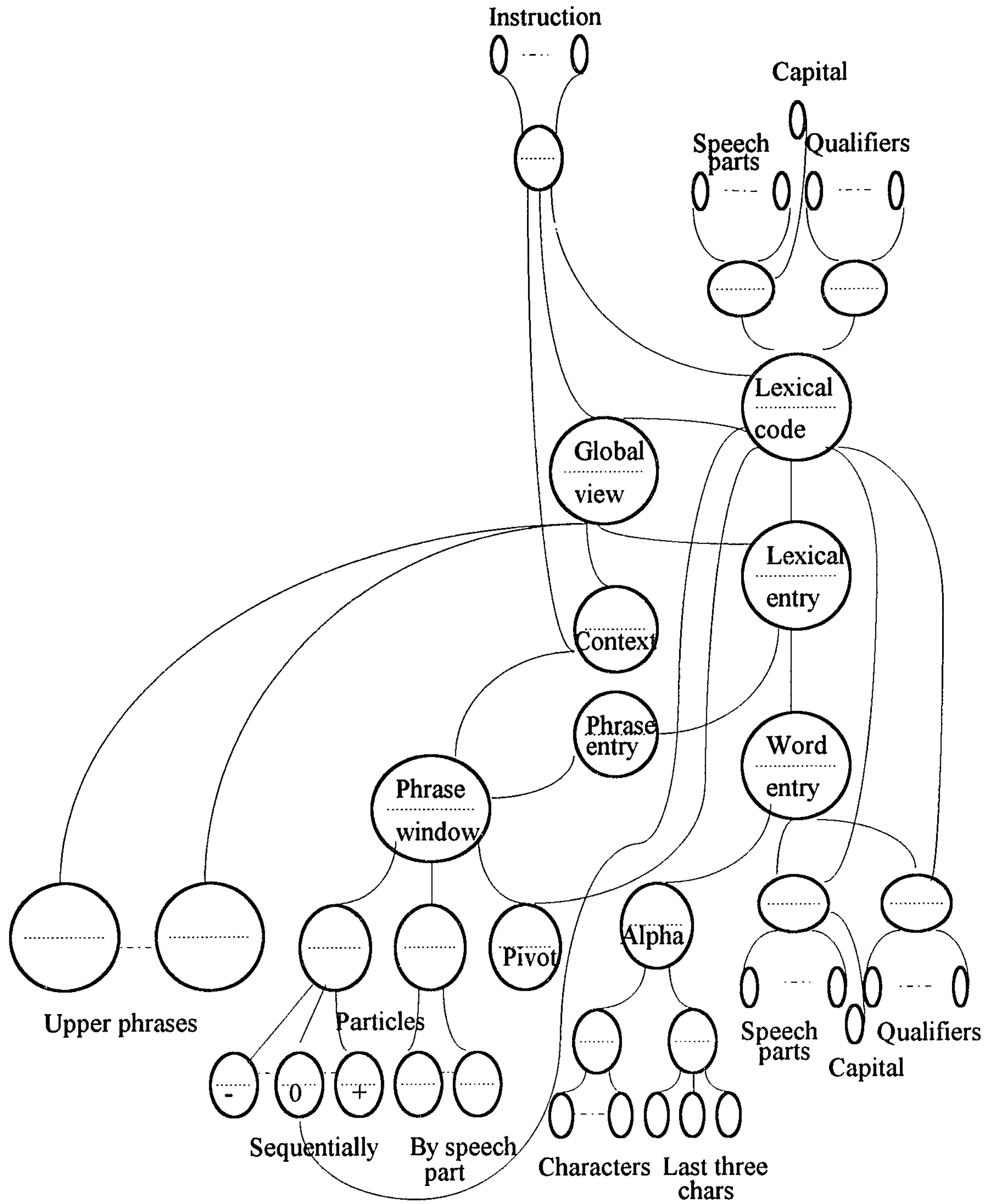
{see how it does}

I eat a candy.

The system teaches the net to output and encode the lexical values of both words and phrases in a given context, and to give the relevant parse instruction. The directions are used as clues. The parsing process can work with or without them; hen clues are given, the system learns and uses them;when they are not, the system extracts both the lexical values of entities and the parsing instructions from the network. Partial direction of the parsing is also allowed; at any point in which there is no external directive the net's knowledge is used and at any point in which external knowledge is given - the net is trained with it.

The net is therefore structured so that it can receive as input the current context and entity considered, create codes for lexical values, and output the parse instructions. Below is an overview of the net's structure, which will be detailed next.

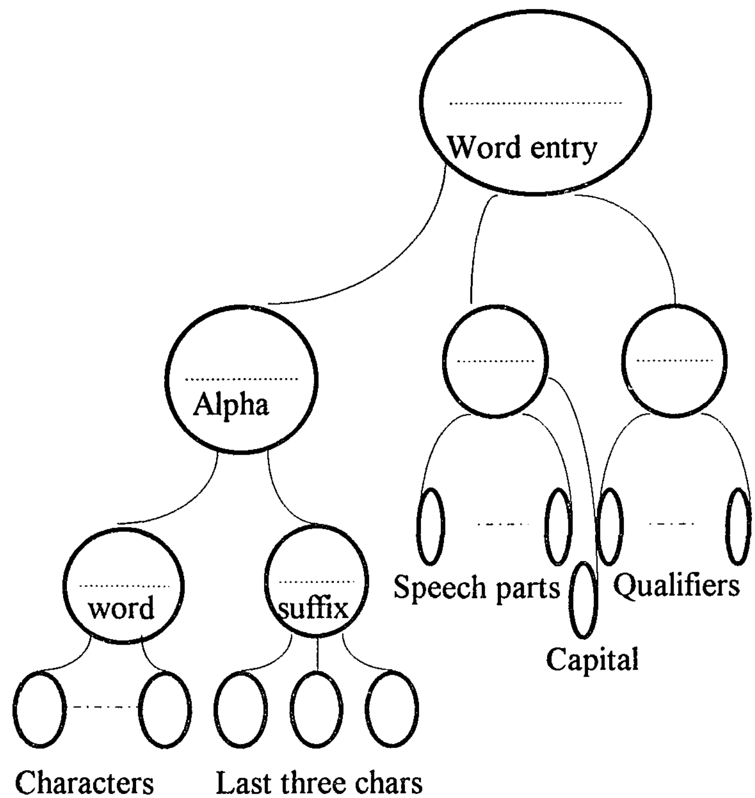
## 5.3 The Language Comprehension Net

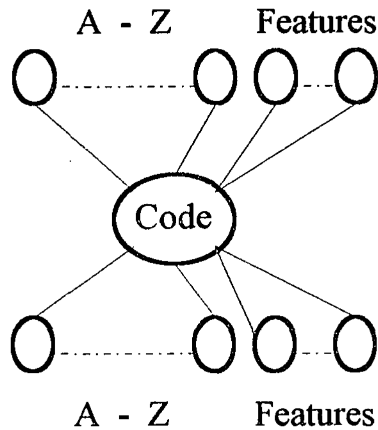


### 5.3.1 Subnets, Connections and Usage

The subnet for word entry, which appears in the lower right part of the net overview above, and is repeated below to the right, enables entry of multiple aspects of the word.

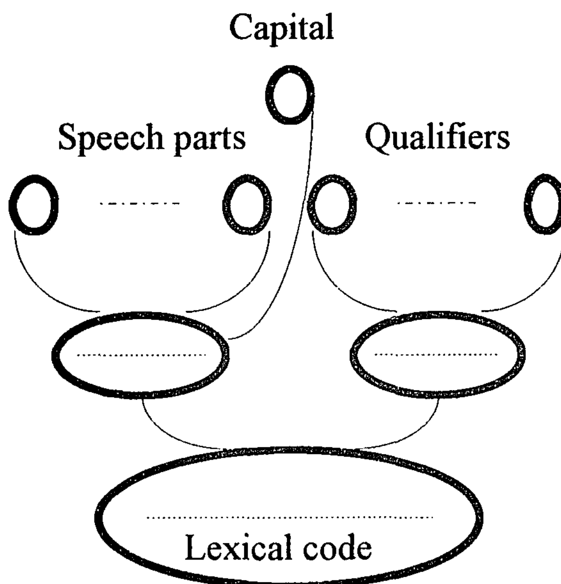
The coded characters of the word are the input to the little nets in the bottom left. The three last characters are also given to the next subnet called 'suffix', as the word ending may be of importance to its meaning ("ing", "ed" etc.). The next subnet is dedicated to the lexical value of the word. Each speech part that appears in the input causes the creation of a one neuron subnet connected to the speech part net. Each value of each qualifier that appears in the input, other than **var syn** and **stem**, causes the creation of a one neuron node connected to the qualifier subnet. The one extra one-neuron node marks whether the word starts with a capital letter or not.






The codes for characters are created by another net that learns to create a representation for each character. Its input are one neuron for each characters and some extra neurons for features such as the character being a vowel or part of a given phonetic group. The net learns to activate the corresponding output nodes when presented with a given character and its features. The activation of the central node is used as the character code.

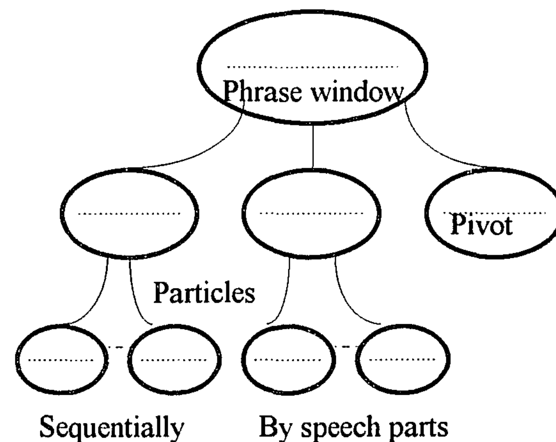
The parse net works to some extent in the same way as the character coding net. It learns to encode lexical values. The usage of the coded values is, however, more elaborate. In order to create a meaningful representation of lexical values it has output nodes that, similarly to the word entry subnet, correspond to lexical components:




The subnet that outputs a lexical value (upper right part of the global view and repeated here at the left) has nodes that correspond to components of lexical value, similarly to the previously discussed subnet that serves to enter a lexical value. Once the net learns a lexical value, which means that it learns to output the

proper activation for the output lexical nodes, the activation of the **lexical code** subnet serves to represent a coded word to the system for further reference.

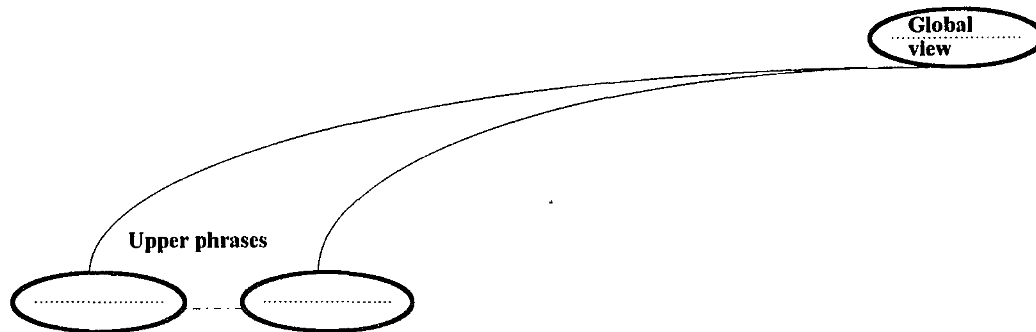
Phrases too have lexical values, which are encoded by the same subnet, so that they are represented in the same way as words. Phrases are entered to the system both for encoding and as context, which will be discussed later. The phrase entry subnet has the structure:




The ground level nodes in the phrase entry subnet receive as input, codes that are created by the lexical code subnet described above. These represent parts of a phrase which may be words or phrases. The pivot node is activated with the code of the phrase head (if there is one). The system can (be trained to) determine while parsing that a given part of a phrase is its pivot, which could help tell the function of the whole phrase. The **"particles sequentially"** subnet is a window to the current phrase; the nodes connected to it correspond to the last few elements in the phrase and (when needed) a few lookahead elements. They are activated with the codes of the phrase parts.

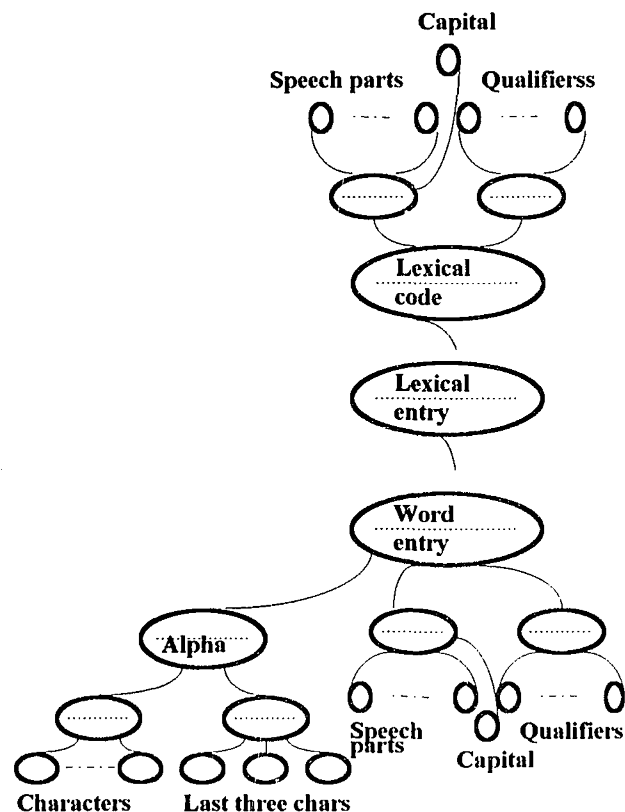
The **"speech part"** subnet has nodes that correspond to elements in the phrase by speech part, e.g. if the phrase has a noun, the noun node would be activated with its code. The inclusion of entities of certain speech parts in the phrase is induced during parsing.

The leftmost nets in the overview are the **upper phrases** nets:




These are the codes of the upper level phrases of which the current phrase is a part, that is, the immediate phrase to which the current phrase belongs, the phrase to which the latter belongs, etc., which give *a complete context in which the phrase is set.*

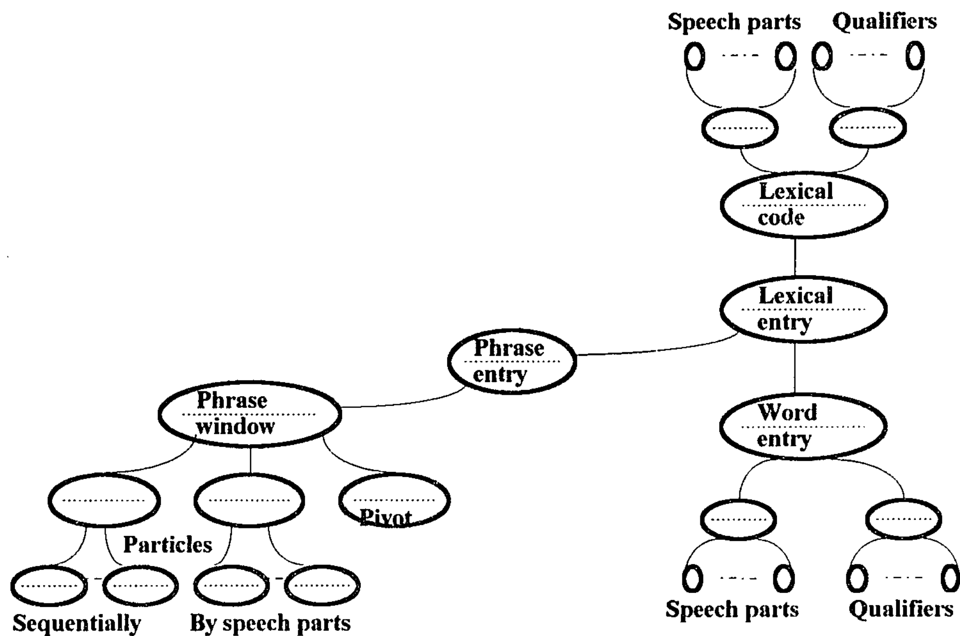
We can now describe larger subnets. Let us suppose that we want to encode a word. Suppose, further, that it is the first word in a sentence, which means it has no established context. Then the subnet used would be:




The word is entered along with its lexical value, but only the lexical value is in the output side. The alphabetical value of the word is entered as described before: the characters are coded, entered to the "alpha" subnet that has nodes for each character in each position in the word and to the "suffix" subnet that has nodes for the last three characters. The lexical value of the word is determined according to the

dictionary, and the input and output nodes that correspond to components of the lexical value are assigned a value of 1. If the word is capitalized, then the input and output capital nodes are assigned a 1 value as well. The net is now trained to output the desired patterns, and once the net is trained, the activation of the **"lexical code"** subnet represents the word.

Now suppose that we want to encode a phrase whose elements are already coded and whose lexical value is known. The subnet we would use to enter such a phrase would be:

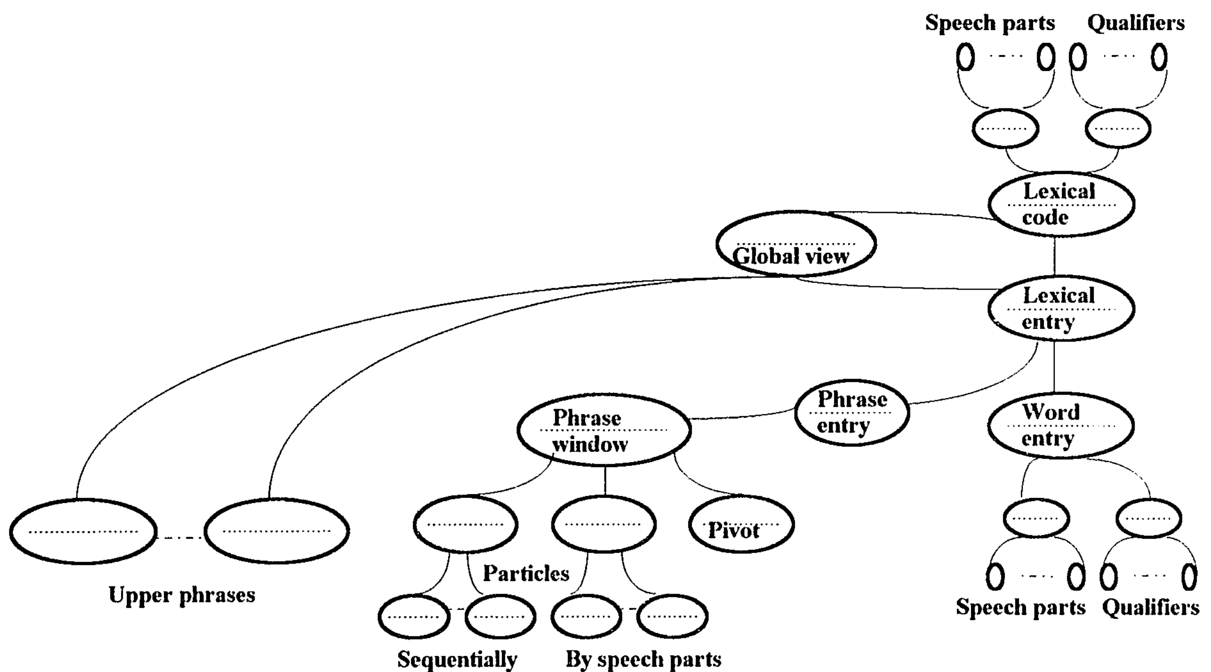



The codes of the phrase's elements are entered in the sequential window; the elements that have a speech part assignment are entered in the phrase speech part subnet. If the phrase has a known pivot (head), its code is entered to the pivot subnet. If the pivot is a word (rather than a phrase), then the subnet for word entry (shown in the previous figure) is used to enter the word. This is meant to teach the net to use its knowledge about the head word to extract knowledge about the phrase. Since the lexical value of the phrase is known, it is entered through the lexical entry subnet of the word entry (bottom right). The output nodes for lexical

value are set correspondingly, and the net is trained to reproduce the lexical value. The code of the phrase is, as was the case for a word, the activation of the "lexical code" node.

Now *we have the same encoding for both phrases and words*, so that *we can use both in the same manner* as parts of bigger structures, i.e. larger phrases. The code of a phrase, like that of a word, tells its function as a speech part and gives clues as to its form and meaning, to an extent depending on the level of detail given in the input, and the previous knowledge of the net. Recall that these are clues, not a complete and definite set of micro features.

The encoding described so far is insufficient, though, because it ignores the context in which the word or phrase appears. We want the net to be flexible and clever enough to take into consideration the context when determining the function and meaning of an element. We therefore handle any entity within its context. Encoding a phrase within its context is only slightly more elaborate then before:

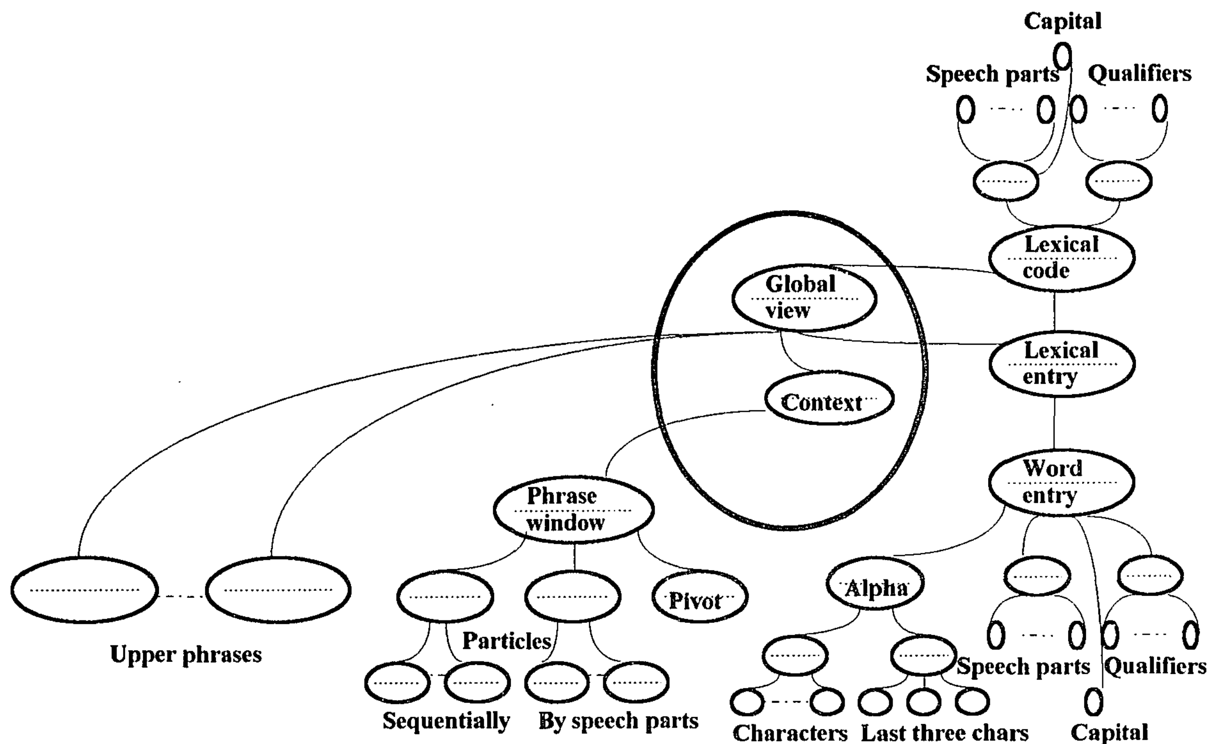


We just had to add the codes of the upper phrases (the phrase within which the phrase is enclosed and so on to higher levels). The upper phrases have dedicated nodes and they are connected to the "**global view**" net in the center, which is connected, in its turn, to the lexical code node, and supplies the context.

In the case of a word, the situation is more difficult, because the immediate (and crucial) context of a word is the incomplete phrase in which it appears. This phrase is being constructed while parsing; it would be insufficient to use its intermediate encoding; we need an accurate description of its elements. This means we should use the phrase entry subnet to input the context of a word. But this provokes a problem. The system learns to output the lexical value of a phrase in response to the phrase coded through the phrase entry subnet, but when the phrase entered is used as context we want the net to output a possibly different lexical value, that of a given word in the phrase. The same subnet is supposed to serve for two different purposes that may be contradictory. (So far we did use the same subnets for a few purposes, but they always coincided in the sense that they utilized and enhanced the same skill.)

The solution to the problem is to encourage the phrase subnet to create local representations of the structure of the phrase, a representation which can then be used by other nodes for more than one purpose. There are several measures that can be taken to achieve this. The methods to be used have the extra benefit of demonstrating the need for some non-trivial facilities in the $C^{+NET}$ language.

To begin with, the phrase subnet can be connected via different nodes to do different tasks. The weight changes can then be encouraged to occur in the interim connection nodes and connection layers, rather than in the phrase subnet itself. Here, when serving as context, the connection used for the phrase subnet is changed to the **context** connection rather than the **phrase entry**:

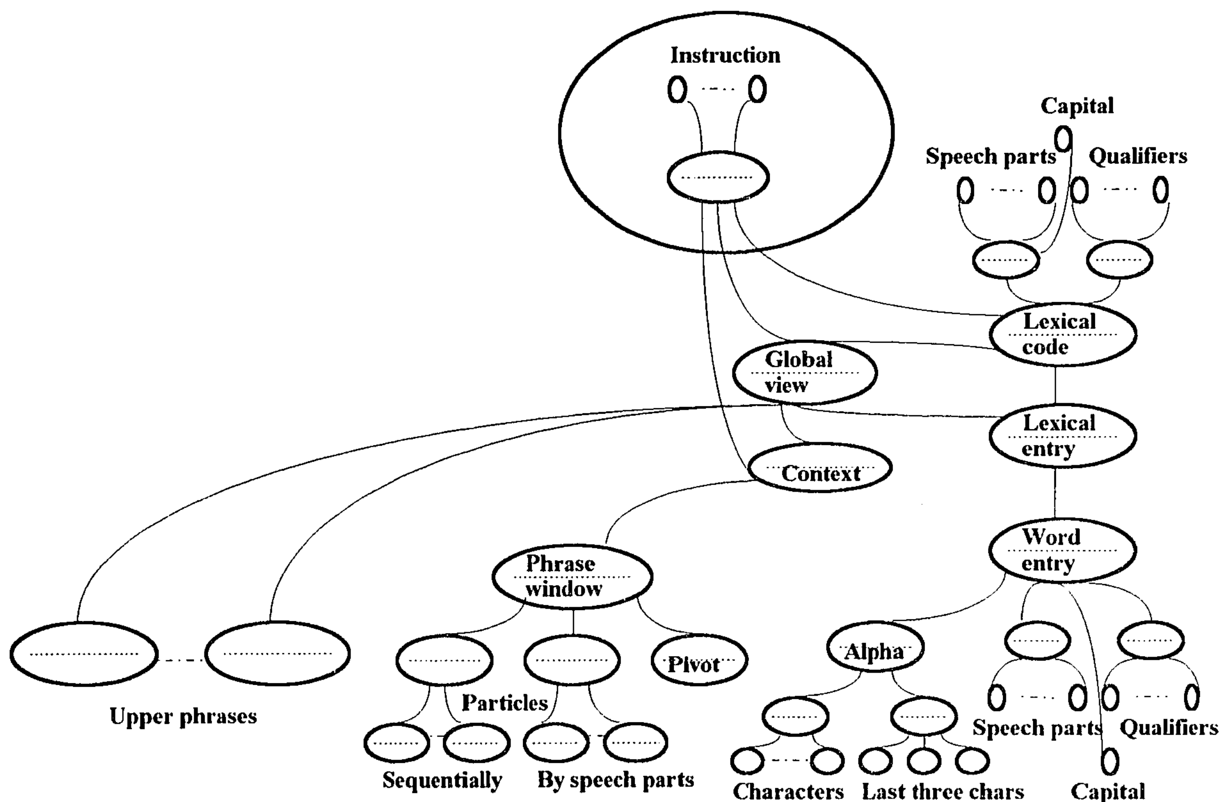



The connection of the phrase net as context has a lower backpropagation probability than its direct connection to the lexical entry subnet. The backpropagation probability (see above in the C+NET section) tells the learning engine at what frequency it should consider the errors that originate in a certain connection for correcting the weights backward beyond the connection. This means that the weights in the phrase subnet will be corrected more often and change more when learning the code of a phrase than when serving as a context of a word, so they would be biased toward reflecting the learning of a phrase code. Additionally, the learning rates in the internal nodes of the net can be increased when encoding phrases and decreased when serving as context, so that the plasticity of the weights will be higher when creating an internal representation of a phrase than when using it. (See **setparm** in the C+NET section above; the parameters of nodes are also maintained when using the STM facility). Whenever possible, a phrase is first encoded, even with partial information gathered up to the point of its use, and then used as context. With the STM facility the usage in

training of a phrase as a lexical entity and as context is done within the same learning set, and controlling the backprobability and learning parameters ensures that the weights will be biased toward creating a local representation of a phrase.

So far, our net has learned to encode words and phrases separately or within their context. The encoding operation served, of course, for more than encoding; it also trained the net to determine the lexical value and the function of words and phrases within a context. By classifying words first, it could then classify, with the help of the parsing engine, phrases of increasing complexity.

But we want more. We want the system to be able to reconstruct and mark the phrases in a sentence. The structure of the net was designed with this as its foremost task, and now that we have most of the net, we can add the top of the little edifice. That top is the instruction subnet:



The instruction subnet decides what to do when presented with a parsing state. The state is characterized by a partial parse tree on one hand and a word or phrase on the other hand. The instruction determines the connection of the element that is being considered to the partial tree. The instruction subnet is a straightforward addition to the net we have so far; it simply has to know the global state of the parsing and details about the next element.

The output nodes of the instruction subnet are a set of one neuron nodes, one node for each possible component (operator) of the parse instruction. They are activated as an integral part of the global net: when the net is presented with a new element to be encoded or to extract knowledge upon, it is also presented with its context, which is the partial parse tree. The output would be the activation of the lexical value nodes telling the lexical value of the element considered, and the instruction in the form of the activation of the instruction nodes that tells the relation of the element to its context, i.e. to the parsing tree.

There are a few more connections that we have not yet described. There are direct links from the input **speech parts** subnet and input **qualifiers** subnet to the **lexical code** node. This is to encourage the net to exploit the lexical value of a word when creating a representation for it. There is a similar direct link from the **pivot** node to the **lexical code** node. This link is being used only when creating a representation for a phrase and not when using the phrase as context for a word that is being coded. This is meant to enhance the linkage between the head of the phrase (which code is the activation of the **pivot** node) and the meaning and function of the phrase, and to help separate the functionalities of the **phrase entry** subnet as a depictor of context and a depictor of an element in a larger phrase. Conversely, there is yet another direct link from the node representing the current element, that is the **particle[0]** node in the **particles sequentially** subnet, to the

**lexical code** node. This link is used only when coding the current element, and provided that there is an existing code for the current element. There can be an existing code for an element that is being coded as a result of a **lookahead** operation, of back track in the parsing, or the code can come from the dictionary.

Re-coding an element is needed because of the weights' plasticity. The net is always trained using the STM (Short Term Memory) facility. This means that the recent elements and their lexical values have accurate and current representations. This is not necessarily the case for words appearing in previous sentences or in remote phrases or in other texts. As the system sees more words in more contexts, it learns and it changes. Old representations may then become less representative. One way of dealing with it is using the learning parameters: the rate parameters of the output speech part and qualifier nodes are low. Nevertheless, this cannot prevent changes in representations as the learning of new words and new ways of using known words needs to occur. But this need not be a disadvantage. Rather, this can help emulate the natural way of learning words.

The meaning of words is probably learned in an accumulative fashion. Each time we encounter a word, we probably get an enforcement to some nuances of its meaning. To effect a similar process, word representations should also be learned cumulatively. The old codes for words can be used in the learning of new codes within the new context. A similar strategy has been taken in [16] and [19], but in our model the very meaning depends upon usage and context, and the encoding is aided by them as it is performed within the global net. The code is created and used within a context; it need not render a full exact array of micro features as it is not a simple compression of such an array. Cooperatively with context and usage, though, relevant micro features are taken into consideration when encoding, and they, too, are reflected in the code.

We are saying, then, that our *accumulative meaning reflects the history of usage.* The net encodes words and phrases while learning some semantics but, more effectively, while using the words to make parsing decisions. Furthermore, because of this design *the system continuously learns during operation.*

Returning to the practical net program level, the codes created for words are kept, beyond the STM, in the dictionary, along with the words' lexical values. When a word is to be encoded, the components of its (partial and relevant) lexical value, if given, are entered through the **word entry** subnet, whereas the existing code, if there is one, is entered through the node for the current particle (which is the **particle[0]** node in the **particles sequentially** subnet).

### 5.4 The Parser

Now that we have presented the net, we can describe how it is used to help do the parse work.

The engine of the language comprehension program is the parser. The parser is directed by the input text, the directives intermingled in the input text and the decisions of the neural network.

The parser maintains a structure of phrase elements which contains all the information that is known about them (e.g. their lexical value and their function in the phrase). The information can either come explicitly from the input or be induced by the nets. The structure also contains pointers that define the parse tree. Initially, the first element (word) in the phrase is considered against the sentence level, and then the parser decides the relation of each next element, in its turn, to the position held in the tree. This process is not necessarily sequential - during parsing, as phrases are constructed, the parser can be called to decide the relations of phrases at various levels.

When the parser is called to decide a relation it checks to see whether lexical information about the element considered (from the input or the dictionary) and the parsing directives are available. If either type of information is available, the parser trains the net with it, otherwise it tries to extract the information from the net.

If the parse instruction gotten from the net is not decisive, the parser tries to look ahead and extract some information about the next few elements and then retry using the net. If this does not yield a parsing instruction, the parser can add the element under consideration to the current level of phrase and go on to the next element, or continue to parse the next part of the phrase and then come back to

complete the current parsing. The lookahead procedure can also take place while training: if the net is stalling when learning an instruction, the same learning is tried with a lookahead, because the difficulty may indicate lack of evidence. Assumedly, similar difficult situations will arise in training and usage, so that the lookahead training will be of benefit for the usage of the net in situations that need lookahead.

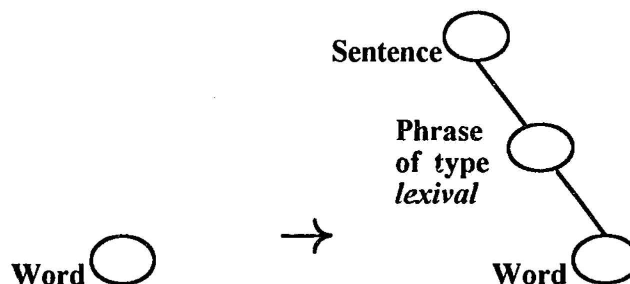
The information about the lexical value of words is taken from a dictionary, but the net is trained to give that information while encoding it. The same complex of nets learns to give the parsing decisions and lexical information about words and phrases. The net does not learn to encode words as such, but to output some information as a response to parameters of a word and give an encoding for that information. Since the network has as input the spelling of a word and its context, it can give some intelligent codes for words that do not appear in the dictionary. This means that the parser does not depend on a fixed vocabulary, it can have the "feel" for unknown words.

The word level directives (given within square brackets) direct the creation of the dictionary entries, the extraction and usage of information from the dictionary, and the training of the net for lexical values. When given within a text they relate to the last word in the text. More than one directive can appear consecutively. Normally, the word level directive for a word would be its lexical value, but more options are available to manipulate values, as detailed above in section 5.2.

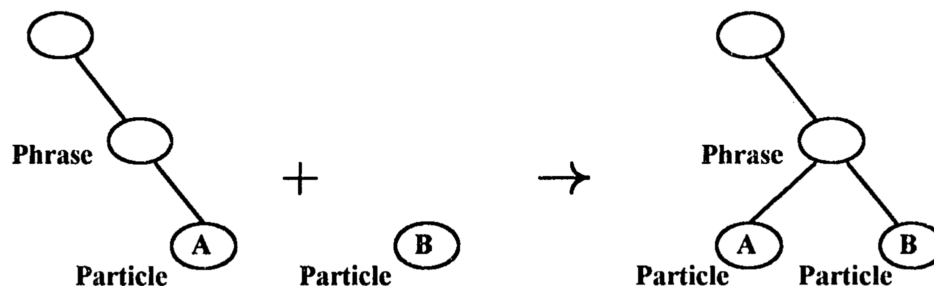
For constructing phrases, the parser uses a set of parsing instructions. A sequence of parsing instructions can define the parsing of a sentence. The parsing instructions were detailed above in section 5.2. As a rule, the parsing instruction relates to a situation which is defined by a partial parse tree and a "**particle**" (word

or phrase) that is being considered. The instruction directs the parser where in the partial tree it should place the particle and how to proceed with the parsing.

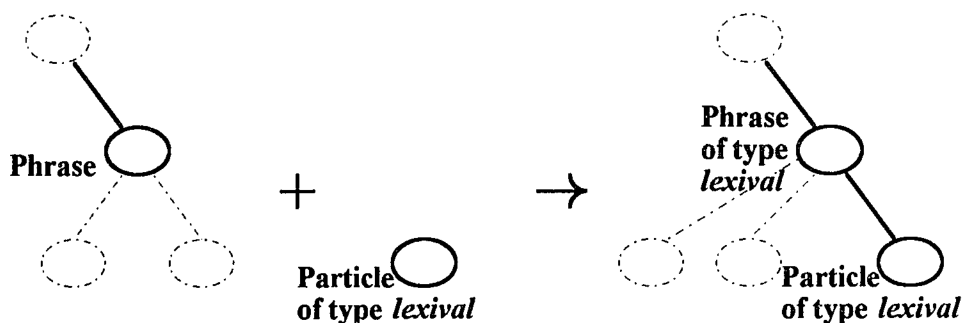
The parsing of a sentence would typically start with an instruction like (start,*lexival*) that follows the first word of a sentence. This means start a phrase headed by the entity being considered which is, in this case, the first word of the sentence, and assign *lexival* to the phrase (the word has its own lexical value assigned to it by a word level directive, or found in the dictionary, or extracted from the net). After the start operation, the parser moves to the next particle. Recall that the basic elements for parsing are "particles." Particles may be words or phrases that have been constructed in performing previous parse operations. Pictorially,the start operation would look like:



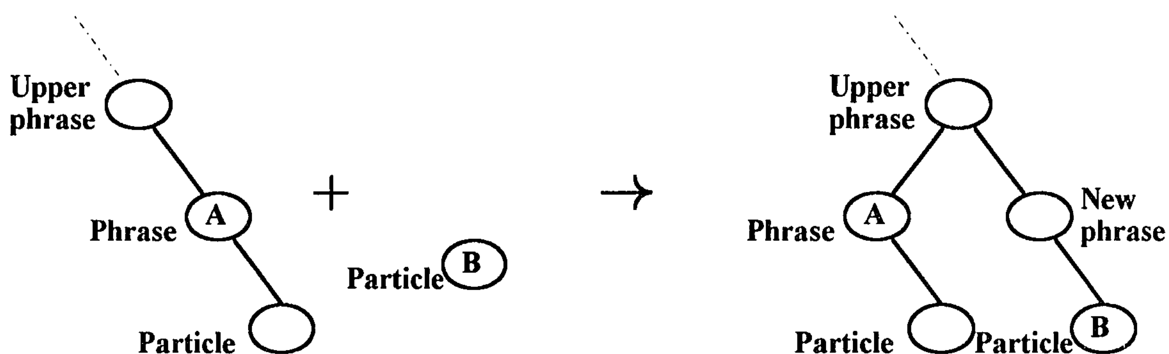

If the next particle belongs to the same phrase it would be followed by (cont), which would tell the parser to attach it to the same phrase, and go on for the next input. The particles slide "to the left," the previous particle moves backward, the particle just attached will now be the point of reference for parsing, the next particle will be the particle on focus with which the next parse instruction is associated:



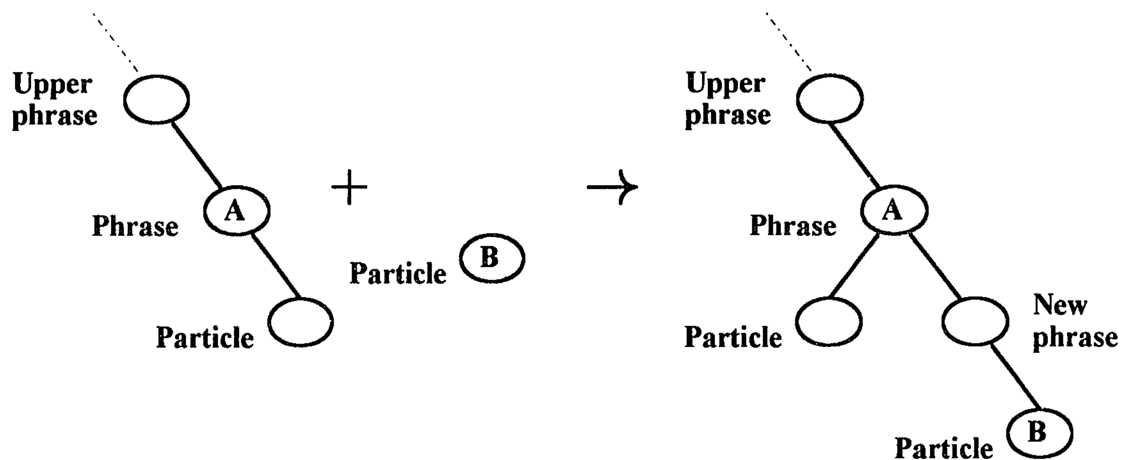
If a particle in the phrase serves as a head, which means it determines the lexical value of the phrase, then the instruction associated with it can have pivot as one of the operators of which it is comprised e.g. (cont pivot). This would cause unification of the lexical values of the head particle and the phrase to which it belongs. As both *lexival* and pivot are optional, either one or both can determine the lexical value of the phrase.



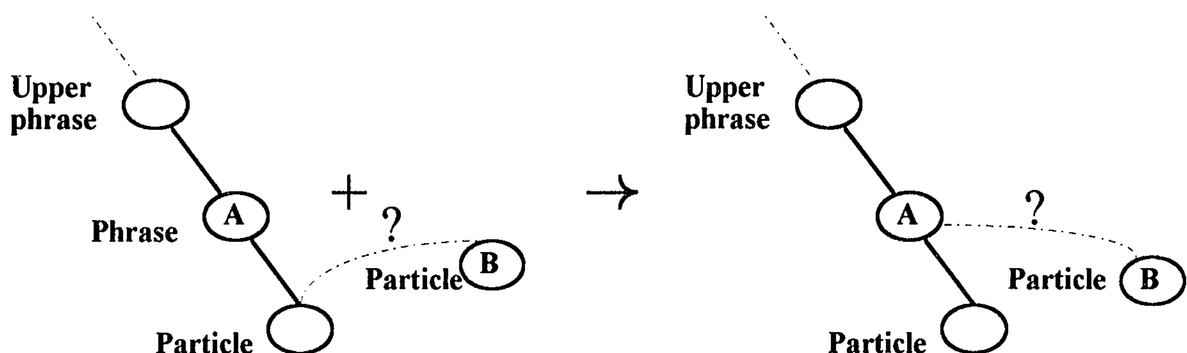

There are a few ways to begin a new phrase now. (start cont) will close the current phrase and start a new phrase headed by the particle with which it is associated. The new phrase will be attached to the partial parse tree at the same level of the previous phrase. The parser will then go on to the next particle, using the new particle as the point of reference.



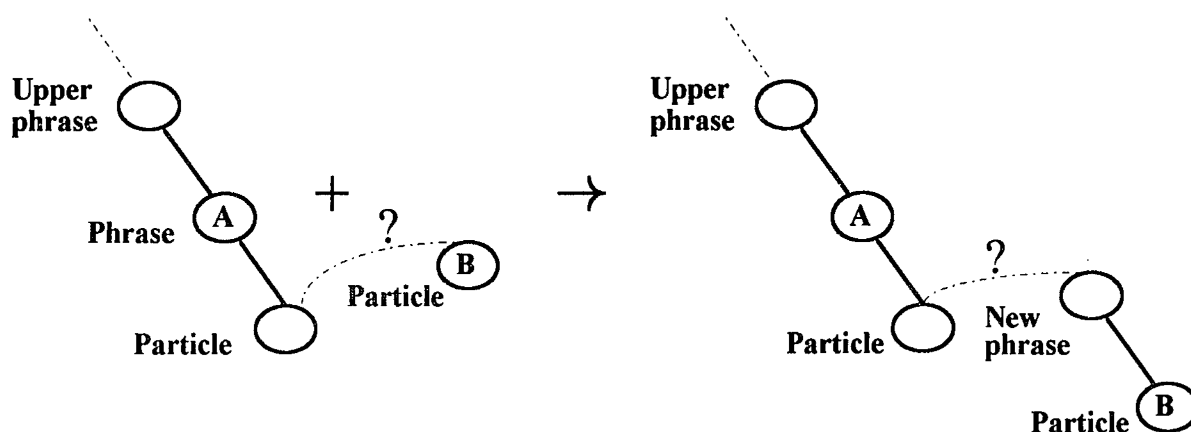
start push will also close the current phrase and begin a new one with the particle being considered, but, conversely to start cont, it will attach the new phrase to the parse tree as a subordinate of the previous phrase in its continuation. The parsing will then continue with the next particle.




The pop operation is an antecedent to other construction operations. It indicates that the current entity must belong to a phrase in a level higher than the current open one. When the parser encounters pop, it closes the open phrase, and then looks for an instruction defining the relation of the particle under consideration to the phrase in the next upper level. The next instruction could be yet another pop or any of the construction operations.

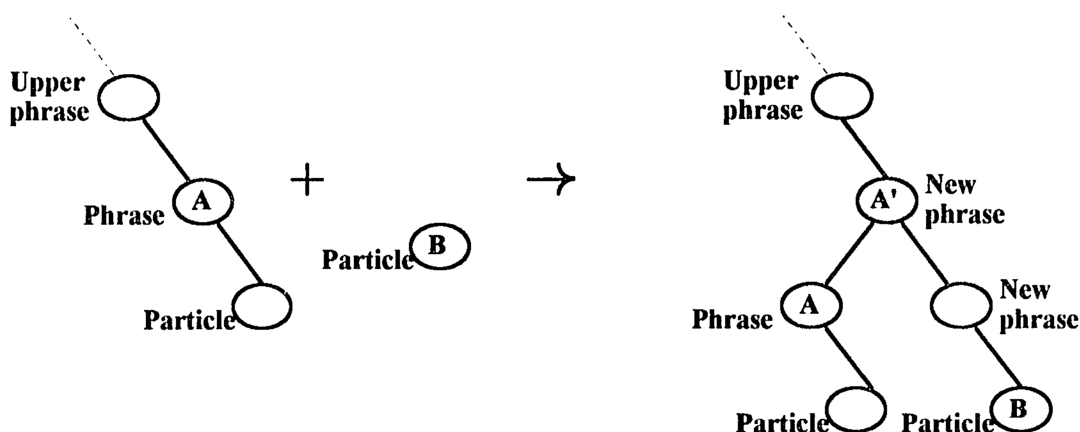


**start pop** is used to start a phrase that is not immediately connected to the tree. It starts a new phrase, closes the open one and looks for an instruction to associate with the new phrase.

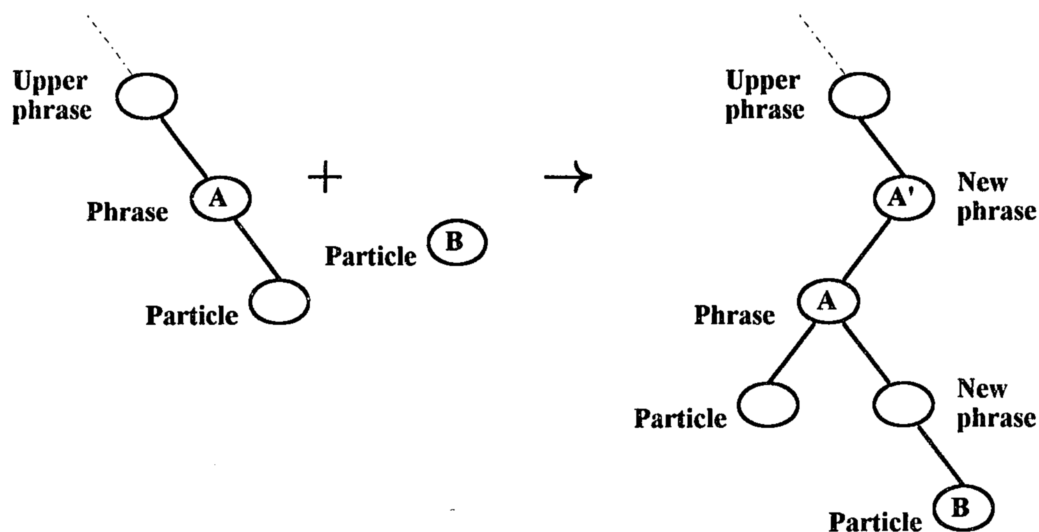



**enclose** is yet another operation that can be performed when begining a new phrase. It is used in conjunction with **start cont** or **start push**. After closing the phrase being constructed, the parser inserts a new phrase of the same type above it. It then performs the **start push** or **start cont** operations. This mechanism can be used, for example for a noun phrase followed by a few relative clauses. Each relative clause causes, in its turn, the creation of a new upper level of a noun phrase, that includes a noun phrase and a relative clause.

**start cont enclose:**



start push enclose:




When the current information is insufficient, a lookahead operation can be performed. It causes the parser to get and encode the next few words. Then, the parser returns to the same point in parsing, though richer in data.

adjoin tells the parser that the current word is a part of a bigger expression and should be attached to the previous word to form a particle.

record tells the system to perform the required dictionary operation and training for the current word, but skip the word as far as parsing is concerned. It can be useful, for example, to introduce a finite form of a verb, then use another form in a sentence and refer to the now familiar finite form in the dictionary instruction for the new word. A similar usage could help handling singular and plural forms of a word or variations of spelling.

fulllstop will finish the parsing of a sentence.

This is the general flow of the parser:

```
procedure parse(previous element tree position, current element tree position)

if there is no previous element
        point to current element from the sentence level

gething:
get the element in position (retain if there, read from input if not)
perform dictionary instructions for position
get the next parse instruction for the position (if any)

if instruction is record
        goto gething

if the element is a word and its lexical value is not given
        try to look the word up in the dictionary
        if the current element or the previous one is part of another phrase
                encode the phrase
                if the lexical value of the phrase is known
                        set input and output nets for lexical value
                        set input nets for phrase
                        train the network to output the lexical value
                else
                        set input nets for the phrase
                        run  the lexical value nets
                set lexical value for the position

if a parse instruction was given for the position
        set input nets for element and context
        set output nets for element's lexical value
        set output instruction nets
        train the network to output lexical value and instruction for the element
```

```
        get result
        if training was not successful
                if firstry
                        encode the next few elements
                        set nets for extended context including new elements
                        retrain the network to produce instruction
else (no instruction given)
        set input nets for element and context
        run output nodes to get instruction

if instruction is fullstop
  fullstop:
        print parse result
        goto parsed;

if instruction contains the enclose operator
        push the open phrase under a new phrase level (connected upward as
        the current open phrase was)

if instruction contains the start operator
        if it contains push
                begin a new phrase as next particle of the open one,
                start it with the current entity
        elsif the instruction contains cont
                close the open phrase,
                begin a new one as its follower
                start the new phrase with the current entity
        elsif the instruction contains pop
                close the open phrase,
                begin a new one, but do not connect it to the tree
                start the new phrase with the current entity
        if the instruction contains pop
                parse the newly started phrase in relation to
```

```
                the same reference point
        else
                parse the next particle in relation
                to the last particle in the lowest level of the partial tree

if instruction is pop
        reparse the particle in relation
        to a phrase one level up than the current reference point

if instruction is cont
  cont:
        set pointers to connect current element to the tree in same phrase level
        if element is pivot (which is induced or input together with the instruction)
                pivotize(upper phrase, element) (propagate information to phrase)
                parse(element,next element)(parse next particle in relation to current)
                goto parsed

if instruction is adjoin
        adjoin element to previous one
        goto parsed;

if instruction is lookahead
        read encode and temporarily atttach the next few elements
        goto gething (to get the next instruction for the same position)

If there is no (decisive) instruction
        if firstry
                lookahead and encode next few elements
                retry parsing
        else
                goto cont

parsed: end
```

## 5.5 A Rather Long Example of Parsing a Sentence

Let us now try parsing this:

The nice girl you like who likes the cute doggy who hates the nasty cat who drank the fat milk it liked ate the cake you liked.

We would parse it as:

```
::noun
___::noun
______::noun
_________The
_________nice
_________girl
______::relative
_________::noun
____________you
_________::verb
____________like
___::relative
______who
______::verb
_________likes
_________::noun
____________::noun
_______________the
_______________cute
_______________doggy
____________::relative
_______________who
_______________::verb
__________________hates
__________________::noun
_____________________::noun
________________________the
________________________nasty
________________________cat
_____________________::relative
________________________who
________________________::verb
___________________________drank
___________________________::noun
______________________________::noun
_________________________________the
_________________________________fat
_________________________________milk
______________________________::relative
_________________________________::noun
____________________________________it
_________________________________::verb
____________________________________liked
::verb
___ate
___::noun
______::noun
_________the
_________cake
______::relative
_________::noun
____________you
_________::verb
____________liked
```

To teach the system the above parsing, assuming that it is not familiar with the vocabulary, we can input the following text:

```
{::noun}
 {::noun}
  {::noun}
    The [article] (start,noun)
    nice [adjective sem(good)] (cont)
    girl [noun gend(anim female -male person) num(singular) sem(young)]
        (cont,pivot)
  {::relative}
   {::noun}
     you [pronoun noun gend(anim) num(singular plural)]
        (lookahead)(start pop pivot,noun)(start enclose cont,relative)
   {::verb}
     like [verb tran form(finite)]  (start cont pivot, verb)
{::relative}
   who [relative pronoun wh] [p,relative]
     (pop)(pop) (start cont pivot enclose, relative)
  {::verb}
    likes [var(like) form(s) tense(present) person(third) num(singular)]
         (start push pivot)
   {::noun}
    {::noun}
      the (start push)
      cute [adjective sem(good)] (cont)
      doggy [noun gend(anim) num(singular)] (cont pivot)
    {::relative}
      who [p,relative] (start cont enclose,relative)
     {::verb}
       hates [var(hate) form(s) tense(present) person(third) num(singular)]
            (start push,verb)
      {::noun}
       {::noun}
         the (start push)
```

```
          nasty [adjective sem(bad)] (cont)
          cat [noun gend(anim) num(singular)] (cont pivot)
        {::relative}
          who [p,relative] (start cont enclose pivot)
         {::verb}
          drank [var(drink) tense(past) form(irregular)] (start push pivot,verb)
          {::noun}
           {::noun}
             the (start push,noun)
             fat [adjective] (cont)
             milk [noun sem(food)] (cont)
           {::relative}
            {::noun}
              it [pronoun] (lookahead)
                (start pop pivot,noun)(start cont enclose,relative)
            {::verb}
              liked [var(like) tense(past) form(ed)]  (start cont pivot,verb)
{::verb}
  ate [var(eat) form(irregular) tense(past)]
     (pop)(pop)(pop)(pop)(pop)(pop)(pop)(pop)(pop)(pop)(pop)
     (start cont pivot, verb)
 {::noun}
  {::noun}
    the (start push, noun)
    cake [noun gend(inanim) sem(food)] (cont pivot)
  {::relative}
   {::noun}
     you (lookahead) (start pop pivot,noun)(start enclose cont,relative)
   {::verb}
     liked (start cont, verb)
. [punct fullstop] (fullstop)

{control:}
 The nice girl you like who likes the cute doggy
 who hates the nasty cat who drank the fat milk it liked ate the
 cake you liked.
```

Let us now follow the flow of some of the parsing of the example.

{::noun} Comment, skipped

{::noun} "

{::noun} "

The [article] (start,noun)

A dictionary entry is created for **the.** Its value is set to **article.**

The instruction node for **start** (top of diagram in 5.3) is set to 1, the other instruction nodes are set to 0.

One neuron input and output lexical value nodes are created for **article,** they are connected to lexical value input and output nodes and set to 1.

After **t**, **h** and **e** are encoded, the character nodes are set to their codes.

Input and output **capital** nodes are set to 1 to indicate capitalization of **T**.

The net is trained to output **start** and **article** in response to **the.**

A noun phrase is started in the parse tree, its first descendant is **the.**

nice [adjective sem(good)] (cont)

A dictionary entry is created for nice with the given value.

The net encodes the partial noun phrase created in the previous step:

The input nodes in the phrase subentry are set to the code of **the** which was created in the previous step, in the article node of the "**particles by speechpart**" subnet (in the phrase entry subnet) and in the entry for particle[-1] in the 'particles sequentially' subnet.

Input and output nodes in the lexical subnets are created for **noun.**

The noun nodes are set to 1. The article node previously created is set to 0 with a low tolerance and feedprobability.

The connection of phrase entry net to dictionary is set via the "**phrase entry**" node.

The net is trained to output **noun** in response to the partial phrase.

The instruction node is set to **cont.**

Input and output speech part and qualifier nodes are created for **adjective** and **sem.good.** They are connected to lexical value input and output nodes and set to 1.

The character nodes are set to (codes of) **n**, **i**, **c**, and **e**.

The input and output **capital** nodes are set to 0.

The node for article is set to 0, and to low tolerance and feedprobability

The code of **upper_phrase[0]** is set to the activation that was taken from the lexical value subnet after the partial phrase encoding performed in the previous step.

The phrase entry nodes are set as in the phrase encoding operation, but the connection is now set to be via the **context** connection node.

The net is trained to output **cont** and the given lexical value for **nice.**

girl [noun gend(anim female -male person) num(singular)sem(young)]

(cont, pivot)

Lexical nodes are created for **gend.anim, gend.female** etc.

The nodes for **gend.male** are set to 0 with normal learning parameters.

The partial phrase window now includes the previous element **nice.**

The lexical value of the partial noun phrase is enhanced by the lexical value of **girl** because **girl** is the pivot of the phrase.

The net is trained for the new lexical value and instruction, then **girl** is added to the current phrase (the noun phrase).

{ ::relative}

{::noun}

you [pronoun noun gend(anim) num (singular plural)]

(lookahead)(start pop pivot,noun)(start enclose cont,relative)

**you** has both the qualities of **pronoun** and **noun** as parts of speech and both the qualities of **singular** and **plural** for number.

Three parsing instructions are given here consecutively

As **you** by itself is an insufficient clue to determine what operation to take next, the parser needs to perform a **lookahead**; it can then decide that **you** starts a new noun phrase, one which is not immediatly connected to the partial parse tree. This is specified by (start pop pivot, noun).

Next, the parser can decide what to do with the newly created noun phrase: it performs the third directive, (start enclose cont, relative)

This creates a relative phrase of which the first particle is the new noun phrase (**you**). The relative clause is connected to the tree at the same level of the previous noun phrase (**the nice girl**), then they are both subordinated to a new upper noun phrase.

This was just the flow of parsing. To be able to use the net to decide the instructions, it must be trained to issue them. Each of the above steps contained a stage of encoding the new words and the new phrases, resetting the net windows to the tree with the codes and training it to issue the given instruction and phrase lexical value for each of the situations.

{::verb}

like [verb tran form(finite)] (pop push pivot)

The relative clause includes two subphrases, the second of which is this verb phrase. The verb phrase is at the same level of the noun phrase preceding it.

{::relative}

who [relative pronoun wh] [p,relative]

(pop)(pop)(start cont pivot enclose, relative)

As opposed to the previous relative clause, this one has a head- **who,** which makes it easier to identify. There are still three parsing operations to perform here. First we (train the net to) discover that **who** is not part of the verb phrase; we therefore (train the net to issue and) perform a **pop** operation. There is one more **pop** to exit the relative phrase above the verb phrase.

Next, we leave the high level noun phrase to which the previous relative phrase belongs, to create a new relative clause for it in its level and enclose both in a new higher level noun phrase.

Note the multiple dictionary directives for **who.** The first directive establishes a full definition for **who** as both a relative pronoun and a wh question word, whereas the second directive picks the usage of a relative (pronoun) for the purpose of the current parsing. As this is the head of the phrase, the lexical information is passed up, to the phrase level.

{::verb}

likes [var(like) form(s) tense(present) person(third) num(singular)]

(start push pivot)

Note the **var(like)** in the lexical value, meaning a variation of **like**. The system will use the lexical value of **like** to set the nets, with changes for the **form, tense person** and **num** qualifiers. The verb phrase is to be part of the relative phrase. It gets the **verb** qualityfrom its head or pivot - likes.

{::noun}

{::noun}

the (start push, noun)

The noun phrase here is part of the (preceding) verb phrase in the relative clause.

cute [adjective sem(good)] (cont)

doggy [noun gend(anim) num(singular)] (cont pivot)

{::relative}

A relative phrase will follow the noun phrase within a new noun phrase within the verb phrase (within the relative clause)

who [p,relative] (start cont enclose, relative)

{::verb}

hate [verb tran form(finite)]
(record)

hates [var(hate) form(s) tense(present) person(third) num(singular)]
(start push pivot, verb)

The first **hate** is to teach the system the basic word. **hate** is entered in the dictionary and the net is trained to encode it. Then **hates** is entered as a variation of **hate** and is used for construction of the phrase.

{::noun}

.

.

.

{::verb}

ate [var eat form(irregular) tense(past)]

(pop)(pop)(pop)(pop)(pop)(pop)(pop)(pop)(pop)(pop)(pop)

(start cont pivot, verb)

To pop all the way to the upper level, we need eleven passes of parsing, after which **ate** is parsed against the (highest level) noun phrase enclosing **girl** to start a new verb phrase at the same level.

{::noun}

the

cake [noun gend(inanim) sem(food)] (cont pivot)

{::relative}

{::noun}

you (push, relative) (start push pivot)

{::verb}

liked (start cont pivot)

. [punct fullstop] (fullstop)

A dot is a dictionary entry with punct and **fullstop** as parts of speech, so that the net can learn the fullstop situation rather than having the parser enforce an end of sentence .

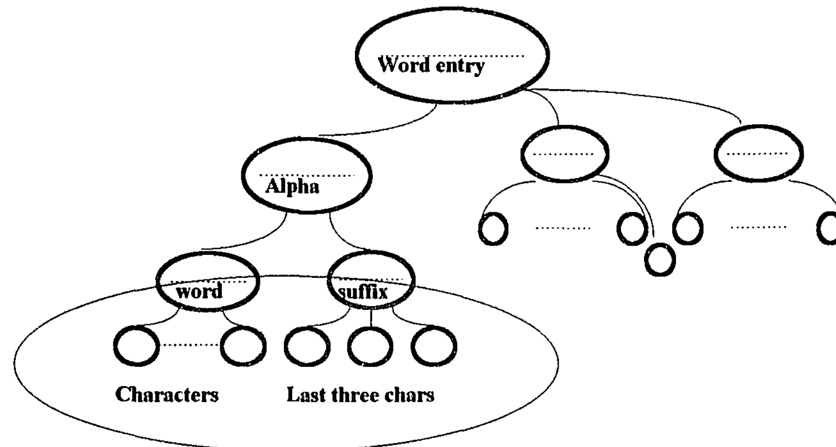
{control:}

Now let the parser try it all by itself ...

The nice girl you like who likes the cute doggy who hates the nasty cat who drank the fat milk it liked ate the cake you liked.

# 6. An Example of a $C^{+NET}$ Procedure in the Parser

The following $C^{+NET}$ procedure is used to input a coding of a word to the language comprehension net

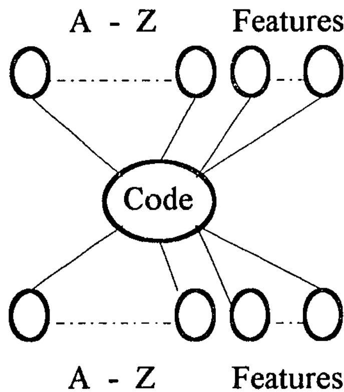



The procedure first creates the characters nodes. The characters of input words are first encoded, then the codes are used as input to the character nodes.

```
void endictword(words word)
 {
  static int firstime on;

  int i,j,k;
  int len;
  netname chari,ia;
  char c;
  codes ccode;
  netsizes sizeword = {wordclen,0};          size of word node
  netparms parmword = {0.7,.8,.0,.8,0.9};    parameters of word node
  netsizes sizesfx = {wordclen/2,0};         size of suffix node
  netparms parmsfx = {0.7,.8,.3,.8,0.9};     parameters of suffix node
  netsizes sizechar = {charclen,0};          size of character node
  netparms parmchar = {0.4,.4,.3,.9,0.9};    parameters of a character node
  netlists charinets;                        list of netnames for input characters

  pushdinfo("endictword ",word,_at);                push information to debug stack
```

```
if (firstime)
 {
  firstime off;                                  Create second level nodes
  CreateNeuNet("wordcodein",parmword,sizeword);
  CreateNeuNet("sfx",parmsfx,sizesfx);
  connets("wordcodein sfx","diclow");         connect them to higher level
  charinets=nullist;
  for (i=0;i<=maxwlen;i++)
   {
      strcpy(chari,"charin");                    create character nodes
      strcat(chari,itoa(i,ia,10));
      CreateNeuNet(chari,parmchar,sizechar);
      charinets=li_ charinets,chari _st;       List them
      connets(chari,"wordcodein");               and connect them
   }
  CreateNeuNets("charsfx charsfy charsfz",parmchar,sizechar);
  connets("charsfx charsfy charsfz","sfx"); same for characters of suffix
  charinets=li_ charinets,"charsfx charsfy charsfz" _st;
 }

len = strlen(word);
if (len > maxwlen)
   len = maxwlen;

 }
ignore(charinets);                               Only assigned character nodes will be active

for (i=0;i<=len;i++)
 {
  c = word[i];
  if (i==len)
       c='\0';
  endictchar(c,&ccode);                          Create a code for each character up to length+1

  strcpy(chari,"charin");
        strcat(chari,itoa(i,ia,10));
  setinpat_func(chari,fromPatterns,genfunc,outward,1,0,0,ccode);
                                                 the code will be input via a function
  if     (i==len-3)
      setinpat_func("charsfx",fromPatterns,genfunc,outward,1,0,0,ccode);
  else if (i==len-2)
      setinpat_func("charsfy",fromPatterns,genfunc,outward,1,0,0,ccode);
  else if (i==len-1)
      setinpat_func("charsfz",fromPatterns,genfunc,outward,1,0,0,ccode);
 }
remove("chars.knw");                             Store subnet and weights to a file
storenets("chars.knw","featsin charcode featsout");
popdinfo();
}
```

```
int endictchar(char c, codes *code)
```




The procedure accepts an ASCII character and returns a concise representation for it created by a dedicated net.
The net creates an internal representation that selects a character out of a set and incorporates some features of the character

```
{
  static int firstime on;
  int i,j,k;
  codes ccode;
  int cycles;
  netsizes sizefeat = {codeflen,0};
  netparms parmfeat = {0.9,1,0.,1,0.9};
   netsizes sizechar = {charclen,0};
   netparms parmchar = {0.4,.4,.3,.9,0.9};

  for (i=0;i<charclen;i++) (*code)[i] = 0;
  if (c == '\0') return (0);
  if (firstime)
   {
    firstime off;
    CreateNeuNet("charcode",parmchar,sizechar);          Create the net in the diagram
    CreateNeuNet("featsin",parmfeat,sizefeat);
    connets("featsin","charcode");
    CreateNeuNet("featsout",parmfeat,sizefeat);
    connets("charcode","featsout");
    stmize("featsout","featsin", chartlen, GlobaLearn);  Create an stm for the net
   }                                                     all characters will be in it

  asci2feat(c,&ccode);                                   Translate ASCII character into a
                                                         sequence of neurons, see below
  setinpat_func("featsin",fromPatterns,genfunc,outward,1,0,0,ccode);
  setinpat("charcode",fromConnections);                  set input and patterns from
  setinpat_func("featsout",fromConnections,genfunc,outward,1,0,0,ccode);
                                                         functions

  cycles=learn ("featsout",GlobaLearn);                  Train net to reproduce its input

  getact(*code,"charcode");                              Use internal representaion as
                                                         code of character
  return(cycles);
 }
```

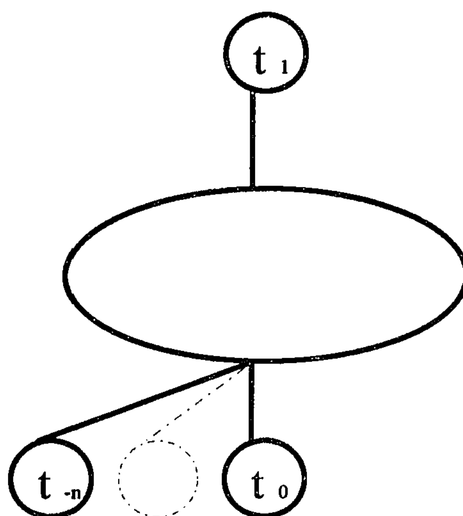
The following procedure translates an ASCII character into a sequence of floating point numbers, which are activations of neurons. For a-z neuron 0-25, correspondingly, is set to one. The rest of the neurons represent features such as vowels and special characters.

```
# define chartlen 26
# define numfeatures 2
# define codeflen (chartlen+numfeatures)
char chartable[chartlen+1] = "abcdefghijklmnopqrstuvwxyz";
int asci2feat(char c, codes *code)
 {
  int i,j;
  static char feature[numfeatures][9]
   =  {"aeiouy",",;?!.-"};

  if (isupper(c))
    c = tolower(c);

  for (i=chartlen,j=0;i<codeflen;i++,j++)
        if (strchr(feature[j],c))
   (*code)[i]=1.0;
    else
   (*code)[i]=0.0;

  for (i=0;i<chartlen;i++)
     if (c == chartable[i])
            (*code)[i] = 1.0;
          else
            (*code)[i] = 0.0;
  return (0);
 }
```

## 7. An Outline of a Model for Financial (or Other) Forecasting

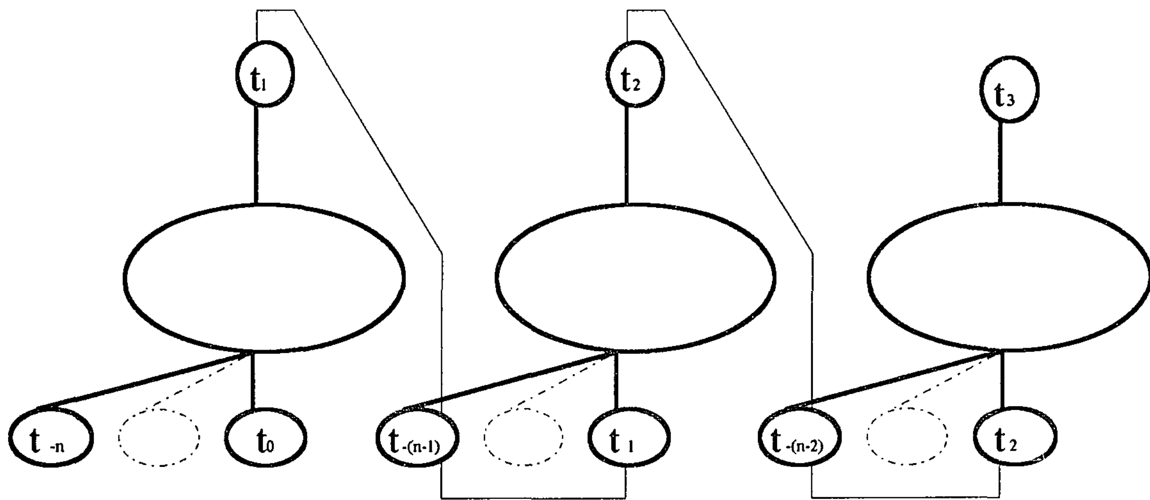
The goal is to forecast behavior of components of the financial market. With the modular approach, prediction of various factors of the market can be used to help forecast interrelated components. Currently, though, we are discussing just one given factor, say interest rates, or the relative value of a given financial instrument, say the value of the US dollar against the shekel. The model presented here could in fact be translated to an arbitrary forecasting problem of a time series nature. It may be appropriate for weather with a change of interpretation.

We start with the known [31] "window" design. A net is trained with series of values taken at a fixed interval. It learns to make a prediction for the next value in a series, as illustrated below.




To predict next values, for $t_2$ and on, the net can be reapplied to the series enhanced, or suffixed, by the value predicted as the next value of the current series, and this can be done repeatedly as needed. The forecasted values are therefore considered the actual next values in the series, and the window is slid

forward one step with each reactivation of the net to produce the next value in the series:



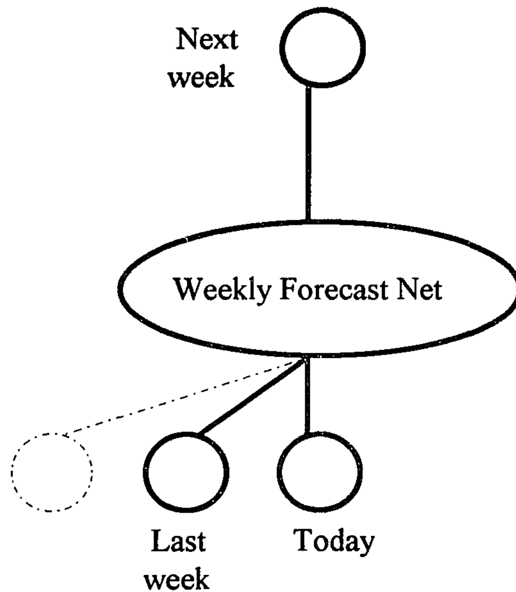

For the value of $t_3$, the net is activated once to obtain a value for $t_1$, then again with $t_1$ as the new $t_0$ and the window moved one step forward to obtain a value for $t_2$, and then again with $t_2$ as the new $t_0$ to obtain $t_3$.

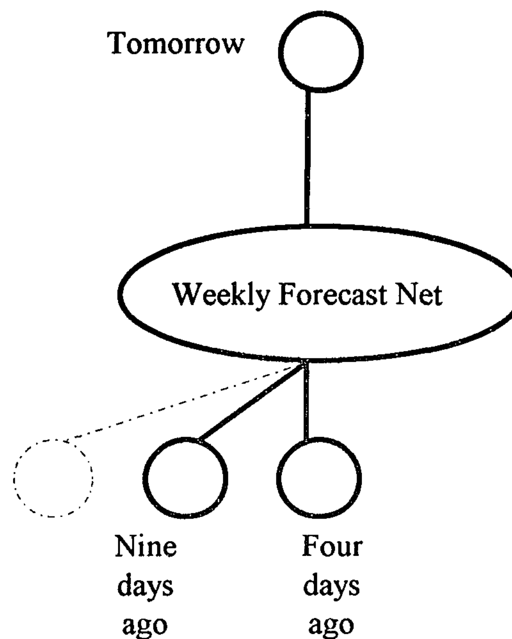
We think the above is a nice start, but it does not appear to be enough to capture the "chaotic" aspect of the market behavior. Further, we think we can progress toward capturing more of the regularities of the market by some considerable elaboration enabled by net programs.

Our suggestion is to link the operation of several windows, functioning at different "resolutions" or sampling intervals.

The basic elements of our model will look like the net described above. We will have one for a daily series, one for a weekly series, and one for a monthly series. Day, week and month are, for the purpose of the current discusion, arbitrary periods of times corresponding to the interpretaion. For financial markets, we would choose the daily value as the value at a given exchange at closing; we shall make the week five business days and the month four weeks. Thus, the weekly net

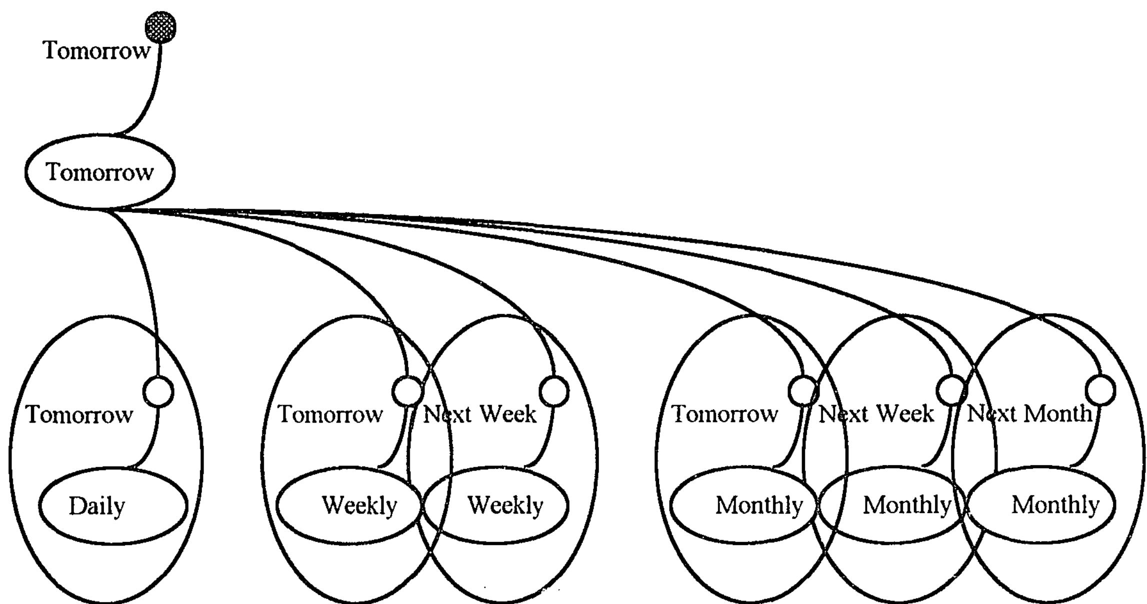
does not forecast the value at the end of a week but the value five days ahead of a given day. It is trained daily, and can, therefore, forecast values of up to five days by sliding the data:




Similarly the monthly net can forecast values of any day up to one month from the last observed value before starting to reuse predicted values.

Now we can put the various predictions of the three basic nets to work in order to predict tomorrow's value. We will consider the daily, weekly and monthly nets' predictions for tomorrow, the weekly and monthly nets' predictions for next week, and the monthly net's prediction for next month. All of these forecasted values, will be input to a net we call **tomorrow**, which arbitrates among the various tendencies detected - by short and long term forecasts for tomorrow and by short and long term forecasts for the farther future.




The **tomorrow** net is connected to procedures rather than net nodes, because it needs to get data from different activations with different data of the *same* three nets, which necessitates the intervention of procedures.

But as usual, we want more. We want to know what will happen beyond tomorrow. For the next few days' forecast we can use the window strategy, applied to the whole construction. To slide the window one day, we will obtain tomorrow's

value from the **tomorow** net, and then use it as input to the daily net, while shifting the observed data used for the weekly and monthly nets.

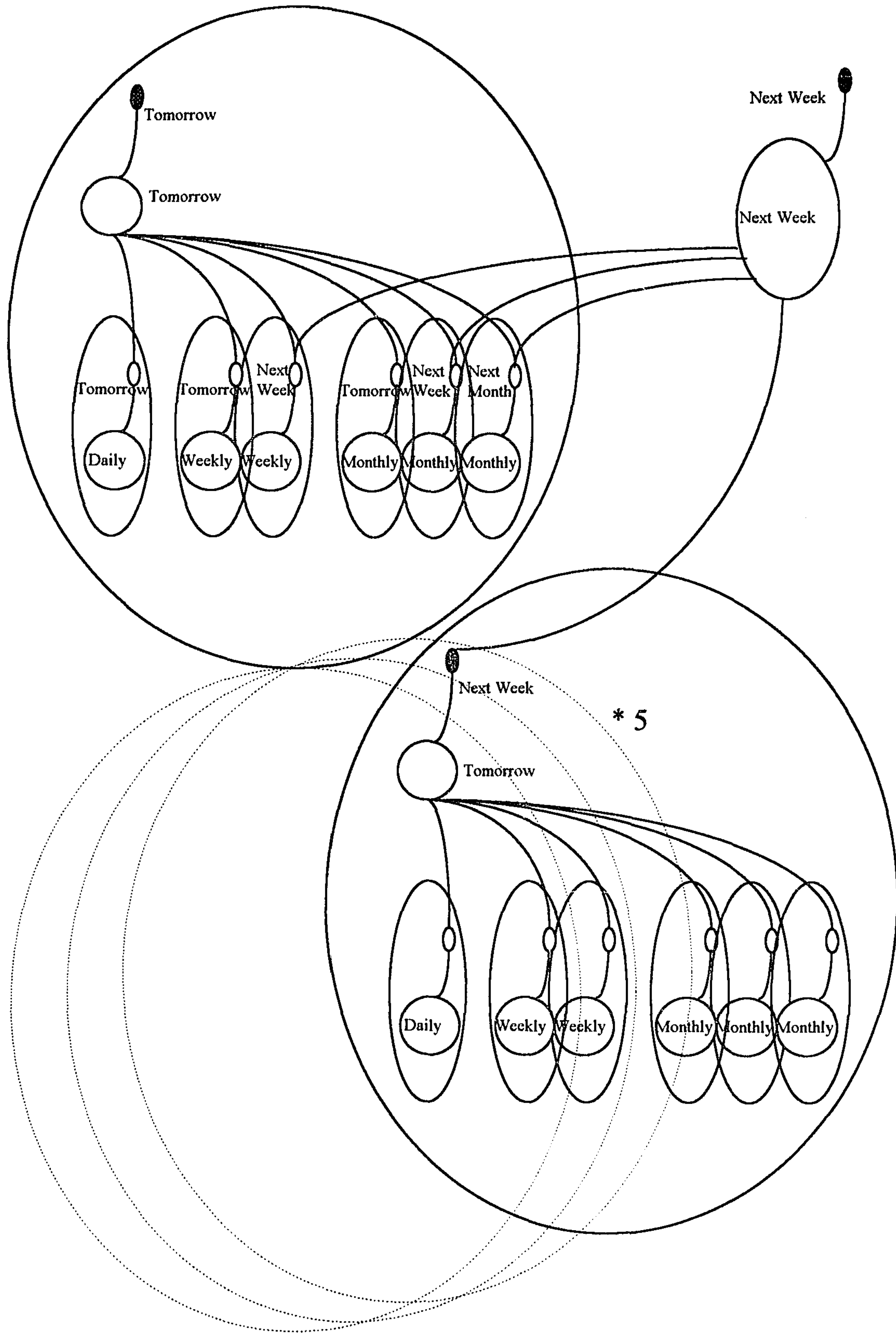
Is this mechanism good for the more remote future? The sliding window strategy has been tried (for a conventional net) and it performs reasonably well [31]. We think we can do better by extending our idea of combining short, (various) intermediate and long term forecasting.

Particularly, combining short and long term forecasts could be useful to predict turning points which may emerge from an incongruity between short and long term trends.

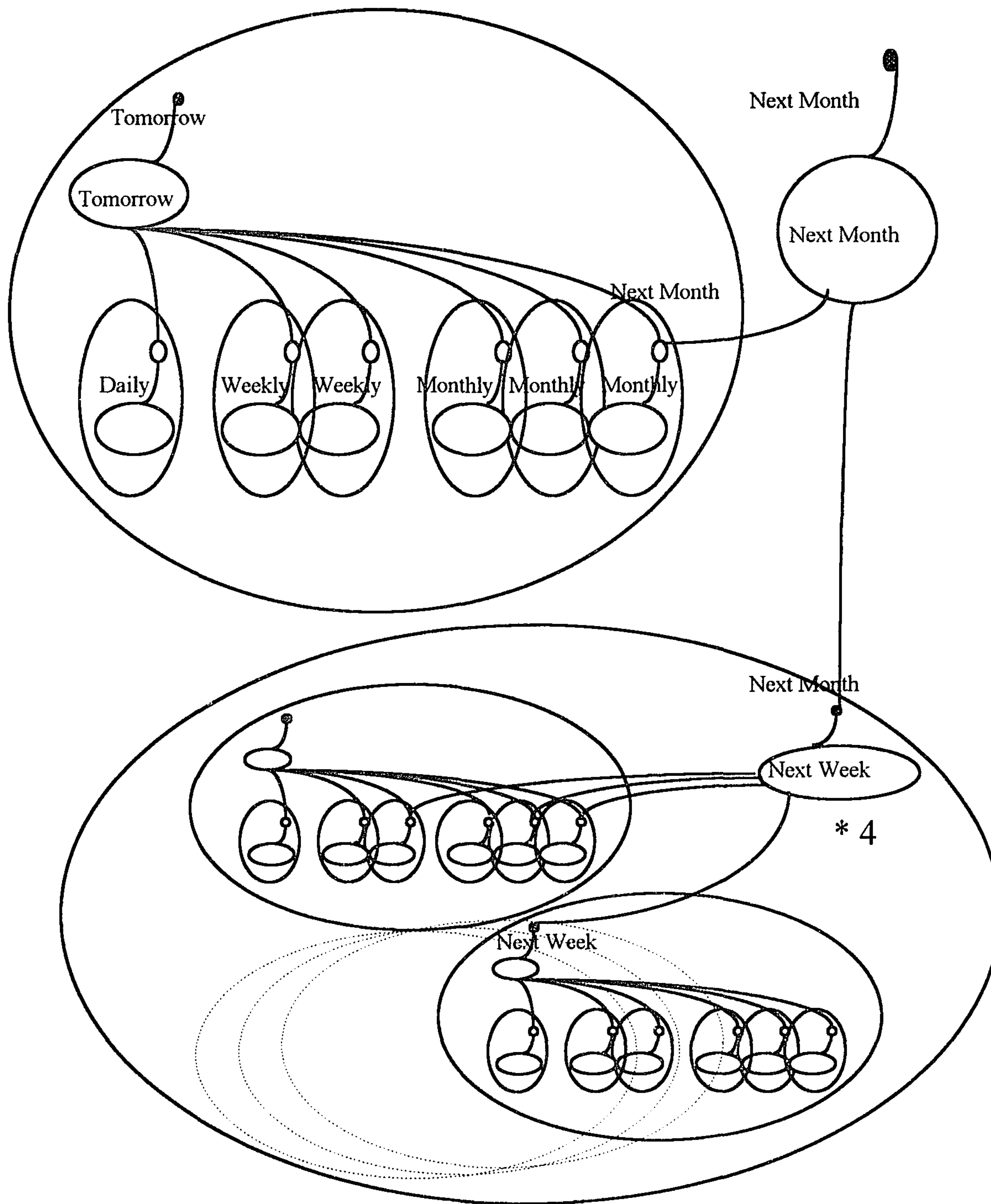
This is where the advantage of the *neural programming* approach becomes more apparent. The model becomes more and more difficult to construct by direct mathematical methods, and neural nets can be the salvation, but for an elaborate task they, too, need a degree of elaboration of structure and operation.

We will improve upon our model with the help of some modular additions. To predict a value a week from now, we want to use the value forecasted by the existing system for next week, the value forecasted by the weekly subnet for next week, and the value forecasted by the monthly subnet for next week, and the value forecasted by the monthly subnet for next month.

We will operate the existing system in two ways: once with today's data to achieve tomorrow, next week and next month forecasts by the daily, weekly and monthly subnets, respectively, and then we want to obtain the extended forecast from the existing system by repeatedly sliding the window until reaching a week ahead. Then we can arbitrate between the various forecasts by adding a **next week** arbitrator net. This is illustrated in the following figure.


Tomorrow
Tomorrow
Next Week
Next Week
Tomorrow
Daily
Tomorrow
Weekly
Next
Week
Weekly
Tomorrow
Monthly
Next
Week
Monthly
Next
Month
Monthly
Next Week
* 5
Tomorrow
Daily
Weekly
Weekly
Monthly
Monthly
Monthly

Similarly, next month's forecast is obtained by considering the forecast of the elementary monthly net with the extended forecast of the **next week** net operating upon its own forecast four times:



To draw a graph of the values forecasted for the foreseeable future we will do the following:

For points of time less than a week ahead:

apply the **tomorrow** net to observed values and reapply to forecasted values as many times as needed.

For one week:

apply the **next week** net.

For more than a week

apply and reapply the **next week** net as many times as needed to get to the beginning of the desired week, then apply the **tomorrow** net as needed.

For one month

apply the **next month** net.

For more than a month

apply the **next month** net as needed to get to beginning of desired month, apply the **next week** net to get to the beginning of desired week, apply the **tomorrow** net as needed.

# 8. A Net Program for Pattern Recognition

## 8.1 Overview

The goal is to lay a basis for some pattern recognition solutions and to make a contribution toward a mechanism to identify and label objects in an image. We will construct a digit recognizer in a manner general enough for it to be reused and expanded for other domains.

A neural program can be appropriate for the task because:

As a neural net:

- It can avoid specialized complex time consuming mathematical models
- It can adapt itself and learn from examples

As a program:

- It can use the serial speed of the electronic computer
- It can use the procedural and symbolic knowledge we have of the domain

And as a Net Program:

- It can utilize nets and procedures in a highly modular way to save time and space
- It can separate the input and output media used from the problem by modularizing the interface between representation and solution
- It can isolate the various tasks that are either common or specific for various pattern recognition tasks and reuse solutions for common tasks.
- It can accommodate for and use actual models of pattern recognition. It can implement such models, or borrow from both natural (biological) and artificial (algorithmic) solutions, as would make sense when solving a natural problem artificially.

The method which we chose to address the pattern recognition problem may look, and perhaps is, quite simple. This may serve to testify to the appropriateness of the Net Program methodology for problems of this class. Generally it is as follows:

i. Turn a given representation of an image into a media independent set of pixels of a parameterized resolution and range.

ii. Turn the image from a set of pixels to a set of elemental features. The feature detector is a Net Procedure that is reused over the image. This stage reduces the size of the problem considerably, while discarding noise and maintaining meaningful information. Trainable neural nets are used on a large image but they are activated on one small problem at a time. The set of features need not be fixed, and can be recreated and retrained to achieve optimal results.

iii. Recursively recreate the image in terms of higher level features/figures/objects. The characteristics of the various "figure detector" levels are parameterized.

We define two types of image reproducing levels:

Scaling levels

Decision levels

They both use figure detector (nets), but a scaling level maintains the figures detected by its next lower level. It scales the current image down to the size that can be operated upon by the next higher decision level. The scalers detect larger features of the same kind of which the image consist and represent them to the next higher level. They can operate recursively.

A decision level turns sets of figures in an image into a set of higher level figures.

There is a basic similarity among the tools that we defined. All of them, starting at the pixel level, and ending at the object level (and may be going on to complex objects) are detectors. They are *net procedures* that are parameterized with the size and type of their input and are trained and operate in their specified level. They are used over the image as if they were an array of detectors by presenting them with a partial view of the image at a time.

At the pixel level the input figures can be defined to be the "blackness", "redness," "blueness" etc. of a region in the physical image.

At the elementary feature level the input is a set of pixels of a variable size.

At the next levels the input is a set of features, either from previous levels or in the case of recursive scaling, from the same level.

Because of the similarities, we chose to refer to the various operations of dealing with pixels, features and objects as various level of *figurette detection.*

### 8.2 Details of an application to Digit Recognition.

A net program has been devised to run under Microsoft Windows using its interactive graphic interface.

A trainer can define, parameterize and train the "figurette detectors" of the various levels. For the digit recognition application, we define the highest level to be that of digits.

As it starts, the program loads a definition file. The file defines the figurettes and their levels, by enumerating and naming the levels, their respective sizes and the figurettes they can detect. The definition file for this application is:

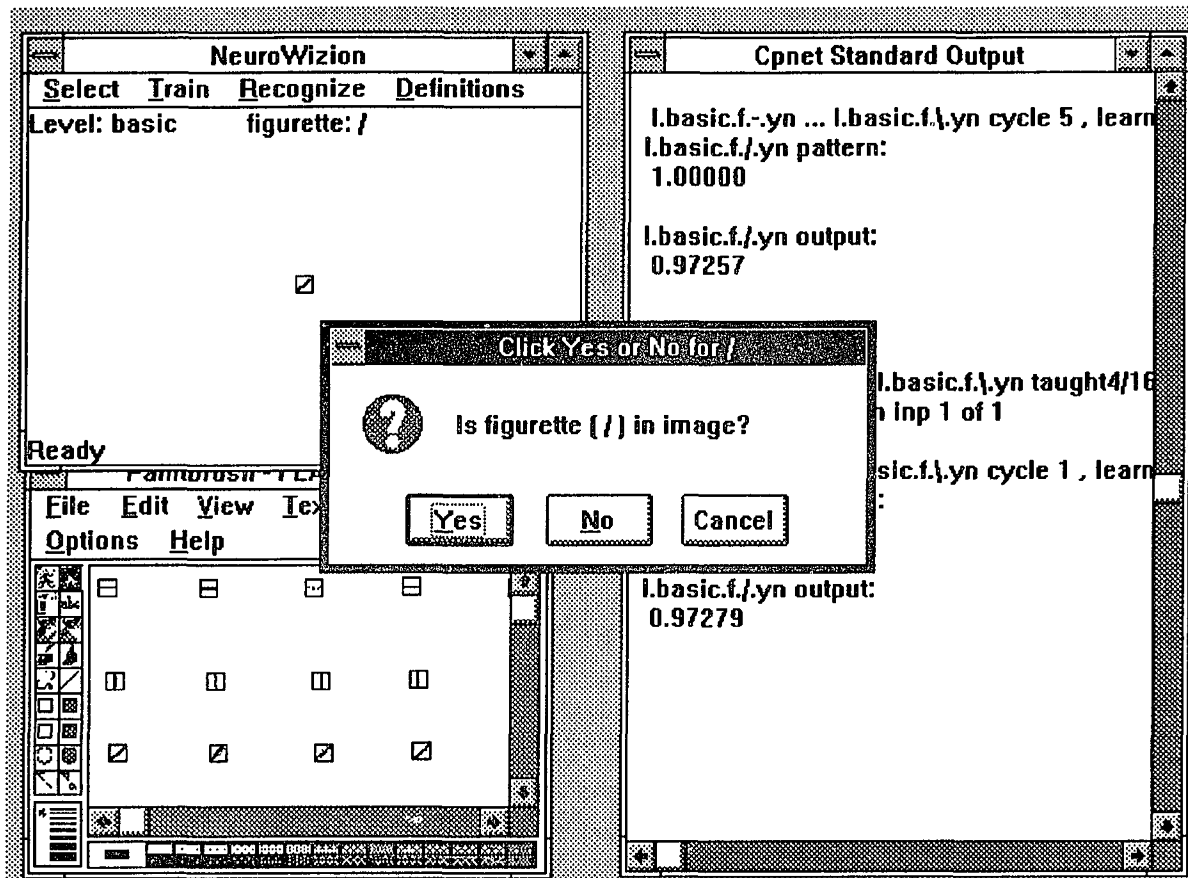
```
level pixel size .25
  black dot
level basic size 2.0
  feature -
  feature |
  feature /
  feature \
level digit size 12.0
  digit 0
  digit 1
  digit 2
  digit 3
  digit 4
  digit 5
  digit 6
  digit 7
  digit 8
  digit 9
```

The above definitions tell the recognizer that the most elementary level of picture elements is of black pixels of size .25 mm. At the next level are features of size 2mm. which are composed of pixels. Their size in pixels is implied to be 8×8. The next level is named digit and is composed of ten elements of size 12mm. Other elements can be added later while preserving the training result for the existing ones.

The program creates the supportive nets for the features at their levels and builds corresponding menus for user interface.

For the training session, the trainer uses a four-way windowed screen. The top left quarter is the main user interface window. The bottom left window is that of a drawing application. The one we use is MS-Windows Paintbrush. The right half of the window is where the application program can display standard output in an old fashioned scrolled manner. This is for debugging and reporting, not for interacting with the user. This output is also saved on an accumulative file on disk.

This is how the screen looks when training the figurette named "/" in the level named "basic":

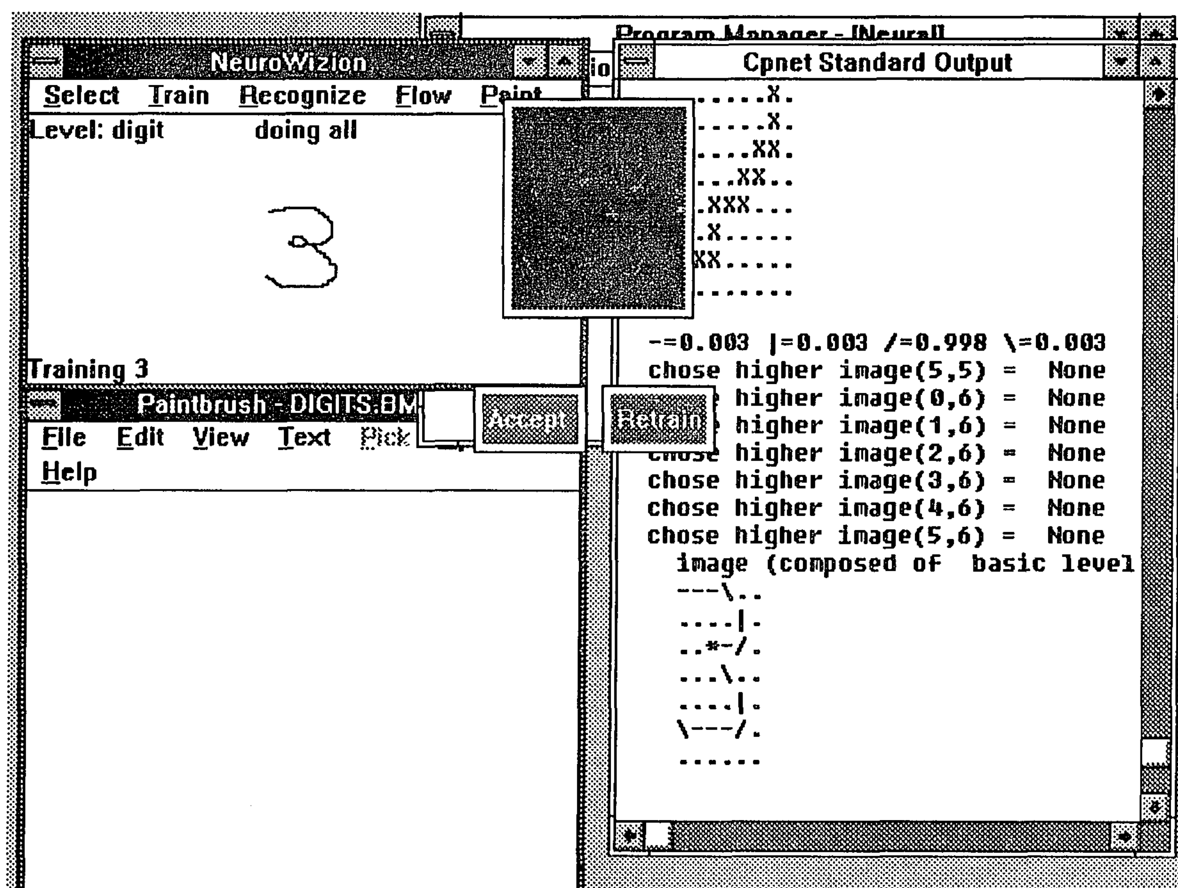



Now the trainer can select and train entities. The images are clipped out of the drawing window. The figurettes at a given level can be trained together or separately. The trainer can choose to train one figurette, and then for each presented image, click Yes or No for whether it is a positive or negative example. Alternatively, he or she can select the figurette name that a given image exemplifies and then along with training the corresponding net, the nets for the other figurettes at the same level are trained not to respond to the given image.

When trained together, all nets participate in the learning. When trained separately, the nets trained are the ones shared by same level figurettes and the one for the trained figurette.

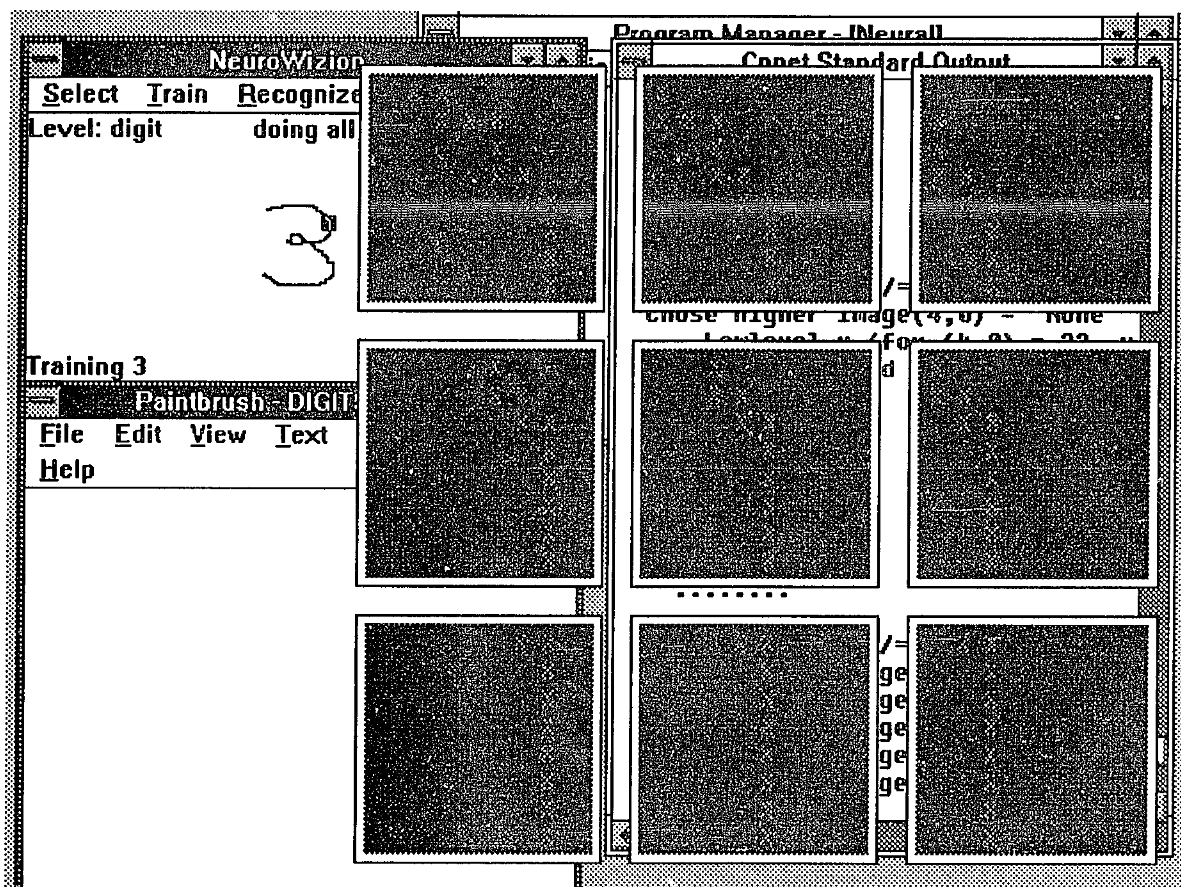
When training the digit level figurettes, an image is first constructed with basic level features. The transformed image is presented to the trainer. If satisfied, the digit level nets are trained to recognize it, otherwise the image is reconstructed interactively, enabling the trainer to retrain for basic features where necessary.

This is how the screen looks after creating a featured image:



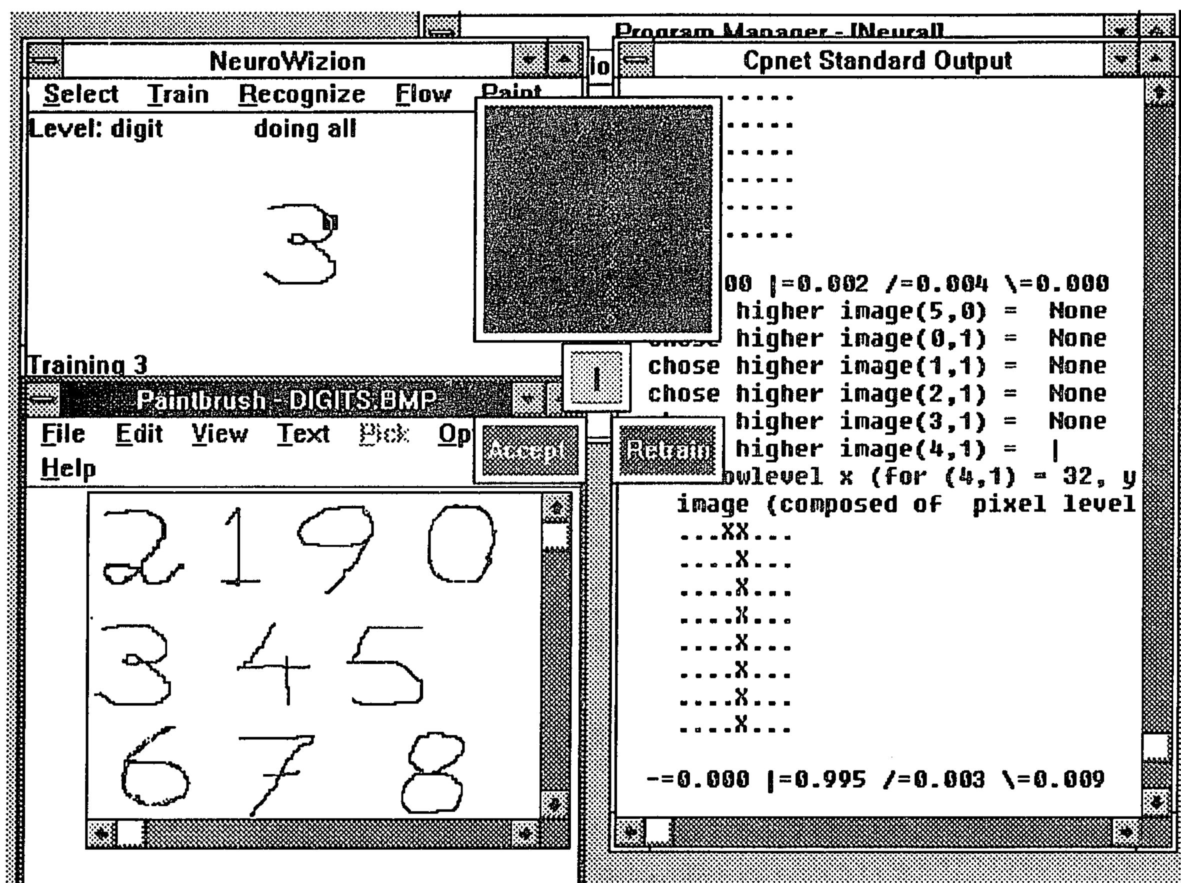
If retaining, the program will scan the image, presenting the trainer with choices regarding each area in the image. The first choice to make is what exact feature pattern in the neighborhood of the square in question should be responded to.

The pattern is chosen using this screen:




The user clicks on the square that best represents the relevant area (highlighted in the original image).

When a choice is made, the program presents the basic level feature it recognizes in the given square, and permits changing it. It uses the following screen:




If the trainer wishes to change the recognizer's interpretation of the pattern, he can chose "retrain". In this case, the training of digits is suspended, and the basic level nets are retrained with the new choice. Thus, all levels of figurettes can be trained at the same time and basic level features are trained by their usage within higher level figurettes - digits.

The recognizer can be trained to recognize all ten digits in one session. Let us now follow some of this training. We will train for two samples of ***5***, one sample of ***8***, and then try the system on a new ***5*** and a new ***8***.

Training for *5*:

The recognizer suggests a reconstruction of *5* out of lower level figurettes, in the center square. We choose not to accept it, and as shown before for *3*, we have it rescan the image with manual interference so as to recontstruct the image.

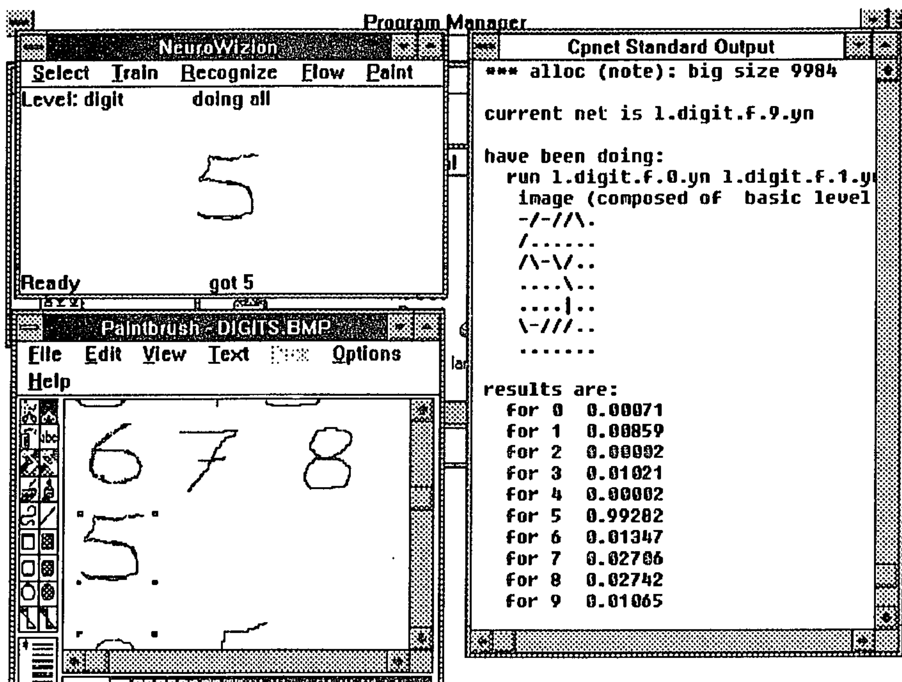
We will train for a second sample of *5*:

```
Program Manager
NeuroWizion
Select  Train  Recognize  Flow  Paint
Level: digit        doing all
Training 5
Paintbrush - DIGITS.BMP
File  Edit  View  Text  Pick  Op
Help
Accept  Retrain

Cpnet Standard Output
.501 |=0.010 /=0.349 \=0.028
se higher image(3,6) =  /
 Lowlevel x (for (3,6) = 24, y
nage (composed of  pixel level
.....X.
....XX.
.XXXX..
XX.....
X.......
........
  .....
  .....
 -=0.225 |=0.064 /=0.779 \=0.021
 chose higher image(4,6) =  None
 chose higher image(5,6) =  None
   image (composed of  basic level
   .--//.
   /.....
   .*-...
   ...\..
   ....|.
   \../..
   \--/..
```

And for an *8*:

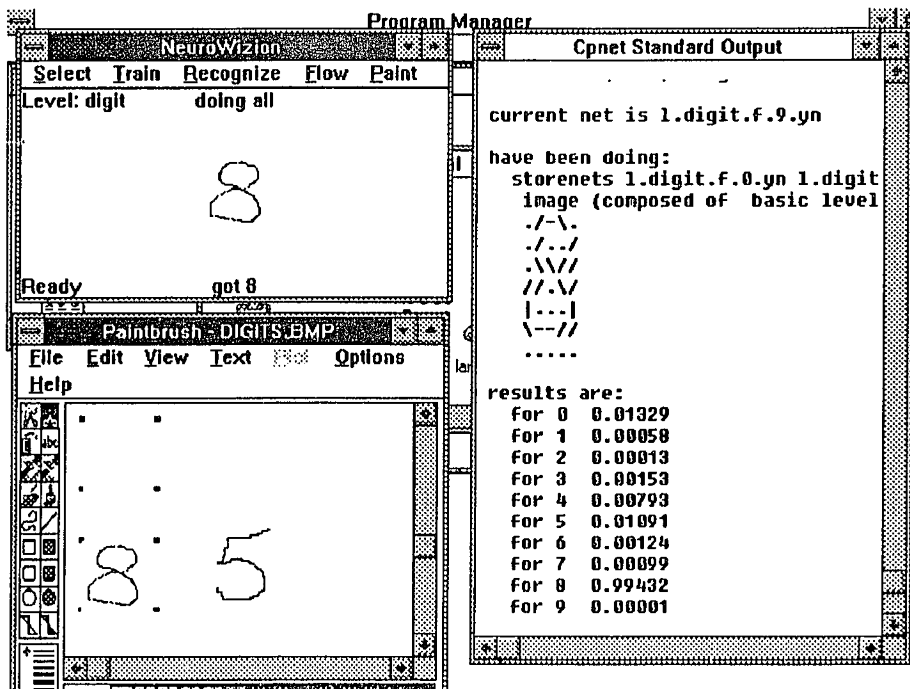
```
Program Manager
NeuroWizion
Select  Train  Recognize  Flow  Paint
Level: digit        doing all
Training 8
Paintbrush - DIGITS.BMP
File  Edit  View  Text  Pick  Opt
Help
Accept  Retrain

Cpnet Standard Output
image (composed of  pixel level
......X.
......X.
.....XX.
.....X..
.....X..
....XX..
...XX...
XXXX....
 -=0.002 |=0.038 /=0.968 \=0.009
     higher image(4,5) =  None
     higher image(0,6) =  None
     higher image(1,6) =  None
 chose higher image(2,6) =  None
 chose higher image(3,6) =  None
 chose higher image(4,6) =  None
   image (composed of  basic level
   .---.
   /...|
   \../.
   /*-\.
   |...|
   \--/.
   .....
```

Let us now try recognizing a new *5*:



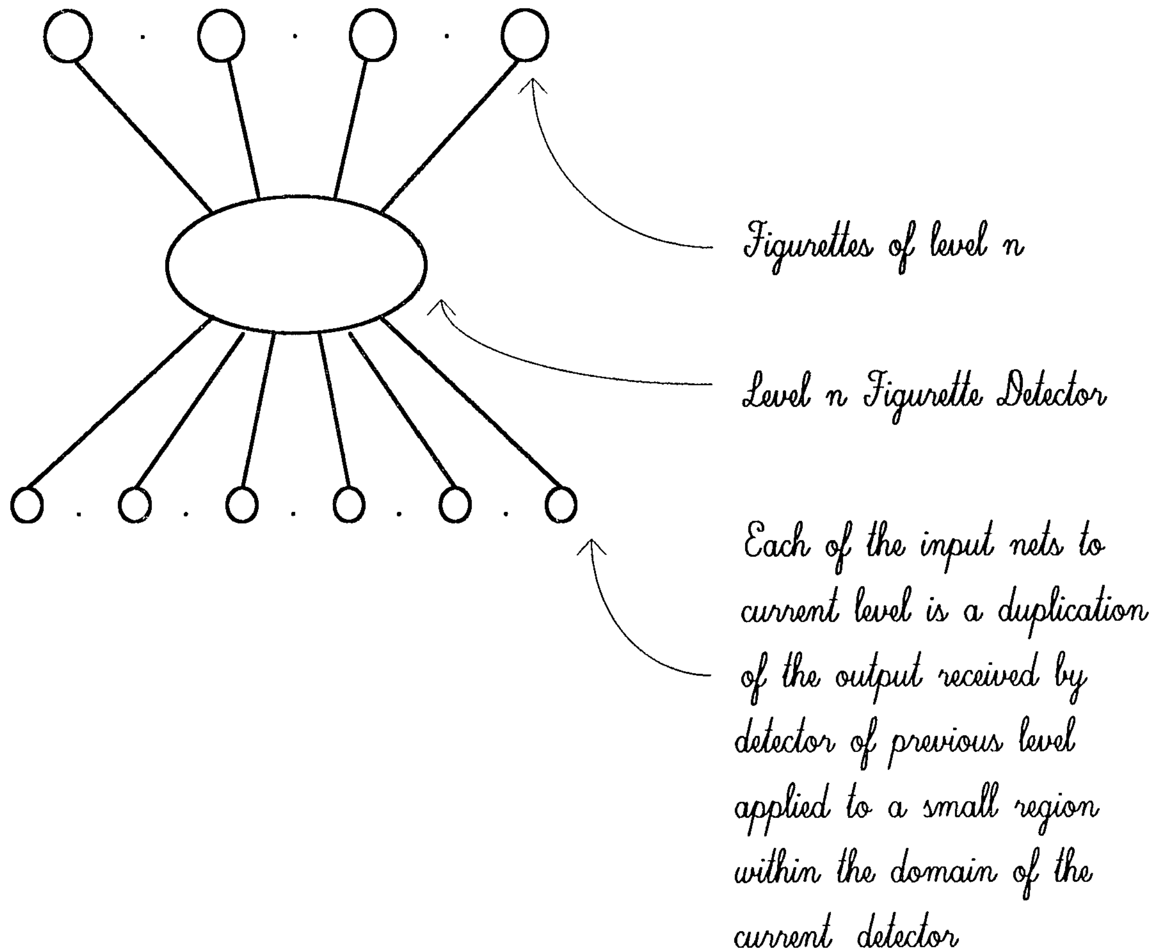

The output (bottom right) is.99 for *5*, and between 0 and .03 for the other digits.

Let us try *8*:




The output is .99 for *8*, and between 0 and .01 for the other digits.

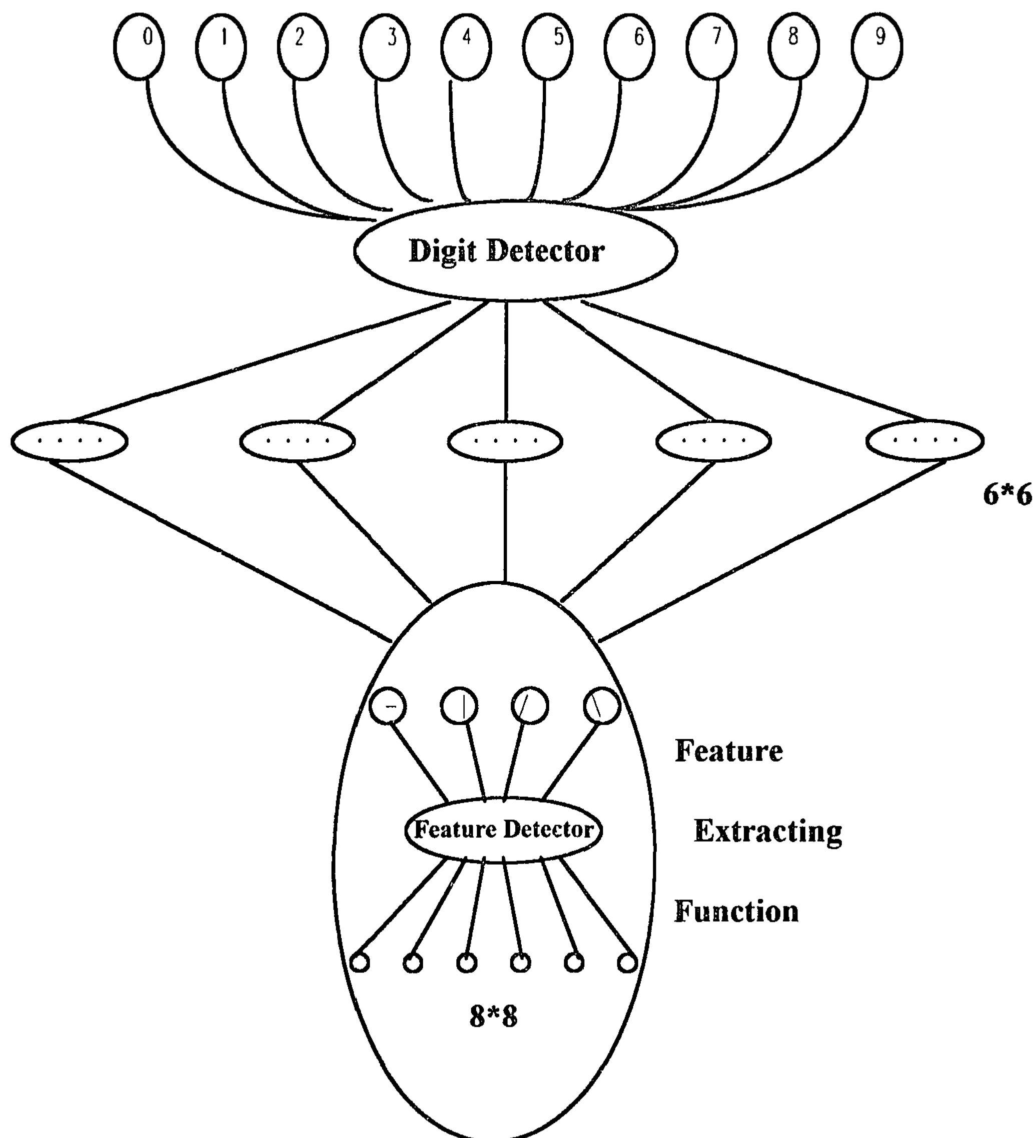
The structure of a typical figurette detection net is this:




At the top level of the structure are the individual nets for each figurette that the detector is capable of recognizing. These are the nets that can be activated and trained (or added and removed) independent of each other. The other nets in this structure are always shared. The central level is the core of the detector which holds knowledge about the image and interrelations among the figurettes. The bottom level nets cover the domain upon which the current detector currently operates. The current detector is applied to a small part of an image at a time. Its input corresponds to yet smaller parts of the image and in its turn is also collected from its parts one at a time. Therefore, the input to the current level is an array of

nets that each represent the output of the previous level detector for parts of the domain covered by the current level detector.

The digit recognition application network can be schematically depicted as follows:




The digit detector input nodes are connected to a procedural function that returns the degree of activation of the various features in the area which each input

node to the digit detector covers. This function in its turn activates a feature detector upon the pertaining (8 by 8) area. In fact, it activates it nine times in each region, slightly sliding the focus in its input region for each activation, and choosing the highest activation, thus creating an effect that we would like to refer to as "local saccadic motion," without delving into the implication of a biological analogy.

Deviating from biological analogy is, in fact, intentional, as we are trying to accommodate for the media that we are using.

It may be of interest to contrast this approach with that of the Neocognitron devised by Fukushima et al. [31]. The Neocognitron addressed the same issue that we did with the same general analysis of the problem: It could successfully recognize written digits by using a hierarchy of features. The conceptualization of the Neocognitron, like that of our model, started with looking at models of the biological visual system. The Neocognitron went on to imitate it. We use the term "imitate" because its authors relinquished exact simulation of the visual system in favor of achieving better results. We will argue that they had to. Conversely, though having adopted the biological understanding of the problem, we composed a solution for a **computer** to be used as an **artificial** intelligence classifier and to be evaluated as a computer algorithm.

The Neocognitron was an ingenious feat. It had an elaborate structure that followed (assumptions about) the structure of the visual system. It had nine alternating layers of "simple cells" and "complex cells". Cells within a layer were organized in planes according to their capacity. Basically, cells could recognize features in an area of the image. Simple cells could recognize actual features in an area of the image and complex cells were used as what perhaps amounted to logical gates. Simple cells were connected to complex cells of previous layers by

connections of trainable weights. Having had as many as nine layers of decreasing cell population and increasing descriptive contents, the net had considerable tolerance for distortion in the input image.

The Necognitron worked thanks to a seeming unreasonably meticulous engineering of connections and specialized complex cells. We think that it had two inherent flaws:

1. It adhered to a biological model which was impractical for a computer.
2. The biological model it assumed was perhaps impractical even for a brain.

The complex cells in the neocognitron joined the reaction of simple cells like logical gates. In the highest level they joined reactions to various forms of digits. This, of course, leads to grandmother cells, but yet further, to specialized hardware for each decision made toward the activation of each grandmother cell. For a biologist doing computer science, this mechanism may be just fine. For a computer scientist doing biology, it may be rejected as improbable. For a computer scientist doing computer science, it should be rejected as irrelevant.

Our method does not aspire to simulate the brain, it tries to reach a good solution for pattern recognition. It tries to use some advantages of an electronic computer and avoid its disadvantages (speed, modularity and recursion vs. massive parallelism). Further it tries to use the advantages of both procedural and connectionist processing.

It appears that the relative simplicity of our algorithm in the application level, the distance from lower level details and ease of training that it inherits from back propagation nets, and the executive power it inherits from procedural processing which enables it to handle a much higher resolution all make it a better algorithm for an electronic computer.

To us, this exemplifies the philosophy behind our approach to doing neural nets on artificial computers.

## 9. Conclusion

The experimentation we have been conducting with our neural programming methodology is encouraging.

The language comprehension application has been able to hold and use knowledge about syntax. It demonstrated the power of the cooperation between inductive and deductive reasoning. The application could parse some elaborate sentences. However, having been implemented on a 386 based personal computer, it performed on a rather small scale (tens of sentences) and the extent to which it could learn the syntax of a natural language still remains to be seen.

The digit recognition application demonstrated the convenience, ease and functionality that neural nets can deliver provided that they are used within a proper methodology. The application is far simpler and more natural than other application of similar functionality, and, further, it can be readily generalized and reused in other pattern recognition tasks.

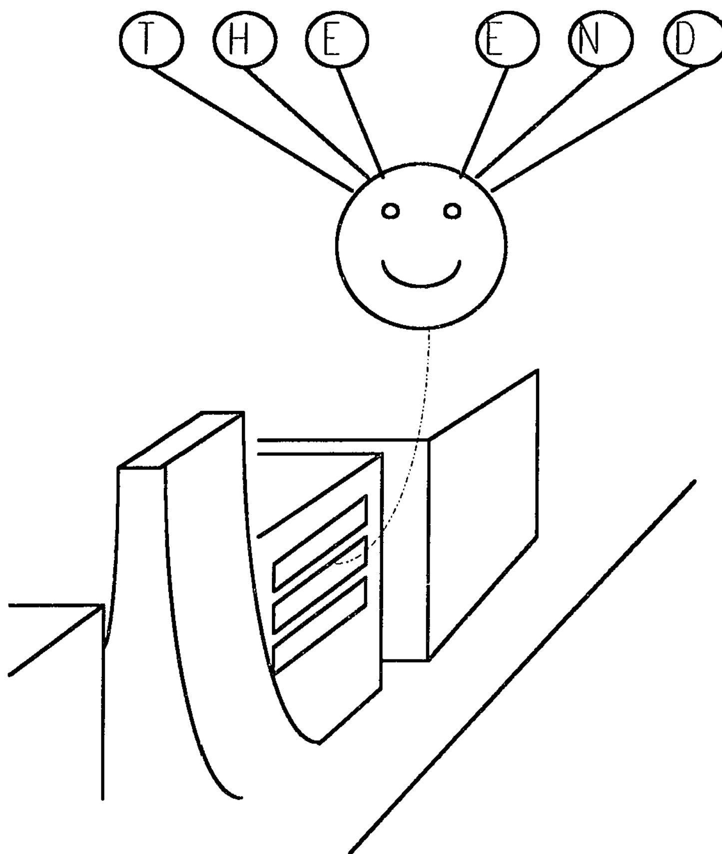
## 10. Future Research

In the long range we think the scheme we have presented can help in the solution of any problem to which neural nets are applied.

Currently, we would like to provide further proof of the soundness of the language comprehension model. To that end we need to implement it on a stronger hardware configuration.

We would like to implement our design of financial forecasting to a specific financial factor. This would necessitate coding the outlined net program and the collection of a sizable amount of data plus experimenting with parameters of the system.

We would like to develop the digit recognition system into an object recognition system. We would do that by adding levels of recognizable "figurettes". We would like the system to expand into being able to analyze a picture and its constituents - objects and their parts.



# Appendix

## Some C Code of $C^{+NET}$

### CPNET.H (C Header file)

```
# define netnamel 20
typedef char netname[netnamel+1];
typedef char lists[1];
typedef lists __far *netlists;

# define on  = 1
# define off = 0
# define yes 1
# define no 0

typedef char str[128];
# define strcatcut(a,b) strncat(a,b,sizeof(a)-strlen(a)-1)
void strcopy(char *to, char *fr, int maxlen);

int putout(char *format,...);
int flushout(char *format,...);
void display(char *message);
void pushdinfo(char *msg1,...);
void popdinfo(void);
void prindinfo(void);
# define _at NULL,__FILE__,__LINE__
int abend(char *format,...);

enum directives {Proceed,Pass,Nomore,Freeze,Ignore,Isinstm};

typedef struct
 {
   float tolerneuron;
   float tolerpattern;
   float tenacity;
   float rate;
   float smoothing;
   float feedprobability;

 } netparms;

# include <cpntmaxl.h>
typedef int netsizes[maxlayers+1];

# define layersize(layer) \
   Players[layer]->layersize
```

```
# define activation(neuron,layer) \
    (*(Players[layer]->acts))[neuron]

# define netput(neuron,layer) \
    (*(Players[layer]->netputs))[neuron]

# define outpat(neuron) \
    (*(Pinst->outpatptr))[neuron]

# define delta(neuron,layer) \
    (*(Players[layer]->deltas))[neuron]

# define weight(to,from,layer) \
    (Players[layer]->weights)[(from)*layersize(layer)+(to)]

# define OlDel(to,from,layer) \
    (*(Players[layer]->oldels))[(from)*layersize(layer)+(to)]

# define outact(net,neuron) \
    (SetNet(net),activation((neuron),outlayer))

# define Poutact(net) \
    (SetNet(net),&activation(1,outlayer))

# define numinpats_of(net) \
    (SetNet(net),Pinst->numinpats)

typedef float matrix[1];
typedef struct l
 {
    struct l *nextlayer;
    int      layersize;
    int      prevsize;
    matrix   *acts;
    matrix   *netputs;
    matrix   *deltas;
    matrix   *oldels;
    matrix   weights;

 } layers;

typedef struct m
  {
      netname namefr;
      netname nameto;
      size_t lnetfr;      /*symtab entry for namefr          */
      size_t lnetto;      /*symtab entry for nameto          */
      netname feedprobability; /*of netfr when run by netto   */
```

```
        float backprobability; /*of netto through netfr        */
        struct m *Pconnfr;   /*next 'from' connection for nameto */
        struct m *Pconnto;   /*next 'to'   connection for namefr */
        layers *frlayer;            /*output layer of namefr */
        layers *tolayer;            /*input  layer of nameto */
        layers *connlayer;
        size_t backingit;     /* is it being backed          */
       } connection;

typedef struct
  {
   matrix *inps[1];
   } inputs;

typedef struct
  {
   matrix *pats[1];
  } patterns;

typedef struct n  /* a network instance */
  {
       netname name;
       netparms parms;
       layers *Plays;
       connection *Pconnfr;
       connection *Pconnto;
       netlists conamelist;
       size_t *visited;
       size_t fed;
       size_t backed;
       size_t freeze;
       size_t ignore;
       size_t skipped;
       size_t symentry;
       inputs *inptr;
       int inpsource;
       float(*inpmanip)(float) ;
       patterns *patptr;
       float(*patmanip)(float);
       matrix *outacts;
       matrix *outpatptr;
       float worsedif;
       float prevwdif;
       size_t outsize;
       size_t inpsize;
       size_t numinpats;
       size_t inpatnum;
       void(*convinpi)(str, matrix);
```

```
        void(*convinpo)(str, matrix);
        void(*convpati)(str, matrix);
        void(*convpato)(str, matrix);
        void *dynafunc;
        int dynaparm1;
        int dynaparm2;
        int dynaparm3;
        int dynaparm4;
        void *dynaparmP;
        void *sidefunc;
        int sideparm1;
        int sideparm2;
        int sideparm3;
        int sideparm4;
        void *sideparmP;
        size_t sidenum;
        struct stmentry *pentry;

        struct n *prev;

        enum directives last_directive;

      } network;

typedef network *netptr;

# define frand(a)  (a)*(float)rand() / RAND_MAX
# define frnd(a,b) (frand((b)-(a))) + (a)
# define MARKrand -177.1
# define MARKEDrand(a) ((int)(a) == -177)

# define fromConnections 1
# define fromFile    0
# define fromFunc    0
# define fromPatterns   2
# define fromNowhere   -1
# define RunFromConnections (Pinst->inpsource == fromConnections)

enum learnmodes {NoLearn,LocaLearn,GlobaLearn};

typedef struct
  {
      enum directives directive;
      matrix *inp;
      matrix *pat;
     }
      funcinpat;
```

```
# define setparm(net,parm,val) (SetNet(net),Pinst->parms.parm=(val))

# define is_ignored(net) (SetNet(net),Pinst->ignore)

typedef float gencode[1];
# define inward 0
# define outward 1
# define inout 2
# define addin 3

void strcopy(char *to, char *fr, int maxlen);
void __far *alloc (size_t size);
void __far *ralloc (void *P,size_t size);
#define dealloc(P)  P = (P ? free((void *) (P)), NULL : P)
size_t lookup (netname net);
size_t SetNet(netname net);

void pushenv(netlists netlist);
void popenv(netlists netlist);
void connets(netlists netsfr, netlists netsto);
int CreateNeuNet(netname net, netparms  parms, netsizes sizes);
CreateNeuNets(netlists netlist, netparms  parms, netsizes sizes);
void CopyNet(netname netfr, netname netto);
int checkpoint();

typedef funcinpat inpats(int,int,int,int,int,void *);
typedef funcinpat sidefunc(int,int,int,int,int,void *);

void setinpat_file(netname net,int source,char *fname);
void setinpat_func (netname net,
            int source,
            inpats *func,
            int parm1,
            int parm2,
            int parm3,
            int parm4,
            void *parmP);
void setinpat_func_dynam
      (netname net,int source,inpats *func,
          int parm1,int parm2,int parm3,int parm4,void *parmP);
void setinpat(netname net, int source);
void freeinpat(int neti);
void resetinpat(netlists netlist);
void resetinpnum(netlists netlist);
void setinpat_manip(netname net,float(*inpmanip)(float),float(*patmanip)(float));
void set_convinpi(netname net,void(*convinpi)(str, matrix));
void set_convinpo(netname net,void(*convinpo)(str, matrix));
void set_convpati(netname net,void(*convpati)(str, matrix));
```

```
void set_convpato(netname net,void(*convpato)(str, matrix));
void set_side_func
      (netname net,sidefunc *func,
            int parm1,int parm2,int parm3,int parm4,void *parmP);
void unset_side_func(netname net);
funcinpat genfunc(int inpnum,int inorout,int howmany,int dumm2,int dumm3,gencode *code);
void freeze(netlists netlist);
void defreeze(netlists netlist);
void ignore(netlists netlist);
void retain(netlists netlist);
void correspond(netname netfr,int neurfr,netname netto,int neurto,float K);
void corresall(netname netfr,netname netto);
void getact(float *to, netname net);
netname *car(netlists netlist);
netlists cdr(netlists netlist);
netlists list(size_t dumm, lists listi,...);
int freelist(netlists netlist);
int whereinlist(netlists netlist,netlists st);
lists *etceterate(lists netlist);
# define li_  list(0,
# define _st ,eolist)
#define isinlist(netlist,net) (whereinlist(netlist,net) >= 0)
int putout(char *format,...);
int flushout(char *format,...);
void pushdinfo(char *msg1,...);
void popdinfo(void);
void prindinfo();
int abend (char *format,...);
enum directives run1(lists netlist,size_t noise,size_t learn,size_t withprob);
enum directives run(netlists netlist);
int forback(lists netlist,enum learnmodes learnmode,int cycle,int checkeach);
void prinonet(int neti);
void prinet(lists netlist);
int learn(lists netlist,enum learnmodes learnmode);
int listlook(lists netlist);
netlists allcons(netlists to, netlists from);

typedef struct stmentry
  {
   int neti;
   netparms (*parms)[];
   matrix *pats[1];
  } stmentry;


typedef struct
  {
   size_t stmsize;
```

```
    matrix *stmuses;
    stmentry **stminps;
    stmentry **stmouts;
    enum learnmodes learnmode;
    size_t active;
    size_t numstminps;
    size_t numstmouts;
    netlists listin;
    netlists listout;
    netlists allinvolved;
    int lastm;
  } stm;

stm *stmize
  (lists netlist, lists innets, size_t stmsize, enum learnmodes learnmode);
int copystm(netlists listo, netlists listfrom);

void clearstm(stm *stmP);
int destmize(netlists netlist);

int initnets(char *outfile);
void finishnets();

int retrieveall(char *fname);
int retrievenet(char *fname, netname net, netname storeas);
int storeall(char *fname);
int storenet(char *fname, netname net, netname storeas);
```

## C Code of Major Functions of C+NET

## Feedforward

```
enum directives feedforward
  (netlists netlist,size_t inpnum,size_t pathlevel,size_t istop,size_t noise,
   size_t learn,size_t withprob)
 {
  netlists rest;
  size_t i;
  int stmpatnum;
  size_t inpn;
  stm *stmP;
  netname *net;
  int neti;
  enum directives feedresult;
  enum directives ret;
  float feedprobability;

  pushdinfo("feedforward ",etceterate(netlist),_at);

  pushnet;

  inpn = inpnum;
  if(!inpn)
   {
    for(rest=netlist;net=car(rest);rest=cdr(rest))
     {
      SetNet(net);
      if ( !(Pinst->ignore||Pinst->freeze||Pinst->skipped||
          (Pinst->inpsource == fromNowhere)))
       {
        inpn = Pinst->inpatnum + 1;
        break;
       }
     }
    if (!inpn)
     abend("feedforward %s, there were no active input nets",
         etceterate(netlist));
   }
  stmP=istm(netlist);
  stmpatnum=checkstm(stmP,inpn,pathlevel,learn);
  if (stmpatnum == -1)
    goto nomore;

  ret = Pass;
  rest=netlist;
  for(i=0;net=car(rest);rest=cdr(rest),i++)
```

```
    {
     neti = SetNet(net);
     feedprobability = 1.;
     if (learn && withprob)
       feedprobability = atof(carnumeral(rest));
     feedresult =
       feedforward1(neti,inpn,pathlevel,istop,noise,stmP,i,stmpatnum,
                 learn,withprob,feedprobability);
     if (debugging)
       flushout("\n feedresult for %s: %d",netname_of(neti),feedresult);
     if (feedresult == Nomore)
        goto nomore;
     if (feedresult == Proceed)
       ret = Proceed;
    }
  if (stmP && !stmpatnum && !stmP->learnmode)
    {
     vistacktop=0; flushout("**** calling putstm from feed");
     putstm(stmP,0);
    }
  if ((ret == Proceed)&&(stmpatnum))
     ret = Isinstm;
  popnet; popdinfo();
  return (ret);

  nomore: popnet; popdinfo(); return(Nomore);
 }

enum directives feedforward1
  (int neti,size_t inpnum,size_t pathlevel,size_t istop, size_t noise,
   stm *stmP,size_t indinlist,int stmpatnum,
   size_t learn,size_t withprob,float feedprobability)
 {
   int i, ii, k, firstfed;
   register int j, highj;
   register float actin;
   connection *Pcon;
   matrix *Pa;
   float maxnoise;
   netname *net;
   enum directives ret;
   float feedprob;

   giveway();

   SetNeti(neti);
   net=netptr_of(neti)->name;
```

```
if (debugging)
  flushout
     ("\nfeedf1 %s inpnum=%d fed=%d,ig=%d,last_d=%d,vis=%d,inpso=%d,pent=%d",
     Pinst->name,(int)inpnum,Pinst->fed,Pinst->ignore,
     Pinst->last_directive,*Pinst->visited,Pinst->inpsource,Pinst->pentry!=0);

if (((*Pinst->visited)+1) > pathlevel)
  return (Pinst->last_directive);

if (Pinst->freeze)
  return (Pinst->last_directive);

Pinst->fed off;
Pinst->backed off;

if (Pinst->ignore) return (Ignore);

if (Pinst->inpsource == fromNowhere)
  {
   Pinst->skipped on;
   return (Ignore);
  }

Pinst->inpatnum = inpnum;

if (   (Pinst->pentry)
    && (inpnum <= Pinst->numinpats)
    && (Pinst->pentry->pats[inpnum-1]))
  {Pinst->parms=(*(Pinst->pentry->parms))[inpnum-1];
    if (!Pinst->prev)
      abend("feedforward1: pentry bad");
    }

Pinst->skipped off;
if (feedprobability < 1.)
  feedprob = feedprobability;
else
  feedprob = Pinst->parms.feedprobability;

if (withprob && (feedprob < 1))
  if (feedprob > frnd(0.,1.))
   {
    Pinst->skipped on;
    return(Ignore);
   }
(*Pinst->visited)++;
```

```
if (Pinst->sidefunc)
 {
  if (((sidefunc *) Pinst->sidefunc)(++Pinst->sidenum,
                                     Pinst->sideparm1,
                                     Pinst->sideparm2,
                                     Pinst->sideparm3,
                                     Pinst->sideparm4,
                                     Pinst->sideparmP).directive==Nomore)
   {
    SetNeti(neti);
    Pinst->sidenum=0;
   }
  SetNeti(neti);
 }

maxnoise = .25*(1.0-Pinst->parms.tolerneuron);

if (Pinst->inpsource == fromPatterns)
 {
  ret = getpattern(neti);
  Pinst->last_directive = ret;
  if (ret == Pass)
    Pinst->skipped on;
  if (ret==Nomore)
    Pinst->inpatnum = 0;
  if (ret !=Proceed)
     return (ret);

  if (trace||debugging)
    flushout("\nrunning %s from patterns inp %d of %d",
           Pinst->name,Pinst->inpatnum,Pinst->numinpats);

  for (i=1;i<=Pinst->outsize;i++)
   activation(i,outlayer) = outpat(i);
  Pinst->fed on;
  if (noise)
    for (i=1;i<=Pinst->outsize;i++)
       activation(i,outlayer) = activation(i,outlayer)
                          + frnd(-maxnoise,maxnoise);

  return (Proceed);
 }
if  RunFromConnections
 {
  if (Pinst->conamelist==nullist)
    abend (" no input/connections to %s ",Pinst->name);
  if (Pinst->pentry)
    if (inpnum > Pinst->numinpats)
```

```
        {
         Pinst->inpatnum = 0;
         Pinst->last_directive = Nomore;
         return (Nomore);
        }
      else if (!Pinst->patptr->pats[inpnum-1])
        {
         Pinst->skipped on;
         return (Pass);
        }
   if (debugging)
     flushout("\n feeding %s by running: %s",Pinst->name,Pinst->conamelist);

   ret = feedforward(Pinst->conamelist,inpnum,
        *Pinst->visited,no,noise,learn,withprob);
   if (ret == Nomore)
     {
      Pinst->inpatnum = 0;
      Pinst->last_directive = Nomore;
      return (Nomore);
     }
   if (ret == Pass)
     {
      Pinst->skipped on;
      Pinst->last_directive = Pass;
      return (Pass);
     }
   firstfed = inplayer;
   Pinst->last_directive = Proceed;
  }
 else
  {
   firstfed = inplayer +1;
   ret = getinput(neti);
   Pinst->last_directive = ret;
   if (ret == Pass)
     Pinst->skipped on;
   if (ret==Nomore)
     Pinst->inpatnum = 0;
   if((ret==Pass)||(ret == Nomore))
     return(ret);
   if (noise)
     for (i=1;i<=Pinst->inpsize;i++)
        activation(i,inplayer) = activation(i,inplayer)
                      + frnd(-maxnoise,maxnoise);
  }
 vistacktop++;
 if (vistacktop >= vistackmax)
```

```
    abend ("vistacktop > max");
   vistack[vistacktop] = istop ? -Pinst->symentry : Pinst->symentry;

   if (trace||debugging)
     flushout("\nrunning %s inp %d of %d",
            Pinst->name,Pinst->inpatnum,Pinst->numinpats);
   Pinst->fed on;

   for (k = firstfed; k <= outlayer; k++)
    for (i = 1; i <= layersize(k); i++)
    {
     actin = 0.0;

      /* a layer may be fed (also) by a connection chain */

     if ((k==inplayer)&&RunFromConnections)
      {
       Pcon = Pinst->Pconnfr;
       Players[inplayer] = Pcon->connlayer;
       Players[inplayer-1] = Pcon->frlayer;
      }
     else
      {
       Pcon = NULL;
       Players[inplayer] = Pinst->Plays;
      }
     do
      {
       highj = layersize(k-1);
       if ((Pcon == NULL)
        || (netptr_of(Pcon->lnetfr)->fed))
        for (j = 0; j <= highj; j++)     /* j is 0 for threshold unit */
            actin += weight(i,j,k) * activation(j,k-1);
       if (Pcon != NULL)
        {
         Pcon = Pcon->Pconnfr;
         if (Pcon  != NULL)
          {
           Players[inplayer] = Pcon->connlayer;
           Players[inplayer-1] = Pcon->frlayer;
          }
        }
      } while (Pcon != NULL);
     Players[inplayer] = Pinst->Plays;
     activation(i,k) = sigmoid(actin);
    }
  return (Proceed);
}
```

## Backpropagation

```
void calcdeltas(int neti,size_t istop,int cycle)
{
 int n, k, lastbacked;
 register int i, highi;
 register float del, delw, rate, smoothing;
 connection *Pcon,*Pc;
 size_t was;

 SetNeti(neti);

 pushdinfo("calcdeltas ",netname_of(neti),_at);

 giveway();

 if (istop)
  if (getpattern(neti)!=Proceed)
   {
    putout("\n *** calcdeltas %s - no pat",netname_of(neti));
    return;
   }

 if (trace) flushout("\nbackproping %s cy %d",netname_of(neti),cycle);

 if RunFromConnections
   lastbacked = inplayer;
 else
   lastbacked = inplayer+1;

 for (k = outlayer;k >= lastbacked; k--)
  {
   for (n = 1; n <= layersize(k); n++)
    {
     if (k == outlayer)
      if (istop)
        del = (outpat(n) - activation(n,k));
      else
       {
        del = 0.0;
        was off;
        for (Pcon=Pinst->Pconnto;Pcon!=NULL;Pcon=Pcon->Pconnto)
          if (netptr_of(Pcon->Inetto)->backed)
           {
            if (Pcon->backprobability == 0.)
              continue;
            if (Pcon->backprobability < 1.)
             {
```

```
          if (n==1)
            Pcon->backingit = Pcon->backprobability > frnd(0.,1.);
          if (!Pcon->backingit)
              continue;
         }
        was on;
        Players[outlayer+1] = Pcon->connlayer;
        highi = layersize(k+1);
        for (i=1;i <= highi;i++)
         del += (delta(i,k+1)*weight(i,n,k+1));
       }
   if (!was)
     goto sof;
  }
else /* (k != outlayer) */
 {
  highi = layersize(k+1);
  for (del=0.0,i=1;i <= highi;i++)
    del += (delta(i,k+1)*weight(i,n,k+1));
 }
del *= SigTag(activation(n,k));
delta(n,k) = del;

if ((k == inplayer)  && RunFromConnections)
  Pcon = Pinst->Pconnfr;
else
  Pcon = NULL;
do
 {
  Pc=NULL;
  if (Pcon != NULL)
      {
       Pc=Pcon;
       Pcon = Pcon->Pconnfr;
       if (!(netptr_of(Pc->Inetfr)->fed))
         continue;
       Players[inplayer] = Pc->connlayer;
       Players[inplayer-1] = Pc->frlayer;
       delta(n,k) = del;
      }
  else
      Players[inplayer]=Pinst->Plays;
  rate = Pinst->parms.rate;
  smoothing = Pinst->parms.smoothing;
  highi = layersize(k-1);
  for (i = 0; i <= highi; i++) /* i = 0 is the  */
    {                       /* threshold unit*/
     delw = del * activation(i,k-1);
```

```
            weight(n,i,k)
                 += (delw * rate) + (OlDel(n,i,k) * smoothing);
            if ((cycle==393) && (Pinst->inpatnum == Pinst->numinpats))
              if (Pc&&(k==1))
              putout("\n%11s,w(%2d,%2d,%2d) (cy %2d) = %8.5f fr %11s",
                 Pc->nameto,n,i,k,cycle,weight(n,i,k),Pc->namefr);
              else
              putout("\n%11s,w(%2d,%2d,%2d) (cy %2d) = %8.5f",
                 Pinst->name,n,i,k,cycle,weight(n,i,k));
            OlDel(n,i,k) = delw;
           }
        } while(Pcon != NULL);
        Players[inplayer] = Pinst->Plays;
        Pc=NULL;
      }
    }

  Pinst->backed on;
sof:
  popdinfo();
  }
```

**Learn**

```
int learn(lists netlist,enum learnmodes learnmode)
 {
   int good, goods, bads;
   int   inpats;
   float hitratio = 0.0;
   int cycle = 0;
   int printeach;
   int checkeach;
   size_t holdtrace = 0;
   time_t reference_time;
   netname *net;
   lists *rest;
   size_t wasinstm off;
   size_t maxicle;
   size_t upto;

   pushdinfo("learn ",netlist,_at);
   pushnet;

   if (trace) flushout("\nlearning %s ",etceterate(netlist));

   rest=netlist;
   while(net=car(rest))
    {
     SetNet(net);
     if (!Pinst->freeze && !Pinst->ignore)
       break;
     rest=cdr(rest);
    }
   if (!net) abend ("no active nets %s",netlist);

   reference_time = time(NULL);

   holdtrace = trace;

   upto = tenacity * tenacity * 5000;
   maxicle = 5*((int)(upto+6)/5);
   cycle = 0;
   while (hitratio < (1. - ((1-tenacity*tenacity)/10.))
        && (cycle < maxicle))
    {
     cycle++;

     goods=bads=0;

     if     (cycle <=  10)  printeach =  10;
```

```
else if (cycle <= 100)  printeach =  50;
else                    printeach = 500;

if     (cycle <=   5)  checkeach =   5;
else if (cycle <=  20)  checkeach =  10;
else if (cycle <=  50)  checkeach =  25;
else if (cycle <= 100)  checkeach =  50;
else                    checkeach = 100;

if (cycle > (maxicle - 5))
  checkeach = 5;

if (!(cycle % 30) == 0)
  check_checkpoint();

for (inpats=0;;)
  {
   if (debugging)
      putout("\n%s inpats %d",netlist,inpats);
   printcycle = ((cycle % printeach)==0);
   checkcycle = (cycle == 1) || ((cycle % checkeach)==0);
   if (printcycle)
     {
      putout("\n%s",etceterate(netlist));
      putout("\ncycle %d ",cycle);
      putime();
      putout("time elapsed %-9.0f ",
             difftime(time(NULL),reference_time));
     }
   if (printcycle) trace = holdtrace;

   good = forback(netlist, learnmode, cycle, checkeach);
   if (debugging)
      putout("\n forback result is %d for %s",good,netlist);

   if (good < 0)
     break; /*not run (end of input)*/

   inpats++;

   trace off;
   printcycle = ((cycle % printeach)==0);
   if (printcycle) prinet(netlist);

   if (checkcycle)
     if (good)
       goods++;
     else
```

```
            bads++;
        }
         if (!inpats) abend("learn %s missing inpats ",etceterate(netlist));

         if (checkcycle)
           hitratio = ((float)(goods)) / ((float)(goods + bads));
    }

    if (printset||debugging)
     if (cycle > 1)
      {
       putout("\n cycle %d %s",cycle,etceterate(netlist));
       putout(", learned time elapsed %f ",difftime(time(NULL),reference_time));
       prinet(netlist);
      }
     trace = holdtrace;
     resetinpnum(netlist);

     popnet;
     popdinfo();
     return(cycle-1);
 }

int forback(lists netlist,enum learnmodes learnmode,int cycle,int checkeach)
 {
  int i,j,right,good=1,giveup=2,wasinstm=3,notrun=-1,bad=0,ret;
  int neti;
  size_t istop;
  netname *net;
  lists *rest;
  enum directives feedresult;
  float dif,improve_rate,adjusted_tenacity,ten;
  int modcycle;

  static int zero_stack[vistackmax] = {0};
       int keep_stack[vistackmax];
       int keep_top;

  pushdinfo("forback ",netlist,_at);
  pushnet;

  memcpy(keep_stack,vistack,(vistacktop+1)*sizeof(int));
  memcpy(vistack,zero_stack,(vistacktop+1)*sizeof(int));
  keep_top=vistacktop;

  modcycle =
     cycle == (checkeach-1) ? (checkeach+1) : ((cycle-1) % checkeach) + 1;
```

```
feedresult = run1(netlist,yes,yes,((cycle>1)&&(modcycle<(checkeach-2))));
  if (debugging) flushout("\n run1 feedresult %d ",feedresult);
if (feedresult == Isinstm)
 {
  ret = wasinstm;
  goto end;
 }
if (feedresult == Nomore)
 {
  ret = notrun;
  goto end;
 }
if (feedresult == Pass)
  flushout("\n*** %s learning pass skipped (no active inputs)",
   etceterate(netlist));

if (modcycle < (checkeach-2))
  ret = bad;

else
 { /*check proximity of result*/
  ret = good;
  ten = Pinst->parms.tenacity ? Pinst->parms.tenacity : tenacity;
  adjusted_tenacity = (pow(10,ten*ten*4)-1)*checkeach;

  rest=netlist;
  j=0;
  for(;net=car(rest);rest=cdr(rest))
   {
    j++;
    neti=SetNet(net);

    if (modcycle < (checkeach-2))
      Pinst->worsedif = 0;

    if (!Pinst->fed)
      continue;

    if (getpattern(neti)!=Proceed)
      continue;
    improve_rate =    (adjusted_tenacity/Pinst->parms.rate)
              / (1+adjusted_tenacity/Pinst->parms.rate);

    right = 0;
    for (i = 1; i <= Pinst->outsize; i++)
     {
      dif = fabs(activation(i,outlayer)-outpat(i));
      if (dif<(1.0 - Pinst->parms.tolerneuron))
```

```
        right = right + 1;
      if (modcycle == (checkeach-2))
         if (dif > Pinst->worsedif)
          Pinst->worsedif = dif;
    }
    if (modcycle == checkeach)
      Pinst->prevwdif = Pinst->worsedif;

    if (right < (Pinst->outsize * Pinst->parms.tolerpattern))
     if ((modcycle != (checkeach-1))
        || (Pinst->worsedif < (Pinst->prevwdif*improve_rate)))
      {
       ret=bad;
       if (modcycle < (checkeach-3))
         break;
      }
     else
      {
       if (ret != bad)
         ret = giveup;
       if (Pinst->inpatnum == 1)
          putout
          ("\n=== giving up on %s cycle %d previous sum dif %8.5f sum dif %8.5f"
                   ,net,cycle,Pinst->prevwdif,Pinst->worsedif);
      }
   }
  }
 for (i=vistacktop;i>0;i--)
  {
   neti=vistack[i];
   istop = no;
   if (neti<0)
     {
      istop = yes;
      neti = -neti;
     }
   if ((learnmode==GlobaLearn)||istop)
      calcdeltas(neti,istop,cycle);
  }
 end:
 vistacktop=keep_top;
 memcpy(vistack,keep_stack,(vistacktop+1)*sizeof(int));

 popnet;
 popdinfo();
 return(ret);
}
```

**Setinput**

```
void setinpat_file(netname net,int source,char *fname)
 {

   FILE *fp;
   int i, j, inpsize, patsize;
   char a[20], nextc;
   size_t numinputs=0, numpatterns=0;
   int neti;

   pushdinfo("setinpat_file ",net,_at);
   pushnet;
   neti=SetNet(net);
   fp = fopen(fname,"r");
   a[0] = 0;

   if (source == fromFile)
     {
      fscanf(fp,"%s",a);
      if (a[0] == 'i')
       {
        freeinput(neti);
        inpsize = layersize(inplayer);
        nextc = '0';
        for (i=1;(!isalpha(nextc));i++)
           {
            Pinst->inptr=ralloc(Pinst->inptr,sizeof(matrix *)*i);
            Pinst->inptr->inps[i-1] = alloc(sizeof(matrix)*inpsize);
            for (j=1;(j<=inpsize);j++)
              fscanf(fp, "%f", &(inp(i,j)));
            while((!feof(fp))&&(!isalnum(nextc=getc(fp))));
            if (!feof(fp)) ungetc(nextc,fp);
            }
          numinputs = i-1;
       }
     }

   if (!feof(fp))
    {
     while (a[0] != 'p')
       fscanf(fp,"%s",a);
     freepat(neti);
     patsize = layersize(outlayer);
     for (i=1;(!feof(fp));i++)
       {
        Pinst->patptr=ralloc(Pinst->patptr,sizeof(matrix *)*i);
        Pinst->patptr->pats[i-1] = alloc(sizeof(matrix)*patsize);
```

```
        for (j=1;(j<=patsize);j++)
          fscanf(fp, "%f", &(pat(i,j)));
       while((!feof(fp))&&(!isalnum(nextc=getc(fp))));
       if (!feof(fp)) ungetc(nextc,fp);
      }
     numpatterns = i-1;
    }

   fclose(fp);

   Pinst->inpatnum=0;
   Pinst->numinpats = numpatterns;
   if (numinputs < numpatterns)
     Pinst->numinpats=numinputs;
   if ((numinputs)&&(numpatterns)&&(numinputs!=numpatterns))
    putout
    ("\n%s: warning, num inputs and patterns not equal, lower assumed",net);

   Pinst->dynafunc=NULL;
   Pinst->sidefunc=NULL;
   Pinst->ignore off;

   Pinst->inpsource = source;
   popnet;
   popdinfo();
 }

enum directives set1inpat_func (int neti,
              int inpnum,
              size_t replace,
              int inpsource,
              inpats *func,
              int parm1,
              int parm2,
              int parm3,
              int parm4,
              void *parmP)
 {
  int i, j, inpsize, patsize;
  funcinpat inp;

  SetNeti(neti);

  inp = (*func)(inpnum,parm1,parm2,parm3,parm4,parmP);
  SetNeti(neti);

  if ((replace)&&(inpnum == 1))
   {
```

```
 if (inp.inp)
   freeinput(neti);
 if (inp.pat)
   freepat(neti);
}

if (inp.directive != Nomore)
{
 if (!(inp.inp||inp.pat))
    inp.directive == Pass;

 inpsize = Pinst->inpsize;
 patsize = Pinst->outsize;

 if ((inp.inp != NULL)||(inp.directive==Pass))
  if (inpsource == fromFunc)
    {
     if (!Pinst->inptr)
        Pinst->numinpats = 9999;
     if ((Pinst->numinpats == 0) || (Pinst->numinpats == 9999))
       {
        Pinst->inptr=ralloc(Pinst->inptr,sizeof(matrix *)*inpnum);
        Pinst->inptr->inps[inpnum-1] = NULL;
       }
     if (inp.inp)
      {
       if (!Pinst->inptr->inps[inpnum-1])
         Pinst->inptr->inps[inpnum-1] = alloc(sizeof(matrix)*inpsize);
       for (j=1;(j<=inpsize);j++)
         inp(inpnum,j) = *(inp.inp)[j-1];
      }
     else
       dealloc(Pinst->inptr->inps[inpnum-1]);
    }

 if ((inp.pat != NULL)||(inp.directive==Pass))
    {
     if (!Pinst->patptr)
        Pinst->numinpats = 9999;
     if ((Pinst->numinpats == 0) || (Pinst->numinpats == 9999))
       {
        Pinst->patptr=ralloc(Pinst->patptr,sizeof(matrix *)*inpnum);
        Pinst->patptr->pats[inpnum-1] = NULL;
       }
     if (inp.pat)
      {
       if (!Pinst->patptr->pats[inpnum-1])
         Pinst->patptr->pats[inpnum-1] = alloc(sizeof(matrix)*patsize);
```

```
        for (j=1;(j<=patsize);j++)
          pat(inpnum,j) = *(inp.pat)[j-1];
        }
      else
        dealloc(Pinst->patptr->pats[inpnum-1]);
     }
  }

  return(inp.directive);
}

void setinpat_func (netname net,
               int source,
               inpats *func,
               int parm1,
               int parm2,
               int parm3,
               int parm4,
               void *parmP)
 {
   int neti;
   int inpnum=0;

   pushdinfo("setinpat_func ",net,_at);

   pushnet;
   neti=SetNet(net);
   Pinst->dynafunc=NULL;
   Pinst->sidefunc=NULL;
   Pinst->inpatnum=0;
   while
    (set1inpat_func
         (neti,++inpnum,yes,source,func,parm1,parm2,parm3,parm4,parmP)
    !=Nomore);

   Pinst->numinpats=inpnum-1;
   Pinst->inpsource = source;
   Pinst->ignore off;
   popdinfo();
   popnet;
 }

enum directives set_dynamic_inputs(neti)
 {
  int ret;

  pushdinfo("set_dynamic_inputs",_at);
```

```
 pushnet;
 SetNeti(neti);

 set_one_inpat:
 ret=set1inpat_func(neti,
            Pinst->inpatnum,
            no,
            fromFunc,
            (inpats *)Pinst->dynafunc,
            Pinst->dynaparm1,
            Pinst->dynaparm2,
            Pinst->dynaparm3,
            Pinst->dynaparm4,
            Pinst->dynaparmP
           );
 if (ret == Nomore)
   Pinst->numinpats = Pinst->inpatnum-1;

 Pinst->ignore off;
 popnet;
 popdinfo();
 return(ret);
}

void setinpat_func_dynam
      (netname net,int source,inpats *func,
            int parm1,int parm2,int parm3,int parm4,void *parmP)
{
 int neti;

 pushnet;
 neti=SetNet(net);

 freeinpat(neti);
 Pinst->dynafunc = (void *)func;
 Pinst->dynaparm1 = parm1;
 Pinst->dynaparm2 = parm2;
 Pinst->dynaparm3 = parm3;
 Pinst->dynaparm4 = parm4;
 Pinst->dynaparmP = parmP;
 Pinst->inpsource = source;
 Pinst->numinpats = 9999;
 Pinst->inpatnum=0;

 Pinst->ignore off;

 popnet;
}
```

**STM**

```
int checkstm(stm *stmP,size_t inpnum,size_t pathlevel,size_t learn)
{
 int i, j, k;
 size_t patsize;
 matrix *Ppat;
 matrix *Pact;
 float noise;
 int bad, okbads;
 netptr Pneti;
 netptr Pthinst;
 stmentry *pentry;
 int neti;
 enum directives fres;

 if (!learn) return (0);
 if (!stmP) return (0);
 if (trace) putout("\nckeckstm %s ",etceterate(stmP->listout));
  if (debugging) flushout("\nstm listin is %s",stmP->listin);
 fres=feedforward(stmP->listin,inpnum,pathlevel,no,no,learn,no);
 vistacktop=0;
 if (fres == Nomore)
   return(-1);

 for (k=0;k<=stmP->lastm;k++)
  {
  for (i=0;i<stmP->numstminps;i++)
   {
    pentry=stmP->stminps[i];
    Pneti=netptr_of(pentry->neti);
    if ((!Pneti->fed) && (!pentry->pats[k]))
      continue;
    if ((!Pneti->fed) || (!pentry->pats[k]))
      goto next;
    Pact = Pneti->outacts;
    Ppat = pentry->pats[k];
    patsize = Pneti->outsize;
    noise = 1.0 - Pneti->parms.tolerneuron;
    if (Pneti->worsedif > noise)
      noise = Pneti->worsedif;
    okbads = patsize*(1-Pneti->parms.tolerpattern);
    bad = 0;
    for (j=0;j<patsize;j++)
     {
      if (fabs((*Pact)[j]-(*Ppat)[j])>noise)
       {
        bad++;
```

```
        if (bad > okbads) goto next;
       }
     }
    }
   goto pat_in_stm;
   next:;
  }

 if (trace) putout("notfound");

 if (learn && stmP->learnmode)
  {
   if (trace) putout(" adding");
   return(teachstm(stmP,0)+1);
  }
 else
   return (0);

 pat_in_stm:
 if(trace) putout("found");

 if (learn && stmP->learnmode)
  {
   if (trace) putout(" replacing");
   teachstm(stmP,k+1);
  }

 if ((*(stmP->stmuses))[k]<9000)
  ((*(stmP->stmuses))[k])++;

 return (k+1);
}

matrix *getstm(stm *stmP, size_t indinlist, int stmpatnum)
{
 return ((stmP->stmouts[indinlist])->pats[stmpatnum-1]);
}

int putstm(stm *stmP, size_t kfound)
{
 int i, j, k;
 size_t patsize;
 matrix *Ppat;
 matrix *Pact;
 netptr Pneti;
 stmentry *pentry;
 float usage, minusage, maxusage;
```

```
pushdinfo("putstm",_at);

minusage=9999; maxusage=0;
if (kfound)
 {
  k = kfound - 1;
  goto put;
 }

if ((stmP->lastm+1)<stmP->stmsize)
  {
   k = ++(stmP->lastm);
  }
else
  {
   for (i=0;i<stmP->stmsize;i++)
     if ((usage=(*(stmP->stmuses))[i])<minusage)
       {
        minusage = usage;
        k=i;
       }
     else if (usage>maxusage)
       maxusage = usage;
  }

(*(stmP->stmuses))[k]=maxusage+1;

if (maxusage > 1000)
   for (i=0;i<stmP->stmsize;i++)
     (*(stmP->stmuses))[i]=(*(stmP->stmuses))[i]-minusage;

put:

flushout(" putstm(%s,%d) ",etceterate(stmP->listout),k);

for (i=0;i<stmP->numstminps;i++)
 {
   pentry=stmP->stminps[i];
   Pneti=netptr_of(pentry->neti);
   patsize = Pneti->outsize;
   if (!Pneti->fed)
      dealloc(pentry->pats[k]);
   else
     {
      if (debugging)
         putout("\n putting %s",Pneti->name);
      if (!pentry->pats[k])
        pentry->pats[k]=alloc(sizeof(matrix)*patsize);
```

```
        Pact = Pneti->outacts;
        Ppat = pentry->pats[k];
        for (j=0;j<patsize;j++)
         {
          (*Ppat)[j]=(*Pact)[j];
          if (debugging)
            putout_skip("%8.5f",(*Ppat)[j],10);
         }
        if (debugging)
          clear_skips();
        if (!pentry->parms)
          pentry->parms = alloc(sizeof(netparms)*stmP->stmsize);
        (*(pentry->parms))[k]=Pneti->parms;
       }
    }

  for (i=0;i<stmP->numstmouts;i++)
   {
    pentry=stmP->stmouts[i];
    Pneti=netptr_of(pentry->neti);
    patsize = Pneti->outsize;
    if (!Pneti->fed)
       dealloc(pentry->pats[k]);
    else
      {
       if (!pentry->pats[k])
         pentry->pats[k]=alloc(sizeof(matrix)*patsize);
       Pact = Pneti->outacts;
       Ppat = pentry->pats[k];
       for (j=0;j<patsize;j++)
         (*Ppat)[j]=(*Pact)[j];
       if (!pentry->parms)
          pentry->parms = alloc(sizeof(netparms)*stmP->stmsize);
       (*(pentry->parms))[k]=Pneti->parms;
      }
    }

  popdinfo();
  return(k);
 }

int teachstm(stm *stmP,size_t kfound)
 {
  int ret;
  int i, j, k;
  netptr Pneti;
  netname *net;
  size_t save_inpatnum;
```

```
stmentry *pentry;
  netlists rest;
pushdinfo("teachstm",_at);

if (trace) putout("\nteaching stm");


for (i=0;i<stmP->numstmouts;i++)
 {
  pentry=stmP->stmouts[i];
  Pneti = netptr_of(pentry->neti);
  if (Pneti->ignore)
    Pneti->fed off;
  if ((Pneti->ignore)||(Pneti->freeze))
    continue;
  SetNeti(pentry->neti);
  save_inpatnum = Pneti->inpatnum;
  Pneti->inpatnum = Pneti->inpatnum + 1;
  if (getpattern(pentry->neti)!=Proceed)
   {
    putout("\ngetpat %s != Proceed inp %d of %d",
    Pneti->name,Pneti->inpatnum,Pneti->numinpats);
    Pneti->fed off;
   }
  else
   {
    Pneti->fed on;
    for (j=1;j<=Pneti->outsize;j++)
      activation(j,outlayer) = outpat(j);
   }
  Pneti->inpatnum = save_inpatnum;
 }

ret=putstm(stmP,kfound);


pushenv(stmP->allinvolved);

for (i=0;i<stmP->numstminps;i++)
  {
   pentry=stmP->stminps[i];
   Pneti=netptr_of(pentry->neti);
   Pneti->pentry=pentry;
   Pneti->inpsource=fromPatterns;
   Pneti->patptr=pentry->pats;
   Pneti->inpatnum=0;
   Pneti->numinpats=stmP->lastm+1;
   Pneti->ignore off;
```

```
    }
  for (i=0;i<stmP->numstmouts;i++)
    {
      pentry=stmP->stmouts[i]; /*assuming input from connections*/
      Pneti=netptr_of(pentry->neti);
      Pneti->pentry=pentry;
      Pneti->patptr=pentry->pats;
      Pneti->inpatnum=0;
      Pneti->numinpats=stmP->lastm+1;
      Pneti->ignore off;
    }

  stmP->active off;
  learn(stmP->listout,stmP->learnmode);

  flushout("\nstm %s taught",etceterate(stmP->listout));putime();
  stmP->active on;

  popenv(stmP->allinvolved);

  popdinfo();

  return(ret);
}


stm *stmize
  (lists netlist, lists innets, size_t stmsize, enum learnmodes learnmode)
 {
  int i,j,k;
# define maxstnets 128
  stmentry *pentries[maxstnets+1];
  netname *net;
  lists *rest;
  stm *stmP;
  netlists etc1,etc2;

  etc1=etceterate(netlist);
  etc2=etceterate(innets);
  pushdinfo("stmize ",etc1," with ",etc2,_at);
  if (trace)
     putout("\n stmizing %s with %s",etc1,etc2);
  freelist(etc1);freelist(etc2);

  pushnet;

  stmP=instm(netlist);
  clearstm(stmP);
```

```
stmP->stmsize=stmsize;
stmP->learnmode = learnmode;
stmP->lastm = -1;
stmP->active on;
stmP->listin = li_ innets _st;
stmP->listout = li_ netlist _st;
stmP->allinvolved = allcons(netlist,innets);

stmP->stmuses=alloc(sizeof(matrix)*stmsize);

rest=innets;
for(i=0;net=car(rest);i++)
  {
   if (i>maxstnets) abend ("stmize: too many nets");

   SetNet(net);
   pentries[i]=alloc(sizeof(stmentry)
               +sizeof(matrix *)*stmsize);
   pentries[i]->neti=Pinst->symentry;
   pentries[i]->parms=NULL;
   for (j=0;j<stmsize;j++)
      pentries[i]->pats[j] = NULL;

   rest=cdr(rest);
  }

stmP->numstminps = i;
stmP->stminps = alloc(sizeof(stmentry *)*i);
memcpy(stmP->stminps,pentries,sizeof(stmentry *)*i);

rest=netlist;
for(i=0;net=car(rest);i++)
  {
   if (i>maxstnets) abend ("stmize: too many nets");

   SetNet(net);
   pentries[i]=alloc(sizeof(stmentry)
               +sizeof(matrix *)*stmsize);
   pentries[i]->neti=Pinst->symentry;
   pentries[i]->parms=NULL;
   for (j=0;j<stmsize;j++)
      pentries[i]->pats[j] = NULL;

   rest=cdr(rest);
  }

stmP->numstmouts = i;
stmP->stmouts = alloc(sizeof(stmentry *)*i);
```

```
  memcpy(stmP->stmouts,pentries,sizeof(stmentry *)*i);

  popnet;
  popdinfo();
  return(stmP);
 }


netlists allcons(netlists to, netlists from)
 {
  netlists net;
# define maxacons 512
  int all[maxacons];
  int done[maxacons];
  netlists wholist,oldlist;
  netlists rest;
  int i,j;
  int neti;
  int lastall,lastdone;

  pushnet;

# define incon(k) k = k+1<maxacons ? k+1 : abend("allcons: overflow")

  rest = from;
  for(lastdone=-1;net=car(rest);rest=cdr(rest))
   {
    incon(lastdone);
    done[lastdone] = SetNet(net);
   }

  rest = to;
  for(lastall=-1;net=car(rest);rest=cdr(rest))
   {
    incon(lastall);
    all[lastall] = SetNet(net);
   }

  for (i=0;i<=lastall;i++)
  {
   for (j=0;j<=lastdone;j++)
     if(all[i] == done[j])
       break;
   if (j>lastdone)
    {
     neti=SetNeti(all[i]);
     incon(lastdone);
     done[lastdone]=neti;
```

```
      rest = Pinst->conamelist;
      for(;net=car(rest);rest=cdr(rest))
       {
        neti = SetNet(net);
        for (j=0;j<=lastall;j++)
         if(neti == all[j])
           break;
        if (j>lastall)
          {
           incon(lastall);
           all[lastall] = neti;
          }
       }
     }
   }

  wholist = nullist;
  for (i=0;i<=lastall;i++)
   {
    oldlist = wholist;
    wholist = li_ wholist, netname_of(all[i]) _st;
    freelist(oldlist);
   }
  popnet;
  return(wholist);
 }

int copystm(netlists listo, netlists listfrom)
 {
  int i,j,k,ifrom;
  stm *stmto; stm *stmfrom;
  stmentry *pentfrom;
  stmentry *pento;
  netname *net;
  lists *rest;
  int copysize,patsize;

  pushdinfo("copystm ",etceterate(listo)," from ",etceterate(listfrom),_at);
  pushnet;

  if (trace)
    putout("\ncopying stm %s from %s",etceterate(listo),etceterate(listfrom));

  stmto = istm(listo);
  if (!stmto)
    abend(" %s not an stm",etceterate(listo));
  stmfrom = istm(listfrom);
  if (!stmfrom)
```

```
  abend(" %s not an stm",etceterate(listfrom));

 copysize=stmfrom->lastm+1;
 if (stmto->stmsize < copysize)
   copysize = stmto->stmsize;
 stmto->lastm = copysize-1;

 for (k=0;k<copysize;k++)
  (*(stmto->stmuses))[k]=(*(stmfrom->stmuses))[k];

 rest=stmto->listin;
 for(i=0;net=car(rest);i++)
   {
    SetNet(net);
    ifrom = whereinlist(stmfrom->listin,net);
    if (ifrom >= 0)
     {
      pento=stmto->stminps[i];
      pentfrom=stmfrom->stminps[ifrom];
      for (j=0;j<copysize;j++)
       {
        dealloc(pento->pats[j]);
        if (pentfrom->pats[j])
         {
          patsize = Pinst->outsize;
          pento->pats[j] = alloc(sizeof(matrix)*patsize);
          memcpy(pento->pats[j],pentfrom->pats[j],sizeof(matrix)*patsize);
         }
       }
      if (pentfrom->parms)
        {
         dealloc(pento->parms);
         pento->parms = alloc(sizeof(netparms)*stmto->stmsize);
         memcpy(pento->parms,pentfrom->parms,sizeof(netparms)*copysize);
        }
     }
   rest=cdr(rest);
  }
rest=stmto->listout;
for(i=0;net=car(rest);i++)
  {
   SetNet(net);
   ifrom = whereinlist(stmfrom->listout,net);
   if (ifrom >= 0)
    {
     pento=stmto->stmouts[i];
     pentfrom=stmfrom->stmouts[ifrom];
     for (j=0;j<copysize;j++)
```

```
        {
         dealloc(pento->pats[j]);
         if (pentfrom->pats[j])
           {
            patsize = Pinst->outsize;
            pento->pats[j] = alloc(sizeof(matrix)*patsize);
            memcpy(pento->pats[j],pentfrom->pats[j],sizeof(matrix)*patsize);
           }
        }
      if (pentfrom->parms)
         {
          dealloc(pento->parms);
          pento->parms = alloc(sizeof(netparms)*stmto->stmsize);
          memcpy(pento->parms,pentfrom->parms,sizeof(netparms)*copysize);
         }
      }
    rest=cdr(rest);
   }
 popnet;
 popdinfo();
 return(0);
}

# define refrestm(stmP) {if (stmP) stmP->lastm = -1;}
```

## References

[1] D. E Rumelhart,. G.E. Hinton, & R. J. Williams, Learning internal distributed processing, Cambridge, Mass: MIT Press, 1986

[2] D. Marr, Vision, W.H. Freeman, SF, 1982

[3] P. Smolensky, Tensor product variable binding and the representation of symbolic structures in connectionist networks, Artificial Intelligence, Vol 46, 1990, 159-216

[4] Stephen Grossberg & Ennio Mingolla, Computer Simulation of Neural Networks for Perceptual psychology, in S. Grossberg (editor) Neural Networks and Natural Intelligence, MIT Press, 1988, 195-211

[5] G. Edelman, Neural Darwinism, Basic Books, 1987

[6] Marvin Minsky & Seymour Papert, Perceptrons, Cambridge, Mass: MIT Press, expanded edition 1988

[7] Zenon Pylyshyn, Computation and Cognition, , Cambridge, Mass: MIT Press, 1986, 147-191

[8] Howard Gardner, The Mind's New Science, New York: Basic books, 1987, 245-253

[9] Jerry A. Fodor, The Modularity of Mind, MIT Press, 1983

[10] Marvin Minsky, The Society of Mind, Basic Books, 1987

[11] Michael S. Gazzaniga, The Social Brain, Basic Books, 1985, 4-5, 81-146

[12] Tim Shallice, From Neuropsychology to Mental Structure, Cambridge University Press, 1988, 381-404

[13] E. Kandel, J. Schwartz, Principles of Neural Science, Elsevier Science Publishing, New York, 1985

[14] S. Zeki, Functional Specialization in the Visual Cortex of the Rhesus Monkey, Nature, Vol 274: 423-428, August 1978

[15] S. Zeki, The Visual Image in Mind and Brain, Scientific American, September 1992, 69-76

[16] T. J. Sejnowsky & C. Rosenberg, NETtalk: a parallel network that can learn to read aloud, The Johns Hopkins University EE & CS technical Report, in Neurocomputing, J. Anderson &E. Rosenfeld (editors), MIT Press, 1988

[17] Massimo Piattelli-Palmarini, Editor, Language and Learning, The Debate between Jean Piaget and Noam Chomsky, Harvard University Press, Cambridge Mass., 1980.

[18] M.F. St. John & J.L. McClelland, Learning and Applying Contextual Constraints in Sentence Comprehension, in Artificial Intelligence 46, 1990, 217-257

[19] Risto Miikulainen and M.G. Dyer, Encoding Input/Output in connectionist Cognitive Systems, in Proceeding of the 1988 Connectionist Model Summer School, Morgan Kaufmann, 1989, 347-356

[20] A.L. Gorin, S.E. Levinson, A. N. Gertner and E. Goldman, Adaptive Aquisition of Language, Computer Speech and Language 5, 1991, 101-132

[21] Terry Winograd, Language as a cognitive process, Syntax, Addison-Wesley, 1983

[22] John Moyne, Undestanding Language Man or Machine, Plenum Press, New York, 1985

[23] Jeremy Peckham (editor), Recent Developments and Applications of Natural Language Processing, CP Publishing, England 1989

[24] Noam Chomsky, Aspects of the Theory of Syntax, M.I.T. Press,1965

[25] V.J. Cook, Chomsky's Universal Grammar, Basil Blackwell, Worcester, England, 1988

[26] C. L. Baker, English Syntax, M.I.T. Press, 1989

[27] Greenbaum and Quirk, A Student Grammar of the English Language, Longman, London,1990

[28] J.A. Fodor & Z.W. Pylyshyn, Connectionism and cognitive architecture, Cogntion: International Journal of Cognitive Science, vol 28, 1988

[29] M.R. Quillian, The Teachable Language Comprehender, CACM 12:459-476

[30] P.S. Churchland & T.J. Sejnowsky, The computational Brain, MIT Press, 1992

[31] A..N. Refenes, M. Azema-Barac & P.C. Trelevean, Financial Modeling Using Neural Networks, in Liddell H., Commercial Applications of Parallel Computing, UNICOM, to appear 1993

[32] Kunihiko Fukushima, Sei Miyake and Takayuki Ito, Neocognitron: a neural network model for a mechanism of visual pattern recognition, IEEE Transaction on Systems, Man, and Cybernetics SMC-13:826-834

[33] Jacob Weiss, Neural Programming, Proceeding of the International Joint Conference on Neural Networks, Baltimore 1992, Vol 1, 781-787

[34] Jacob Weiss, A Neural Program Natural Language Comprehension Model, accepted to be published by Springer -Verlag in the Proceeedings of the NATO Advanced Study Institute in Bubion Spain, 1993