1991

Algorithm animation and its application to artificial neural network learning

Walter Bubie C.

Rochester Institute of Technology
Computer Science Department

Algorithm Animation and its Application to Artificial Neural Network Learning

by

Walter C. Bubie

September 18, 1991

A thesis, submitted to the Faculty of the Computer Science Department, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Approved by:

Professor Peter G. Anderson

3 Oct 91

Dr. James Wilson

10/3/91

Dr. Walter Wolf

September 18, 1991

Algorithm Animation and its Application
to Artificial Neural Network Learning

I, Walter C. Bubie, hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

10.5.91 .

Acknowledgements

I extend my sincere thanks to Dr. James Wilson for his constant encouragement and thought provoking suggestions that helped meld the different disciplines behind this thesis. His corrections of earlier drafts are also very much appreciated.

Many thanks to Professor Peter Anderson for his constant interest in the theme of this work, his suggestions towards view improvements, and his patience.

Thanks also to Stan Caplan of the Design Resource Center/Human Factors group at the Eastman Kodak Company, for his willingness to give me more than three months off from work in order to focus on completing this thesis.

Finally, I wish to thank my lovely wife, Doris, who has known me only as a student to date. Her encouragement, patience, and love were key in getting me through the thick and thin of this work.

To my loving *μπουμπουκι*, my *αγαπη*...

Abstract

Algorithm animation is a means of exploring the dynamic behavior of algorithms using computer-generated graphics to represent algorithm data and operations. Research in this field has focused on the architecture of flexible environments for exploring small, complex algorithms for data structure manipulation. This thesis describes a project examining two relatively unexplored aspects of algorithm animation: issues of view design effectiveness and its application to a different type of algorithm, namely back-propagation artificial neural network learning. The work entailed developing a framework for profiling views according to attributes such as symmetry, regularity, complexity, etc. This framework was based on current research in graphical data analysis and perception and served as a means of informally evaluating the effectiveness of certain design attributes. Three animated views were developed within the framework, together with a prototype algorithm animation system to "run" each view and provide the user/viewer interactive control of both the learning process and the animation. Three simple artificial neural network classifiers were studied through nine structured investigations. These investigations explored various issues raised at the project outset. Findings from these investigations indicate that animated views can portray algorithm behaviors such as convergence, feature extraction, and oscillatory behavior at the onset of learning. The prototype algorithm animation system design satisfied the initial requirements of extensibility and end-user run-time control. The degree to which a view is informative was found to depend on the combined view design and the algorithm variables portrayed. Strengths and weaknesses of the view design framework were identified. Suggested improvements to the design framework, view designs and algorithm system architecture are described in the context of future work.

CR Categories and Subject Descriptors: I.1.2 [Algebraic Manipulation]: Algorithms–*Analysis of Algorithms;* I.2.6 [Artificial Intelligence]: Learning–*Connectionism and Neural Nets;* I.5.2 [Pattern Recognition]: Design Methodology–*Classifier Design and Evaluation;* I.3.6 [Computer Graphics]: Methodology and Techniques; I.6.8 [Simulation and Modeling]: Types of Simulation–*Animation;*

Additional Keywords: Algorithm Animation, Graphical Data Analysis, Interactive Program Visualization.

General Terms: Algorithms, Design, Experimentation.

Table of Contents

1

Introduction

1.1 The Problem

Algorithm animation is a visualization technique for exploring the dynamic behavior of algorithms using computer generated graphics to represent algorithm generated data and operations. Exploring algorithms this way can potentially provide the viewer with insights into the behavior of the algorithm that are not possible by conventional means of reading values listed at key junctures of the program. Algorithm animation has been used to explore and teach a variety of fundamental algorithms based on sorting and searching, and has proven useful in algorithm optimization, debugging and general research [Brow88a]. The potential benefits of algorithm animation have resulted in the development of several general purpose prototype systems aimed at facilitating animated view construction by the end-user [Hyrs87], [Bent87], [Stas90].

One area relatively unexplored by algorithm animation is artificial neural networks (ANN). ANN learning is the process of simulating the way interconnected brain neurons learn to recognize a specific class of patterns, generally using simplified mathematical models of brain structures. This is a subject of intense interest to computer scientists, neurophysiologist, and a variety of other disciplines, as indicated by the published literature describing new algorithms and applications. This research is providing solutions to problems that were incompletely solved for the last twenty years.

Much of the interest is also attributable to the curiosity of how and what ANNs learn [Tour89]. Unlike expert systems, an ANN cannot provide a trace of its reasoning. The nearest analog of knowledge in an ANN, compared to an expert system, is encoded in the weights of its interconnections. These are simply real numbers, which out of some context are usually meaningless. Understanding such ANN internals typically provides the key towards the development of optimized or new learning algorithms and architectures. Reaching an

understanding appears to require a combination of penetrating mathematical analysis and trial and error, "brute force" experimentation.

ANN training[1] can be described as "self-programming." To explore the inner structure of ANNs, many ANN researchers use some form of static visualization, usually depicting the system's state at selected stages. In contrast, a conventional program is typically developed with the aid of an interactive, *dynamic* debugger: a tool for confirming, in a stepwise fashion, that the program behaves as expected. Since an ANN only becomes useful when trained successfully, it seems reasonable that an analogous tool can be of benefit for ANN development. It is proposed here, that algorithm animation, by providing interactive and dynamic graphical exploration of ANN learning, is such a tool.

A key issue not yet addressed in the published literature on algorithm animation are the elements that make for an "effective view," i.e., a view that triggers insight about the algorithm in the viewer's mind. According to the published literature, the way around this issue is to iterate on the design of an animated view, for a particular audience, until a "good design" is found according to informal criteria of the designer [Brow85], [Stas90]. Although it can be postulated from reported experiences with systems such as BALSA [Sedg84], [Brow88a], Tango [Stas90], and Animus [Duis86] that this approach enhances the viewer's understanding, designers could benefit – especially if they are the end-users – from design guidelines that maximize the probability of a successful view. Similarly, there is a need for a framework for describing basic elements of algorithm animation views in a way that the effectiveness of different views of the same algorithm can be evaluated.

This thesis describes work that couples algorithm animation of back-propagation learning with elements of "good form" and methods of multidimensional data display for view design. The contributions of the work are:

- a framework for informally determining the effectiveness of algorithm animation views for neural network learning
- an extensible prototype system for exploring back-propagation and potentially other learning algorithms through different dynamic views.

---

1 Although the normal complement to "learn" is "teach," "train" (and its tenses) is used in reference to ANNs.

1.2 Thesis Outline

The first part of Section 2 is a general discussion on visualization and more specifically on algorithm animation. The second part is an overview of ANNs, with an emphasis on layered network architectures that use the back-propagation learning algorithm. Section 3 is a review of related work: principles of good design according to Gestalt Psychology and statistical data analysis, algorithm animation systems, and ANN systems providing visualization methods. Section 4 is the project description: the objectives and approach. In Section 5 the NetViz system architecture is described, a prototype algorithm animation system for back-propagation, created for this research. Section 6 is a discussion of observations and findings made during a set of specific investigations. Finally, conclusions and suggestions for future research are discussed in Section 7.

Following Section 7 are the references, a glossary, and appendices. The glossary contains terms specific to the topics of this work; their first occurrence in the text is generally depicted in italics. Three appendices include details of the back-propagation algorithm, details of the NetViz architecture and a NetViz user's guide.

1.3 Restrictions and Limitations

It is not the purpose of this project to formally validate the value, or lack thereof, of using dynamic visualization to gain insight about ANN learning. Rather, the purpose is to establish an approach towards formal experiments, a system for future work, and initial empirical guidelines for designing good views.

A key assumption is that the user of NetViz has a prior understanding of artificial neural networks, in particular, multi-layered topologies that use the back-propagation learning algorithm. Furthermore, the new user is not expected to automatically be able to interpret visualizations, but through enough experimentation with the system, it is expected that the user will develop an appropriate "visual vocabulary" of meaningful images.

The NetViz system is a prototype, tailored to dynamically visualize ANN learning. Although designed to easily annex other neural network paradigms as well as other views, in its current form it can only visualize the back-propagation

learning algorithm, using three different view designs. The prototype is designed to be easily transported to other graphics-based platforms; however, in its current form it is dependent on Digital Equipment Corporation's (DEC) kernel-based VAX Workstation Software (VWS).

Finally, the classification ANNs investigated in this project are cases with known solutions that are used to demonstrate the animated views of the system. Finding optimal architectures, optimal learning parameters, etc. for these networks is beyond the project's scope.

2

Background

2.1 Visualization

2.1.1 Program and Data Visualization

Today's graphics-capable computers (PCs to supermini workstations) are important tools for visualizing large volumes of multidimensional data. Two terms, *program visualization* and *data visualization*, describe a rapidly advancing domain of software and methods specific to computer-based visualization.

Program visualization (PV) is the use of computer generated graphics to map data produced by one or more linked algorithms that describe a known model with built-in bounds [McCor87], [Neil89], [Wrig90]. Data visualization (DV), generally employing super/super-minicomputers, is the application of computer graphics, using data originating from sensors setup to "watch" phenomena for which a modeling algorithm is unknown or difficult to formulate. DV also encompasses visualizations of data produced as a "by product" of an applied algorithm, as in the case of finite element analysis of a modeled object.

The more familiar examples of PV application are those created using high end/super mini computers, for studying turbulent flow and vortex formation [Meir91], collision of a star with a black hole [Dunc90] and visualizing chaotic networks [Pick88a]. Computer-aided tomography (CAT) volume rendering [Fuch89], [Schw90] and meteorology [Hibb89] are common examples of data visualization.

PV is also pursued on less powerful, graphics-based PC/workstations, involving correspondingly simpler problems, and the use of simpler, two-dimensional abstract graphics. Myers [Myer86] defines a two-dimensional taxonomy of reported PV systems: one axis for whether *code* or *data* is graphically depicted and the other axis for whether the display is *static* or *dynamic*. Systems fitting all quadrants (and some bordering them) of this taxonomy appear in the published

literature: PegaSys [Mori85] and SEE [Baec86] are systems depicting code using special graphics and typography (static graphics); the PV Prototype [Hero85] and PECAN [Reis85] are systems supporting code development and debugging using dynamic code visualization; INCENSE [Myer83] uses static graphics to aid in debugging complex data structures; and BALSA [Brow88a], Animus [Duis86], and KAESTLE [Boec86] are systems for visualizing dynamic data.

2.1.2 Algorithm Animation

More research attention appears to be directed towards the PV domain of dynamic data , more commonly referred to as *algorithm animation* (AA). Algorithm animation displays are dynamic views showing a program's fundamental *operations* in conjunction with views of changing program variables. Operations include transformations and accesses to data and to a lesser extent, flow-of-control. In most AA systems, data structures and program operations of "interest" are mapped to graphical attributes such as location (in two dimensions), line weight, symbol size, color, etc. In addition, special statements are placed in the algorithm code, preceding and/or following algorithm statements that trigger changes to the graphic attributes according to updates in the data structures of interest. Thus, during the normal execution of the algorithm, the rapid sequence of changing graphics appear as an animation.

Algorithm animation was not originally computer-based; pioneering examples are elaborately produced films [Baec81] [Boot75]. However, with computer-based AA, users are potentially active viewers, able to control the temporal flow of the animation, the mappings of graphic elements to data structures, and the assignments of graphic attributes.

The predominant application of AA has been in the instruction of introductory programming, and algorithms and data structures courses [Sedg84], [Gian86], [Raml85]. It has also been applied in research and analysis of algorithms and processes, as described in [Lond85], [Duis86], and [Brow88]. Brown [Brow88a] further suggests the use of AA generated images as "technical drawings" of data structures, that can optimally illustrate desired algorithm properties with greater accuracy and ease than if created manually. This capability is in fact provided by a suite of AA tools [Bent87].

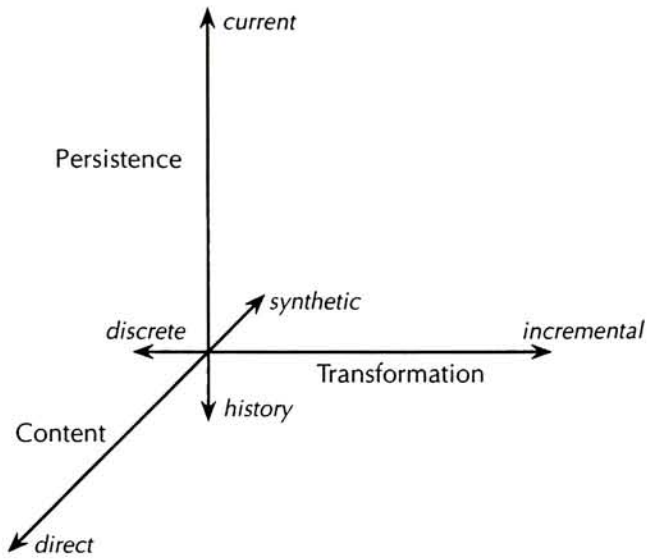


Figure 2-1. A taxonomy for algorithm animation systems.

Brown [Brow88] defines a cubic taxonomy of prevalent displays used in AA systems, as a basis for automatic algorithm annotation. The three axes describe the attributes: *content, transformation,* and *persistence* (Figure 2-1). At opposite ends on the content axis are *direct* and *synthetic*. Direct displays are isomorphic to the data structures depicted; a view of a horizontal series of bars at different heights mapping the contents of an array of values is an example. Synthetic displays are abstractions usually of algorithm operations, or of data that is a "by product." An example is a histogram showing the number of times key functions of an algorithm were executed (an execution profile).

The transformation axis describes the component of animation ranging from *discrete* to *incremental* displays. In discrete displays the animation is often perceived as uneven, reflecting a direct temporal mapping of algorithm data. Incremental displays embody smoothed transitions, either by some form of artificial interpolation of the visualized data, or by the nature of the algorithm. For example, Figure 2-2 as a discrete display would animate the "exchange" event of sorting, by the sudden swap of two highlighted sticks. As an incremental display, the two sticks would appear to slide across the view towards each others new location.

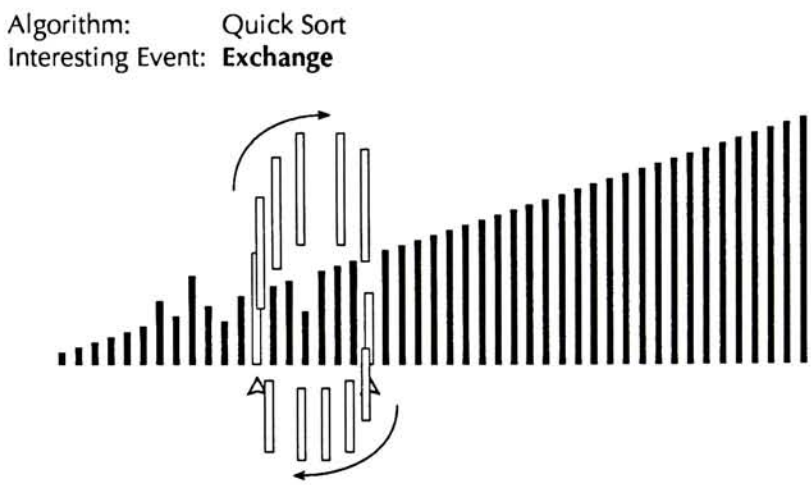


Figure 2-2. An incremental view of a sorting algorithm ("sticks" view) of overlapping update frames

The final axis, persistence, ranges from *current* and *history*. Displays characterized as current show only the most recent data values and therefore project the truer form of animation. Displays that are historical maintain in view some number of sequential *update-frames*. More specifically, in a current display each update-frame is rendered over the top of its predecessor (thereby erasing it), and in some forms of historical views, each update-frame is rendered adjacent to its predecessor, effectively forming a time-series plot.

Baecker characterizes the domain of AA as the combined use of "the crafts of graphic design, typography, animation and cinematography to enhance the presentation and understanding of computer programs..." [Baec86:p325]. Asserting that "presentation is enhanced" over the usual textual form, is similar to the justification for the use of graphic analysis of statistical data. However, the assertion that understanding is enhanced is not well verified, since little testing has been done and published results are still preliminary [Gian86], [Cros89]. Brown describes rewarding experiences with BALSA systems used over six years as a lecture aid and laboratory tool. For example, explorations of dynamic Huffman trees led to an improved variation of the algorithm. Similarly, a variation of Shellsort was "discovered" during observations of concurrent animations of several sorting algorithms [Brow88a]. Although informal

evidence, it may be postulated that algorithm animation enhances understanding; however, it is also possible that these claims are confounded by powerful and useful system features, overly "tailored" animated presentations, and the fact that the problems explored are of a similar class (typically algorithms for data structures taught in an introductory Computer Science course).

Some of the general difficulties of AA (and visualization in general) are [Brow88]:

- *Choosing an informative view* — especially when using an animated view as a communication vehicle (e.g., a classroom). The conceptual view of the algorithm as pictured by the creator may not coincide with the picture (if one even exists) in the viewer's mind. Complicating a view's implementation is the need to know during view design, the behavior and potential range of values for each data variable.
- *Capturing operations* — which is the basis of AA; algorithm operations do not necessarily correspond to each access or modification of the algorithm's data structures. Accessing a particular variable has different "meanings" at different locations in the code.
- *Real-time performance* — can suffer considerably from simply monitoring algorithm data structures, and computing synthetic view data.

2.2 Artificial Neural Network Back-propagation Learning

2.2.1 A Brief Overview of Artificial Neural Networks

ANNs are named after the networks of nerve cells in the brain. Although lacking in many of the operational details of natural neural networks, the computing models that have been developed and that are being researched form a stimulating domain of parallel computing. ANN are good at pattern matching, pattern completion, and pattern classification. They also contribute to the understanding of neurophysiological phenomena, as reported in [Gros85], [McCle86], and [Zips86b]

ANN systems are comprised of the following basic elements [Rume86a]:

- A set of processing units (or nodes)
- Weighted connections between units
- A propagation rule
- A state of activation
- An activation rule

- An output function for each unit
- A learning rule
- An operating environment

The relationship among these elements is illustrated in Figure 2-3. Given the parallelism of network processes (i.e., learning and recall), it is convenient to describe the network *state* at a given point in time. The set of *activation* values in each of the processing units best represents a network state. Activation values change over time according to the unit's output function, its propagation, and activation rules. Typically, the output function and activation rules are expressed in some form of thresholding function, which endows units with excitory, inhibitory, and neutral properties.

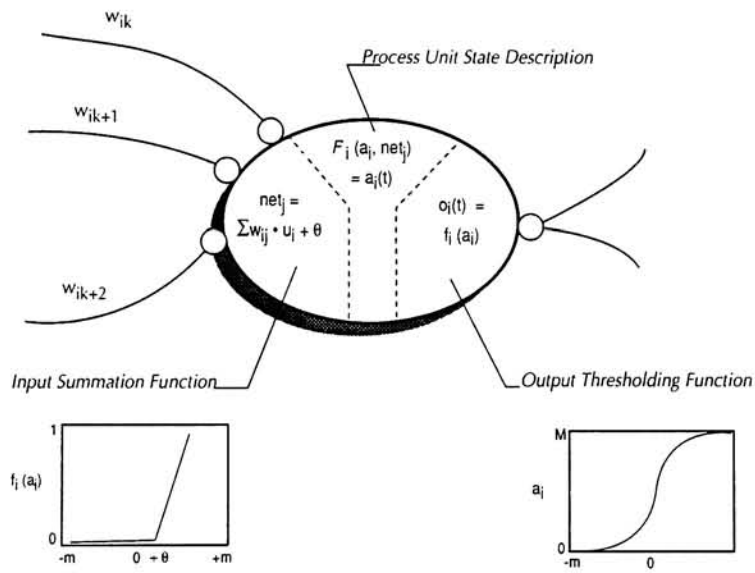


Figure 2-3. An artificial neuron.

A *propagation rule* determines the effect of the network connected to a particular unit. This involves the pattern of connections between units, which at any time constitutes what the system has learned. Learning is fundamental in neural systems; through the process of repeatedly presenting *input patterns* and, in certain types of networks, associated target patterns, the *connection weights* are adjusted. Connection weights in effect modify the connectivity pattern of the

network. Negative weights contribute inhibitory effects to the receiving processing unit; positive weights contribute excitory effects. Weights near zero effectively "shut off" the presence of the connection between two units. Over time, therefore, the network pattern becomes "specialized" to the input patterns presented up to some moment. In a way, a form of knowledge develops.

Many learning rules are descendants of Hebbian learning, modified to overcome limitations caused by non-orthogonal input patterns and the lack of error correction mechanisms. Hebbian learning is defined as [Rume86a:p36]:

> Adjust the connection strength between units A and B in proportion to the product of their simultaneous activation.

The variety of *architectures* based on connection rules, propagation rules, and learning rules that have been explored can be taxonomized in several ways. One way is shown in Figure 2-4. This taxonomy is first divided between two general network topologies: fully interconnected, single "layer" and layered networks. Each topology category is further divided by the type of learning rule: *unsupervised* or *supervised* learning. Networks that learn unsupervised are presented training input patterns without associated "correct answers" (target patterns). Networks that learn supervised do so by being presented the target pattern associated with each input pattern presented. In the category of fully interconnected topologies, to this author's knowledge, only unsupervised learning models have been explored. Furthermore, what learning affects, differs among the predominant network types in this category: in Hopfield networks (and its derivatives) learning results from adjustments to activation values in processing units; in a Carpenter/Grossberg network the connection weights are adjusted. Networks of both learning models exist for layered topologies. Only weights are adjusted in network learning, under this category.

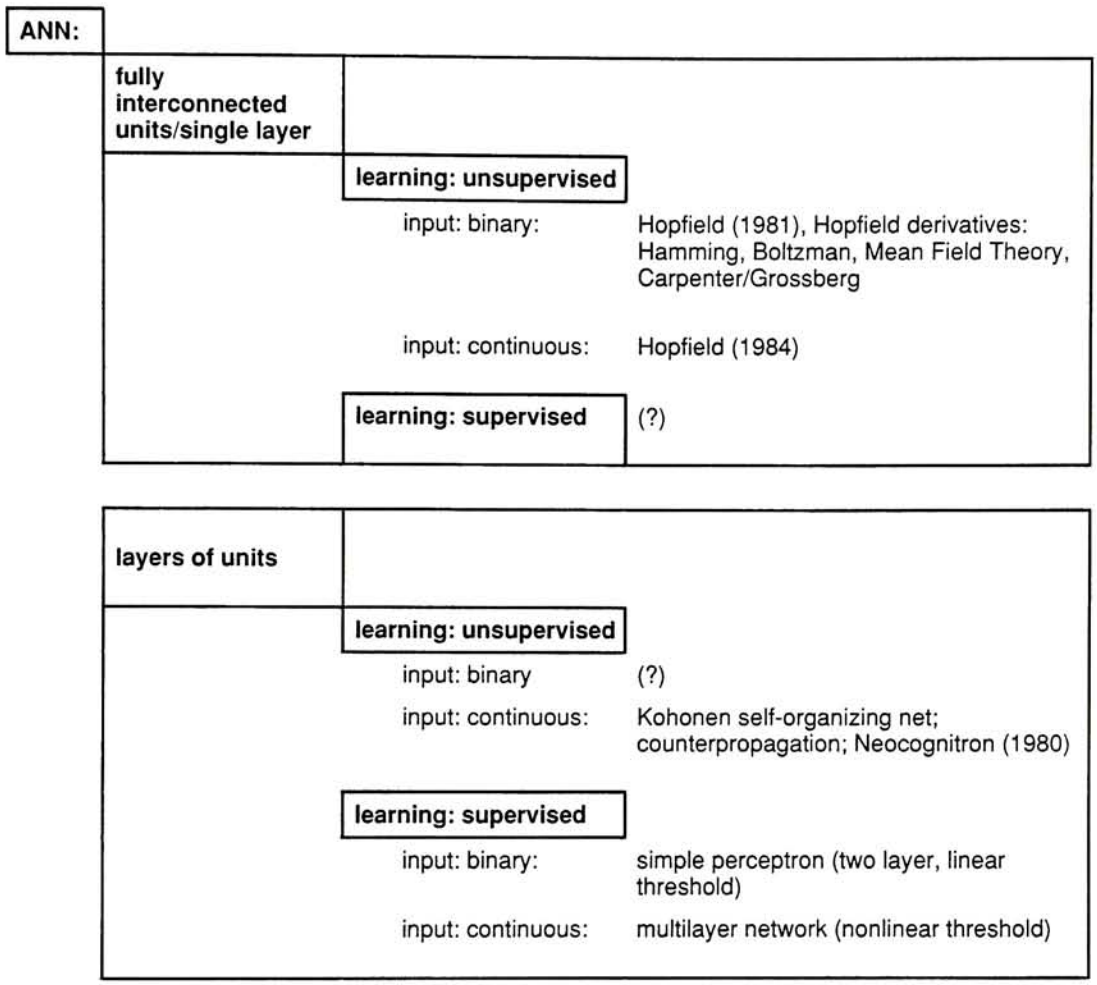


Figure 2-4. A taxonomy of artificial neural networks.

Rumelhart provides a more general classification of common learning models [Rume86a:p54]: *associative learning* and *regularity discovery*. Associative learning networks learn to produce a particular activation pattern on one set of nodes whenever another, associated pattern is presented on another set. In general, such networks have fully connected topologies, designed to store relationships between two patterns in a distributed form among the non-input units[1]. Regularity discovery networks learn to respond to distinctive (and possibly subtle) features in the input patterns. Networks of this type in general have

1 *Non-input* is used here to mean essentially hidden units. However, in auto-associative Hopfield networks input units may be indistinguishable from output units.

layered topologies in which the middle layer units store relationships of singular input patterns again in a distributed form.

2.2.2 Multilayer Networks and Back-propagation Learning

The Generalized Delta Rule

A learning algorithm of interest and the choice for this project is the back-error propagation algorithm (or more commonly, *back-propagation*) [Rume86a] [Werb74: cited in Dayh90]. The associated network topology has three layers: Processing units are classified as *visible* (at the input and output layers), and *hidden*. Unit connections link the input units to the hidden units and the hidden units to the output units[1] as illustrated in Figure 2-5. Intra-layer connections are not defined. The model is a type of regularity discovery which can be trained to compute arbitrary input/output functions.

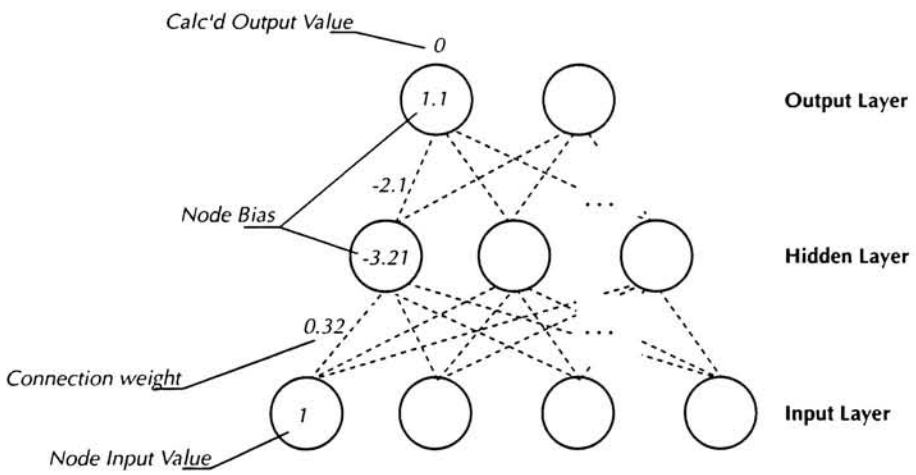


Figure 2-5. Common three layer ANN representation.

The back-propagation algorithm is a two-phase algorithm. The first phase entails the forward propagation of input patterns presented at the input layer, with

1 Certain optimized network architectures of this type can have input units connected directly to output units.

activation levels calculated at successive hidden layers. Activation is calculated for every processing unit by applying a *sigmoid* (also referred to as semi-linear) thresholding function to its summed inputs. At the output layer, activations of all output units constitute the computed output for the presented pattern.

The second, back-propagation phase begins with the comparison of the calculated output and target output associated with the input pattern. The difference, or error, forms the basis for adjusting the weights, starting with those weights between the hidden-output layers and ending with those between the input and (first) hidden layer.

The objective of the back-propagation algorithm is to adjust the system weights in a way that minimizes the total system error between calculated and target output patterns. From a simplified perspective, the system error, with respect to each weight in the system, can be expressed as a paraboloid function; by adjusting the weight in proportion to the negative of the derivative of the error, the result is the descent along the parabolic error gradient. Since there are as many error gradients as there are weights in the system, the back-propagation algorithm uses a "generalized" gradient descent or *delta rule* to compute the amount by which to adjust all weights in the system.

An essential element of the generalized delta rule is that the derivative of the activation function exists. The requirement then is a nonlinear, continuous activation function which is differentiable. A sigmoid function, shown in Figure 2-6(a) provides this requirement and is typically used in the back-propagation algorithm. Since it is continuous, it can also produce undesired activation levels that straddle the "on" and "off" states. However, the derivative of the function, shown in Figure 2-6(b), adjusts those weights entering the processing unit the most, so that subsequent activations are either closer to zero or one. The reader is referred to Appendix A for a more formal description of the error adjustment algorithm.

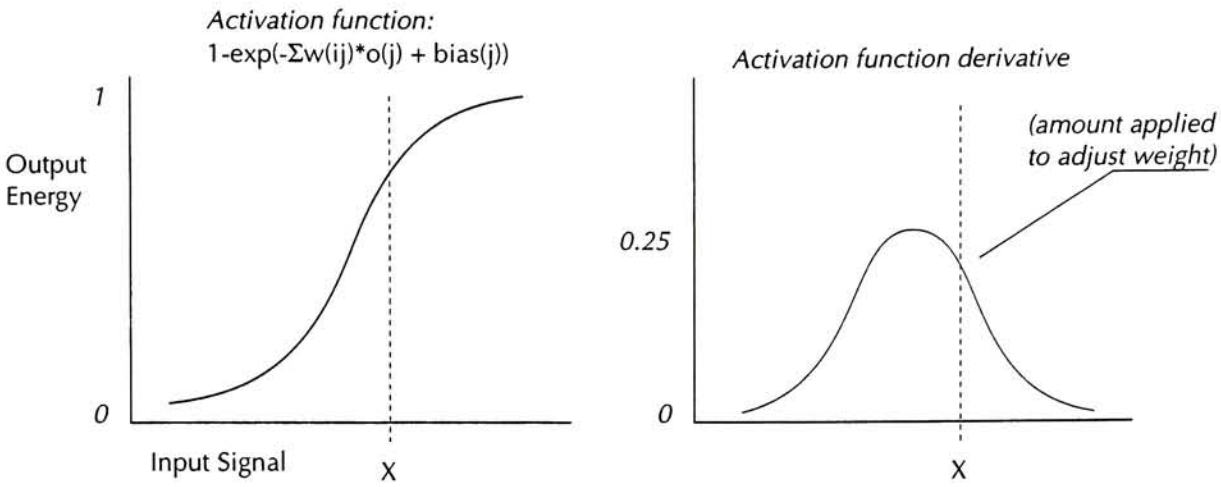


Figure 2-6(a). Sigmoid activation function. (b). Activation function derivative.

Often employed in the back-propagation algorithm is a *bias* term for each hidden layer and output layer unit. Biases are effectively incoming weights from a unit with a constant activation value of one. During the back-propagation phase, they are adjusted the same way that weights are. They provide a constant term in the weighted sum of each in-feeding unit. Computationally, each bias essentially translates the sigmoid function of the particular unit, thereby adjusting the threshold effect (Figure 2-7). The global effect of biases is usually an improvement in learning by the network.

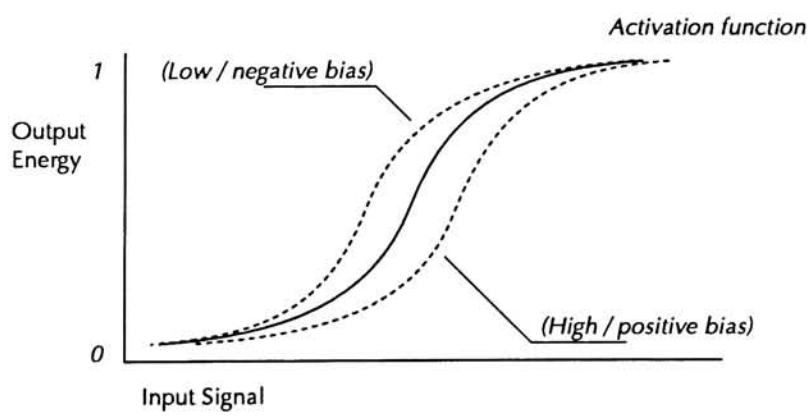


Figure 2-7. The effect of the bias term.

Training a network begins by assigning small random values to weights and biases (e.g., ±0.5). The set of training and associated target patterns are then repeatedly presented, the error for the current pattern computed, and system weights adjusted so that the total system error is reduced on each subsequent iteration. This reduction of the system error is described as *convergence*. Training is stopped when the total error reaches an *error criterion*, a target minimum value, at which point the system correctly classifies all training patterns and usually previously unpresented patterns.

The generalized delta rule process conceptually involves the *error point* traversing a rough error surface in *weight-space*. The error point is conceptually the system error at the network's present state. In certain circumstances the error point may become "stuck" in a surface pocket that is above the error criterion; instead the network has reached a *local minimum*. Convergence is conceptually the reaching of a *global minimum*.

The probability of becoming stuck in local minima depends on the *learning rate*, the *momentum constant*, and the initial weights set prior to training[1]. The learning rate conceptually describes the inherent energy of the error point; at too low a value the error point can be thwarted from reaching a global minimum by the roughness of the surface, and too high a value can cause it to over shoot a path to a global minimum. The momentum constant conceptually imparts an inertia effect to the error point, controlling its direction and thereby enabling training at higher (sometimes double) learning rates. Finally, starting network training with small random weights ensures the probability of reaching a global minimum because of *symmetry breaking*. A successfully trained system generally has unequal weights, which may be difficult to form if initial weights are equal. (Consider: the delta weight computed for a particular output unit is added to each of the weights leading into that output unit.)

Strengths and Limitations of Back-propagation

The predominant strength of back-propagation learning is its relative general-purposeness. Networks can be configured to learn a variety of pattern-mapping problems, for example, text-to-phoneme conversion rules [Senj87],

1 Back-propagation implementations often enable users to define training schedules whereby the learning rate is changed mid-course for the entire system, or potentially for specific units [Rume86c].

signal noise filtering [Klim88], and hand printed numeral recognition [Burr86]. The utility of this ANN paradigm is enhanced (or hampered) by its flexible architecture, i.e., choices for number of layers, interconnections, processing rules, and learning constants.

Back-propagation learning's two main drawbacks are its time to train larger networks (more so with respect to training set size), and its tendency to become stuck in local minima. Usually, complex recognition problems (i.e., a problem with many decision planes) require sets with a huge number of training patterns in order to learn well enough to generalize, all of which must be presented hundreds to thousands of times. Also affecting training time is each additional input and hidden unit which increases the number of weights by the number of units in each layer, all of which need to be adjusted. The use of adjustable biases is one method of minimizing the occurrence of local minima. Other solutions that address both drawbacks are preprocessing of training patterns, subdividing the input layer to create diameter-limited receptive fields [Rume86b: p348] (which reduces the number of input-to-hidden layer weights), and the use of new training methods [Jaco88], [Cail88].

3

Related Work

This chapter is a review of published work that influenced this project, including: research on effective view design; noteworthy algorithm systems; and visualizations of ANNs. The review for effective view design encompasses principles of "good form," from Gestalt Psychology and from the graphical display of data. The algorithm animation systems reviewed are selected according to the unique contributions each makes to the field. Finally, the review of ANN visualization covers the predominant static views found in the literature and systems capable of limited dynamic algorithm variable visualization. To this author's knowledge, there are only two, unpublished papers specifically investigating the dynamics of ANN learning using dynamic visualization.

3.1 Visualization and Good Design

The way a problem is represented strongly influences the way it is understood and ultimately (if ever) solved. According to Simon [Simo81: cited in Boec86], solving a problem means representing it in a way that its solution is evident. One common practice used to structure a problem and to conceptualize solutions is through visual thinking [McKim72]. Simple visual thinking involves sketching mental images of problem elements thereby creating a manipulable diagram for generating solutions. Flowcharts, statistical data plots, design renderings, and story boards are just a few examples of visual thinking.

According to several major studies on graphical information processing in humans [Kahn81], [Jule81], [Wick88], [McCor87], [Legg89], the human visual information processors are optimized for extracting large amounts of information, quickly, from non-symbolic graphics and images. The underlying reasoning is that graphics can be assimilated and processed in parallel while text requires serial and logical processing [Jule81]. In assimilating large amounts of

visual information, complex and unanticipated patterns can be recognized. Certain image designs are optimal for a given task, and targeting such displays increases the chances of gaining insight from the information.

Successful interpretation of an image (especially by the reader) partly determines its effectiveness. Effectiveness in turn depends on the viewer's *visual vocabulary* [Brow88], i.e., the set of imagery that an individual learns through experiences in their environment [Kapl82], [Berl71], and through the properties of the information being displayed.

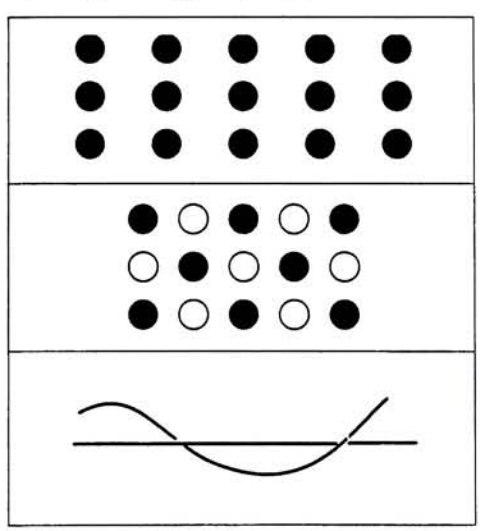
3.1.1 Gestalt Psychology

There are no simple heuristics for creating effective graphical representations. However, a basis for effectiveness seems viable through the application of *Prägnanz* or "good form." This is one of the principles of Gestalt Psychology, whose tenant is that the whole of an image imparts a different experience than its parts. *Prägnanz* describes the tendency for humans to perceive simple, symmetrical, and regular forms when only parts of the forms are displayed (i.e., "ambiguous" forms, that are overlapped by other forms). Good forms tend to be attractive, "easier to see", in the sense that their image components are not readily noticed (i.e., harder to decompose) [Hubb40], [Levi81], [Kapl87]. The basis of Prägnanz is a set of ten primary principles of perceptual grouping:

1. *Proximity* — the perception of grouping by elements that are close together.

2. *Similarity* — the perception of grouping by elements that are like each other.

3. *Continuity* — the perception of a single entity within ambiguous forms.



4. *Common fate* — the perceived grouping of image elements that appear to move in the same direction and speed.

5. *Closure* — the tendency to see closed images as unitary, wholes.

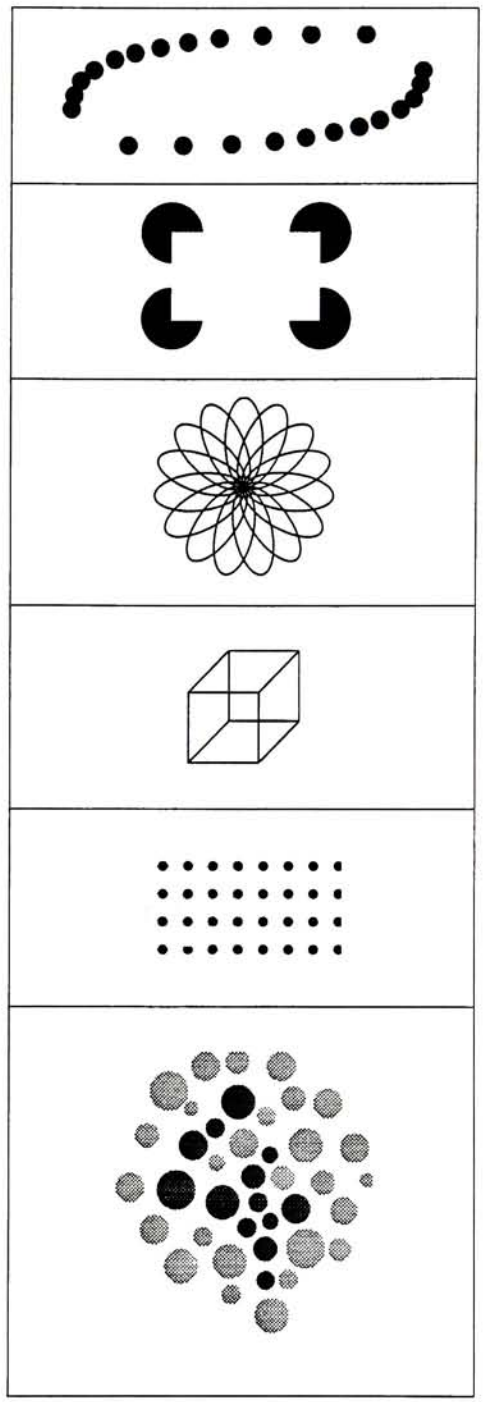
6. *Symmetry / regularity* — the attractiveness of when the left half of an image is mirrored in the right half (the vertical mirror axis is perceived the strongest of many possible axes).

7. *Connectedness* — the tendency to perceive any uniform connected region as a single unit.

8. *Common region* — the tendency of an observer to group elements that are located within the same perceived region.

9. *Complexity* — the description of an image in terms of the number of distinct form attributes (e.g., curvature, number of corners, etc.).

10. *Information content* — describes the simplicity of a figure and its "predictability; simple figures have a low information content and are more "predictable" (recognizable without seeing the details).



Connectedness (7) and *common region* (8) are two principles recently postulated [Rock90]. Rock points out that connectedness may be a fundamental property of each of the original four principles. The last two properties of information content and complexity are used to categorize the remaining six properties.

3.1.2 Elements of an Engaging Image

Extending the theory of Gestalt Psychology is the theoretical framework of Kaplan [Kapl82: cited in Leve88], regarding *informational richness* and *structure*. Images which are informationally rich tend to be engaging. Informational richness is characterized in part by a high degree of complexity in an image. Structure relates to the symmetry and regularity in an image. Images that are complex, regular and symmetric are considered to be engaging in a way that enhances comprehension; at the same time, exploring an informationally rich scene is also a non-trivial task [Leve88].

The preference for complexity (i.e., informational richness), regularity and symmetry in images is reported in studies by Hubbell [Hubb40], and Holynski [Holy85], [Holy88]. Hubbell studied the elements of good form in the modifications made to ambiguous geometric objects by aesthetically naive test subjects. Holynski and Lewis, in a series of studies, examined the relationships between viewer preference and several visual principles for a knowledge base of aesthetic criteria. In both cases, the findings that complexity, rather than simplicity, makes for an attractive image seems contradictory to the principles of Gestalt. A likely explanation comes from Hubbell's findings: the complexity in preferred images is "simplicity with differentiation." That is, from an atomistic view, an image is complex by its countable features. However, from a global perspective, as long as the features form a perceptual grouping, they then contribute to the image's differentiating, attractiveness.

Several studies have determined that images (as well as text) meaningful to the viewer are better assimilated, remembered, and later recognized [Cros89] [Chas74]. Furthermore, studies by Shneiderman [Shne82: cited in Cros89], examining the comprehension of ordered and disordered code fragments, and Rock [Rock90], examining images, show that patterns of "good form" are remembered more quickly and better than those with "bad form."

3.1.3 Graphical Data Analysis

Good form is also key in images dealing with graphical data analysis [Bert81] [Tuft90]. Graphical data displays are useful in providing insight into the

structure of analytical and physical data. Insight is gained by looking for patterns and relationships in the data; proving or disproving predicted structures; and by suggesting corrective action when used diagnostically.

Chambers et. al [Cham83] discuss principles that contribute to an effective display of quantitative information, key ones being:

- spatially (x, y) coded information is most effective
- symbol encoding by varying in shape and size is "next best"
- simple patterns (formed by the data) are perceived more quickly than complex ones
- large clusters of objects are more easily seen than smaller, isolated ones
- symmetry, in particular bilateral and circular, make a data view more engaging

The key is to use these principles to maximize a view's visual impact. Bertin [Bert81] and Chambers et. al offer these guidelines:

- *Exclude elements common to all data* — since the common, non-differentiating element(s) reduces the impact of the image. Similarly, the impact of an image's elements matches their importance in the context of the analysis.
- *Use separate representations for size and sign* — in symbol coding. If the magnitude of the mapped value is important, then it should be size-coded, and its sign non-size-coded. Alternately, the symbol size can represent its full value (i.e., smallest size: negative; largest size: positive), together with a complementing sign coding.
- *Scales varying length, size, and direction are superior* — for symbols. These attributes cause variable visibility, providing the strongest perceptual effect of value differences. However, even the optimally designed symbols depict at best approximations, because a viewer's interpretation of the symbol is relative. Therefore, different symbols should always be distinguishable.
- *Symbol component mapping can affect the overall "texture"* — of a symbol plot. The result also affects the relationships in the data portrayed. A guideline is to map related variables in component groups that will form textures that are easier to interpret.

Graphical data analysis is essentially a form of DV, and possibly PV (Section 2.1.1). However, most of the published work using PV and DV so far has focused primarily on two types of data forms: "real objects" and "scalar–valued" or "vector–valued" functions of two to four dimensions [Beck91]. The type of real objects studied, have, by their nature, a spatial structure (e.g., a 3D molecular model). Functions do not need to have spatial domains, however, most of the functions studied are given spatial structures. The graphical data display of both data forms have characteristically involved 3D volumetric rendering, with photo-realistic shading; none of the more established graphical display techniques (and generally, more analytical) seem to be used, such as multivariate scatter plots, distributions, and two-way classifications [Cham83].

Becker and Cleveland in [Beck91] propose broadening current PV and DV to include established techniques of graphical data display. Their argument is that there are many data forms that are not real objects or functions, and that many have more than four dimensions. Moreover, 3D objects (especially abstract, solid shapes) are considered more difficult to explore and interpret [Cham83], because they require the viewer to mentally "navigate around" the partial image in view. Human depth perception is less reliable (quantitatively) than spatial perception within a two-dimensional plane. In fact, knowledge about "3D things" comes from studying them from different angles.

Multivariate Data Display

The analysis of multidimensional or *multivariate* data poses interesting graphical display problems since the two-dimensions of a plotting surface are inadequate. Several techniques have been devised to display multivariate data in two dimensions:

- symbol encoding
- generalized draftsman displays
- multi-window / two-way classifications
- color coding

Symbol encoding uses a basic icon having various adjustable components to which each of the many data variables (or sources) is mapped. The icon's components may change by shape, length or orientation. Examples are: star

symbols, Kleiner-Hartigan tree symbols [Klei81: cited in Cham83], and Chernoff face symbols [Cher73]. In certain circumstances, symbols can be "plotted," i.e., two dimensions of the data set are spatially encoded, and the remaining dimensions are encoded in the multi-component symbol. Figure 3-1 illustrates the use of a star symbol to depict 12 variables of car data. The length of each ray is proportional to the variable's value.

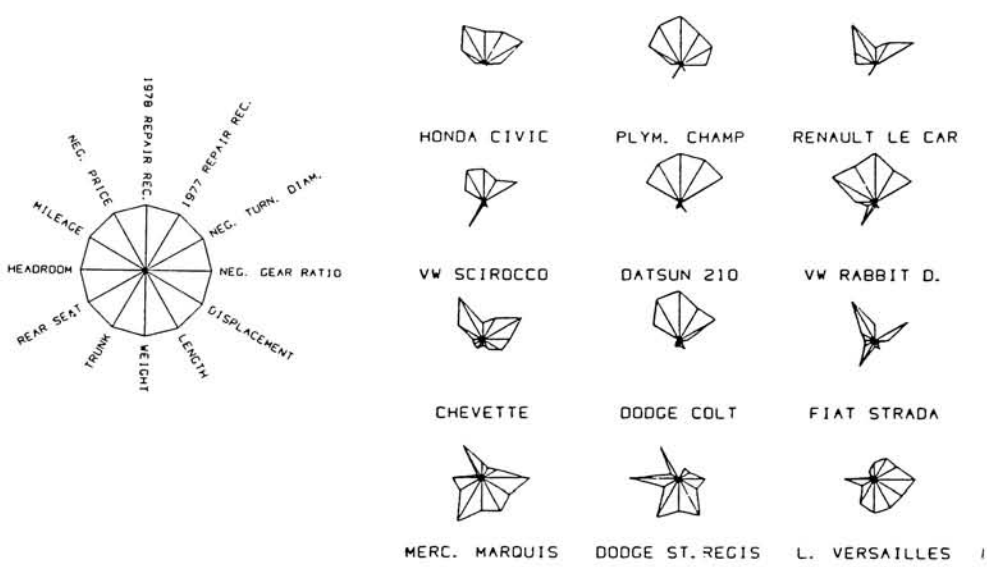


Figure 3-1. Use of a star symbol for multivariate data (from [Cham83: p161]).

Generalized draftsman's displays [Tuke81: cited in Cham83] derive their name from the top-front-side view set of 3D objects commonly used by draftsmen. However, with the "generalized" form, any pair-wise display of multiple dimensions is possible. Typically plotted in each of the views is a scatter plot of the two data dimensions defined for the particular view. An example is shown in Figure 3-2, of six pair-wise plots of four data variables.

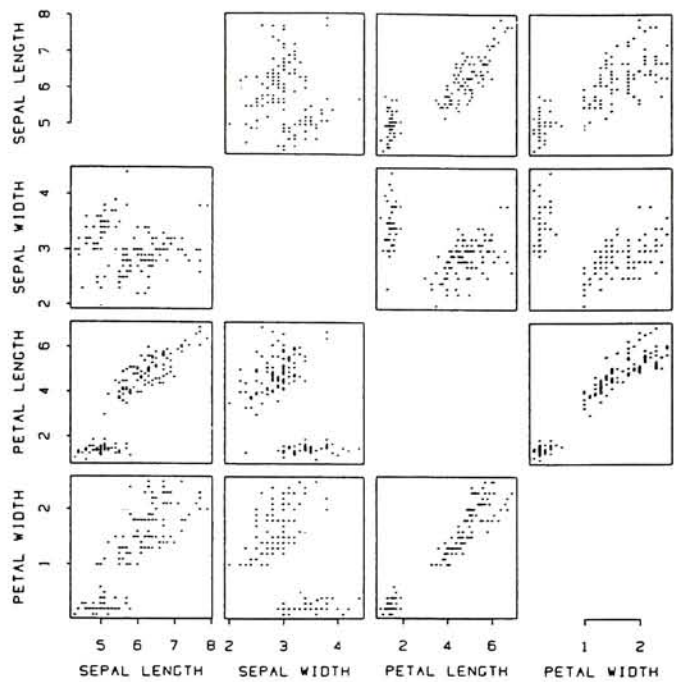


Figure 3-2. A generalized draftsman display (from [Cham83: p146]).

Multi-window/two-way classifications [Beck91], [Cham83] are useful ways to display a multivariate data set having two categorical variables that are cross-classified. That is, each level of each variable appears exactly once with each level of the other. In the example multi-window display in Figure 3-3, weight and price (of cars) are the cross-classified variables. The display is constructed such that the number of rows and columns of "windows" equals the number of data levels defined for the two variables. The two variables are examined within two other data categories. Often in this display, a row of windows at the top and a column of windows on the right of the main window matrix display the aggregate of the corresponding data in the windows below and left. The top, right-most, window is a "super plot" of all data points.

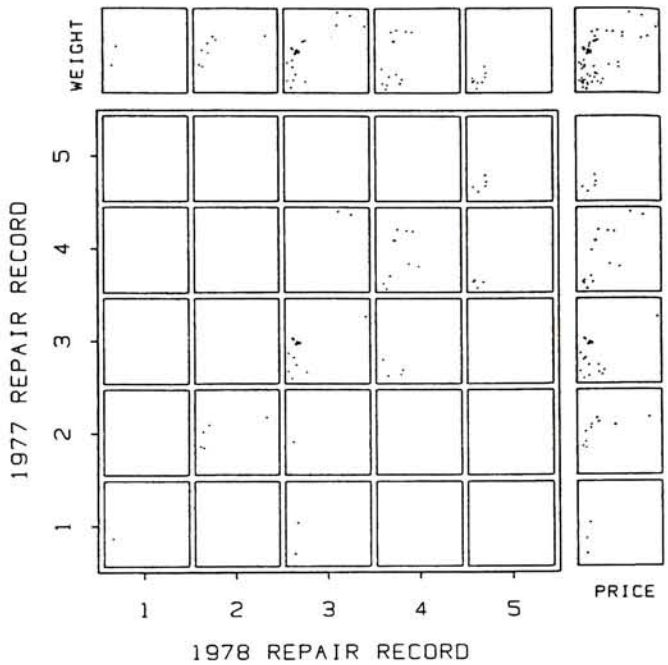


Figure 3-3. A two-way classification plot (from [Cham83: p146]).

Color Coding

Information encoding through color is next most "powerful" to geometric encoding, i.e., very large and distinctive scales can be constructed. At the same time, the use of color mapping requires being very selective. Color discrimination is constrained by the early limits of visual memory rather than the capacity to discriminate locally between tints [Tuft90]. Color is also sensitive to interactive contextual effects such as simultaneous contrast. Color acuity can vary widely within a small sample of people. Confounding the perceptual issues is the frequent problem of monitor to monitor color consistency and color precision [Marc82].

Used correctly though, color is a useful attribute for domain mapping. It is inherently multidimensional when described by either components: hue, saturation, and value (a spatial-perceptual classification) or red, green, and blue (additive classification for video displays). Devising a color map that models a continuous range can be problematic. Ware [Ware88] analyzes the problems of defining effective univariate color tables for data extraction. Univariate color

tables or "false color" tables are commonly used for geophysical maps of satellite imagery, X-radiographs, and astrophysical maps. The color in these tables is described by a continuous plane or sequence of the visible color space. For his experiment, Ware partitioned the kinds of information portrayed in univariate color tables into two categories: *metric* and *form*. Metric information denotes the quantity stored at each point on the surface (e.g., of a map). Form information denotes the shape or structure of the surface. Therefore, metric color tables reveal "fracture lines," through abrupt color density changes, and form color tables reveal differentiations of surface variations, such as cusps and gradients.

Ware conducted a series of experiments to ascertain that certain color sequences are more prone than others to induce contrast related errors in the viewer. Two forms of contrast related errors were of concern: *simultaneous contrast* and *chromatic contrast.* Simultaneous contrast is a human perception problem where one patch of color is perceptually "shifted" by another adjacent or surrounding color patch. For example, a gray patch is perceived as darker on a white background and relatively lighter on a black background. Chromatic contrast is the correlate, involving color patches.

Ware found that monotonically varying sequences along either three primary color channels[1] tend to cause the greatest errors. Examples were "gray scales," "saturation scales," such as gray-to-red, and red-to-green sequences. User testing indicated that non-monotonic sequences are resistant to contrast error effects, such as the pure hue scale (the color spectrum). Hue based sequences were found to work well as metric univariate tables; however, they failed as form-type tables. Based on tests, a table constructed by varying along the achromatic (gray scale) channel (which is suitable as a form-type table) and spiraling the hue sequence can function as a form univariate table that is resistant to contrast errors. Ultimately, the color sensitivity of the viewer and the color precision of the display determine the success of the table.

1 The three color channels are: achromatic (white...grey...black), the red-green and yellow-blue opponent channels. These channels are according to the canonical theory of cortical color processing.

3.1.4 Texture Perception and Emergent Features

Two large bodies of empirical work derived from Gestalt Psychology are on texture perception and analysis [Jule81], [Pick70], [Beck74], and "emergent feature" detection [Barn88], [Wick88], [Sand90]. The common basis of these domains is that the viewer can potentially perceive "higher level," or "integrated" information from a global view, i.e., a "whole" view of parts in the image. Both domains also seek to prove or disprove elements of Gestalt theory. Their relevance to this project is as a basis for developing engaging and informative views of multivariate data.

Texture in an image is the projection of a large number of spatially varied elements, each to some degree visible, and generally densely and evenly arrayed in a field of view. Some form of texture can be perceived in everything humans view. Furthermore, humans often rely on texture rather than outline recognition of an object. Pickett [Pick70] suggests two kinds of subjective analyses that occur in texture perception: *deliberate* and *impressionistic.* In deliberate texture analysis the observer "reads" the texture (a predominantly local perception process) and arrives at a conclusion based on a visual "glossary" of the observer's expertise. Impressionistic texture analysis occurs virtually instantly (a predominantly global perception process), probably because of a largely, unconscious knowledge of the object seen.

Pickett and Grinstein [Pick88], [Grin89] explored the visualization of multivariate data by creating synthetic textures. Their system, Exvis, maps each data record to an icon, that are then densely arranged in a matrix (using two of each data record's fields). Two general classes of data that Exvis has been used for are: *coherent* databases, containing inherently continuous or visual data (medical scans, satellite imagery; cf. "real objects"[Beck91]); and *non-coherent* databases, containing statistical data having no inherently spatial elements. Examples of texture plots published exploit the strong percept of segment angle variations in a stick-figure type icon (resembling simplified Kleiner-Hartigan tree symbols). The design and variable mapping to icons is controlled by the end-user. Each datum from a record is mapped to an icon component (width, angle, and color). Thereafter, repeated view generation and experimentation leads to the "most informative" texture image, as determined by the end-user. From examples seen by this author, textures of unknown coherent and non-coherent databases

require deliberate analysis; a known coherent database, however, is much more readily perceived.

The concept of emergent feature described here refers to the study of multicue *object displays*, based on the "principle of compatibility of proximity." [Wick88]. This principle is based in part on the Gestalt principles of proximity (in both space and time). Object displays, under certain design conditions, improve the viewer's reaction time in a "failure detection task." The basic assertion is that channels of information that must be mentally integrated must also be displayed integrated. The assertion is based on the theory established by Garner [Garn70: cited in Barn88] that predicts a "benefit for processing two integral dimensions if the information conveyed by those two dimensions is redundant, but a cost for integral dimensions if the information is uncorrelated."

Object displays that have been explored include stand-alone rectangles, where each axis represented one data channel, compound rectangles made up of four simple rectangles joined at a common corner [Barn88], a penta-prism display [Jone86: cited in Barn88], and a triangle display [Cars90: cited in Sand90]. Overall, the findings support the basic claim of a benefit for object displays, but some results [Sand90] suggest that new viewers need to be instructed on how to "read" object displays, and that some non-optimal object display designs project the important information more effectively.

3.2 Algorithm Animation Systems

3.2.1 Algorithm Movies

The earliest examples of AA are two independently produced 16mm films of computer-generated algorithm animation. The earliest film, PQ Trees [Boot75: cited in Brow88], depicts the effects of various algorithms on PQ-tree data

structures[1]. In the film, a "stick figure" data tree has its nodes and branches highlighted in colors encoded to changes taking place. Smooth animation transitions are used to show tree transformations.

The film, Sorting Out Sorting [Baec81], is a vivid display of animated sorting, using multiple views and color. Baecker used two different view templates: sticks of various heights and a scatter-like plot. In the first type of view, vertical sticks shuffle horizontally, the height of each corresponding to the value of the datum being sorted. The second view type begins with tiles in seemingly random positions within a square grid. At specific sweeps of the sorting algorithm, the tiles (each representing value (x-axis) and current location (y-axis)) are shifted and, over time, line up along the main diagonal (where x = y). The simultaneous depiction of three types of sorting algorithms in the movie – insertion, exchange, and selection sorts – in separate adjacent views of the same type, clearly exposes characteristic behaviors of each algorithm. In this author's opinion, the views in the film exhibit many principles of good form, primarily symmetry and regularity, and this makes them attractive and effective.

3.2.2 Movie/Stills

Bently and Kernighan [Bent87] have developed a suite of tools for algorithm animation and the typesetting of animation frames. Their design objective is to provide an easy to use system, where "users can animate a program in a couple of hours." The tools are designed as filters that interact on system files (and via pipes), characteristic of Unix commands/utilities, and likewise use the utilities *troff*, *pic*, and *mux*.

Creating an animation begins with the annotation of the algorithm code; examples in the Movie/Stills user's manual are coded in *awk*[2] [Aho88]. Annotations are comprised of animation control "print" statements, placed at points where "interesting" changes in data structures characterize the algorithm. When the algorithm is run, a script file is created containing the animation control statements printed by the algorithm. This script can either be viewed

1 "PQ" stands for "priority queue," a generalization of stack and queue data structures and associated algorithms.

2 Awk is a Unix utility program designed for pattern searching and processing on files.

dynamically using Movie, or processed using Stills which outputs all, or selected frames, of the animation on paper through *troff*.

Eight animation control commands are provided: circle, box, line, text, view, click, erase, and clear. The first four are graphics primitives; for each primitive additional display options can be specified, such as justification and type size for text, line styles and line weights, and fill/nofill for circles and boxes. Labels can optionally precede graphics statements; when a subsequent graphics command with the same label is encountered, the previous rendition is erased. Relative scaling of the graphics is automatically handled by Movie. The remaining four are control commands. The view command specifies that subsequent graphics statements should be rendered in the same window, or frame in a Stills output. The click command designates the updating of all prior "views." Clicks can be arbitrarily named, usually a mnemonic of the interesting event at which it was designated in the algorithm. "Erase *label* " provides controlled erasure, and clear erases all objects in the current view.

The Movie program begins by processing the designated script into an expanded internal form suitable for forward and backward display and for maximizing the speed of the animation. Through the use of a mouse (as on a SUN workstation) and pop-up menus, the end-user can: restart the animation, play it faster, slower, forward, backward, or single-stepped. Graphics attributes such as line thickness and rendering mode ("or," "xor," "and" of pixel bits) can also be controlled. A second pop-up menu provides control of views and clicks specific to the animation. View windows can be resized (resizing causes the window contents to scale uniformly) and removed. Clicks can be turned on and off.

Movie does not provide an explicit means of "racing" two or more algorithm animations concurrently as does BALSA. However, since script files are editable, "properly" designed scripts (i.e., scripts containing comparable event clusters) can be spliced and then viewed.

Bently and Kernighan admit that their animation tools provide a "bare bones" environment (which actually is Unix), and "crude output." Algorithm annotations require associated graphics commands in addition to the interesting event markers ("click *name*"), making rearrangement of "interesting events" during the experimentation phase more difficult. A remedy to this is described

in the user's guide. The most significant weakness is perhaps the requirement of an intermediate file (or data stream that would result in piping the algorithm to Movie), as it precludes the user from interacting with the algorithm and viewing the effects immediately.

In spite of such shortcomings, the tools can be quickly learned, and they provide the important aspect of interactive animation control. Furthermore, since the tools are developed as Unix filters, they provide device independence (the same script can be viewed dynamically on a 5620 terminal, SUN workstation, or typeset in a document with Stills), and simplify the substitution of intermediary tools with special purpose tools (e.g., for rendering 3D views [Bent87:p.24]).

3.2.3 ALLADIN

ALLADIN (for ALgorithm Animation Design and Description using INteraction) [Hyrs87] is a system for the teaching and research of algorithms using animation. The goal of the system is the "automatic generation of animations." The preliminary report describes the main capability for the end-user to dynamically alter the graphical layout, attributes of graphic objects, and mechanism of animation transitions (e.g., between discrete and smooth).

The ALLADIN environment is loosely coupled and modular to allow specialized components to be easily incorporated. Two suggested components are an algorithm profiler, and a shell processor. Two concepts central in ALLADIN are *animation environments* and *animation specifications*. An animation environment is an extensible set of graphical objects that are animated in a view. Each graphical object is described by data structure. An animation specification requires binding a particular algorithm variable to an instance of an appropriate graphical object thus creating a *graphical variable*. Complex algorithm data structures (such as vectors or linked lists) can be arbitrarily divided to allow binding to different types of graphical objects. Thus, for example, a binary tree can have its left nodes animated differently than its right nodes.

According to the preliminary system description, annotating the algorithm with "interesting events" requires implanting several calls to animation related procedures, along with extraneous support code (such as "if...then" constructs).

This approach encumbers the user during an annotation optimization phase, when interesting events require relocating . Furthermore, the approach precludes integrating a process for automatic algorithm code annotation, especially wherever support statements surround animation procedure calls (such as in the example in the report).

ALLADIN appears to be far from its primary goal. Its concept of an animation environment and specification are viewed as an improvement over the authors' perceived constraint in BALSA-2: that the end-user is restricted to viewing the algorithm through externally created views. However, the objective in BALSA was to absolve the student from the algorithm design task, as long as they are unfamiliar with the algorithm to begin with, and therefore, unlikely to be capable of optimally depicting the algorithm graphically. At this point, ALLADIN appears to be better suited for algorithm research instead of teaching since a student would probably not know how to best use the view editing capability.

3.2.4 BALSA

BALSA (for Brown University Algorithm Simulator and Animator) is an integrated software environment for algorithm animation [Brow85]. Its primary design objective was to provide for interactive, real-time animation. In this, and many other ways, BALSA, and later BALSA-2, shaped several elements of current algorithm animation systems.

The BALSA environment uses a client-server model that includes an interpreter, a display manager, and a shell processor. Algorithms are animated through the interpreter which can control the execution of two or more concurrent algorithms. The display manager manages the dynamic graphics displayed in multiple windows, and the shell processor interprets animation scripts and typed commands.

To animate an algorithm requires four components:

- A structured, annotated algorithm — is an algorithm in which its key functions are represented by procedure and functions calls. An annotated algorithm has additional non-algorithm related code calling out "interesting events."

- A view — is embodied as a modeler and renderer. The modeler maintains a model of the algorithm data passed through interesting events (not the algorithm's data structures) that are displayed according the graphic layout embodied in the renderer.
- An adaptor — converts interesting event messages to update messages passed to the view modeler and renderer.
- An input generator — correlates user input (messages) from the active window, with the appropriate prompt generation in the algorithm code.

The annotated algorithm and the view are two independent programs "linked" by an adaptor and input generator. Interesting event messages generated from the algorithm are directed by an event router to the adapter; mouse clicks and keyboard input associated with the active window (displaying one view) are directed through a message router to the input generator.

The concept of an *interesting event* in BALSA tends toward a monitor of "fundamental operations," rather than simply program variables. A fundamental operation is the encapsulation of accesses and transformations of algorithm variables, that drive input data towards its target configuration. An example is the "swap" operation in a Bubble sort algorithm.

The experienced BALSA user often runs an animation "live"; sequences of adapted interesting events are modeled and rendered producing a sequence of frames in a window. In the more common instructional setting, the end-user views a scripted animation, that has been carefully prepared for maximum pedagogical value. Scripts are a central element of BALSA. They are an editable hierarchical assemblage (called *chapters*) of window and control commands and recordings (called *scenes*) of prior sessions that were saved. The hierarchical design allows the user to browse through the script forward and backwards (viewing animations in reverse), and to jump between chapters.

The BALSA architecture enables the user to visualize an algorithm through several concurrent views. BALSA-2 also supports running multiple algorithms concurrently, each with their own views, so that the viewer can compare behaviors and relative algorithm performance. To achieve this, BALSA-2 schedules each algorithm, using a time-slice that ends at an interesting event. To maintain synchronicity between different algorithms, events are weighted such that "costlier" events are not rendered as frequently as events with lower costs.

A useful component of BALSA was a "code view," a window in which algorithm code could be viewed line-by-line, while at the same time one or more corresponding graphic views were animated. Multiple code views could be opened automatically for each procedure called (and closed when the procedure returns). According to Brown, code views provided a point of reference for new users of BALSA; to them the algorithm code was more meaningful (initially) than the abstract graphics. It was similarly deemed invaluable in the debugging of recursive algorithms by algorithm researchers. For unknown reasons, code views were not implemented in BALSA-2.

BALSA's strengths are clearly in its highly modular and recursive architecture (e.g., "a submodeler is a modeler ... submodelers may in turn also have submodelers" [Brow88a: p115]), its provision of sophisticated scripting, and its approach of unobtrusive algorithm annotation. This sophistication imposes a substantial learning curve for its potential user. Furthermore, the high degree of modularity inherit in the client-server approach of the original BALSA is lost in BALSA-2, a limitation imposed by its platform (Macintosh) operating system.

3.2.5 Tango

Tango (for Transition-based ANimation GeneratiOn) [Stas90] is an AA system for general purpose algorithms. Its primary design objective is the provision of smooth animation transitions that can be easily specified at a high level. This "path-transition" paradigm is influenced by the Smalltalk model-view-controller approach used for algorithm animation in [Lond85] and Animus [Duis86], and by Duisberg's assertion that showing smooth transitions can prevent viewer confusion caused by sudden drastic changes between two animation frames.

Tango supports two-dimensional color graphics in a workstation windowing system. Architecturally it is a monitor program, running asynchronously with the algorithm program, capturing view related interesting events sent from the algorithm. This client-server approach also enables the algorithm program to execute on a host different than that on which Tango is executing.

To create a Tango animation requires three activities:

1. Annotate the algorithm code with appropriate animation events.

2. Design animation scenes.
3. Specify through a control file the mapping from algorithm operations to the animation scenes.

Animation "scenes" in Tango are constructed from four basic data types: images, locations, paths, and transitions, and a set of associated operations as C functions. The path and transition elements are central to Tango's path-transition paradigm, providing very high-level animation descriptions. Paths describe a change in image attributes from one frame to the next; this includes spatial, color, and visibility changes. Smoothed paths are described through the "interpolate" operation. The actual "animation" of this is described through transitions and associated operations. "Composition," perhaps the most powerful operator, provides the specification for multiple object parallel animation.

Like BALSA, animations are driven using interesting event calls strategically dispersed in the algorithm code. These calls send packets of key algorithm variable values along with a label designating the associated animation scene. Tango dynamically maps the label to the actual scene function name; this indirection allows a user to substitute alternate animation scenes (for the same event) without recompiling.

Tango's key shortcomings are its limited image primitives of lines, rectangles and circles, and the inability to construct and manipulate compound objects as single entities. Furthermore, the system can only display one animated view at a time.

3.2.6 Discussion

Table 3(A) summarizes key elements of the reviewed AA systems:

TABLE 3(A) A Summary of AA System Features				
System	*Architecture*	*Alg. Annotation*	*Scripting*	*Primary contribution(s)*
Movie/ Stills	single program; Unix piping to animation processor	several in-algorithm-code calls to graphics routine as "interesting events"	Produced by design (except by piping) as 'off-line' animation and a means of printing animations for publication.	Simple, portable system; relatively easy to learn and use.
ALLADIN	modular, single program	several in-algorithm-code calls to graphics routine as "interesting events"	(none)	Capability to dynamically alter various view details; object-oriented animation graphic elements.
BALSA	client-server; high-level interpreter	non-intrusive "interesting event" calls	Sophisticated, editable command language	Scripting language; ability to run multiple algorithms and multiple views.
Tango	client-server; X11 based	non-intrusive "interesting event" calls	(none)	Path-transition paradigm for smooth animation.

The predominant use of AA systems is in the instruction of fundamental algorithms such as in a Computer Science data structures course. Early experiences with BALSA showed that typical users were unfamiliar with the algorithms visualized and needed to be guided through the animations or provided with more familiar complementary views. Prepared scripts, such as those that BALSA and Movie can produce, are most effective for instruction, especially when the viewer is also given the ability to control the rate and direction of the animation. Also helpful is the support of multiple views (perspectives) of the same algorithm/operation. Brown asserts that "multiple, simple views together form a gestalt of the algorithm's progress."

Animating two or more algorithms in parallel provides a means for comparing algorithm performance. BALSA provides a user-transparent mechanism for doing this that is robust enough to handle algorithms with very different interesting event points. By splicing scripts produced by similar algorithms, the Movie program can do the same, though much "hand crafting" is required to ensure view synchronicity.

A fundamental problem in algorithm animation is that many iterations are required to design effective views. Conceptualizing the general layout, choosing the appropriate data structures, deciding the mappings between them and then writing code required 15 to 25 hours by a BALSA expert for one animation of approximately 15 minutes. The cost is actually higher since typically two or more views are found to better portray the algorithm's behavior, and since a few runs of the animation would be required to determine the proper scaling coefficients.

The characteristic emphasis of depicting algorithm operations over simply data changes appears, paradoxically, not to be particularly rewarding in practice. According to Brown [Brow85], "staying close" to the data structures "tends to produce more revealing views."

A potential shortcoming of the reported AA systems is that their graphics capabilities are limited to primitives such as lines, rectangles, and circles (and text). Apparently, mechanisms for creating compound objects from such primitives are generally not provided.

Most of the general purpose AA systems tend to follow a client-server architecture, a primary reason being the flexibility of modifying algorithm code without disturbing the view code – and vice versa. An additional benefit is that views can be analyzed stand-alone, by being "fed" hand-crafted interesting events. Furthermore, in an interactive setting, a user can easily add and remove interesting event calls thereby exploring the most appropriate timing and update granularity. Principal shortcomings to this approach are the degraded execution speeds with large data sets and multiple views, the complexity of implementing interprocess communications — especially on PCs which typically do not have multitasking operating systems. A frequent side effect of attempts to increase animation speed by reducing the frequency of interprocess communication, is to transfer key algorithm data structures to the animation routines. The outcome is a duplicate algorithm in the animation system.

The primary advantage of system separation can be satisfied by judicious modular design of the system code, ensuring in particular the separation of algorithm code and data structures from the animation modules. Event passing is then achieved through shared global data structures.

3.3 Artificial Neural Network Systems Providing for Visualizations

This section reviews what has been published regarding dynamic visualization[1] of ANN learning. To this author's knowledge, none of the general purpose AA systems described in the previous section have been used to explore ANN learning algorithms. Many of the published ANN reports use some form of static visualization. Examples of some commonly encountered types are shown in Figure 3-4: Hinton maps [Hint86], feature maps [Dayh90], Kohonen maps [Dayh90], and vector maps [Dayh90]. An interesting aspect of these view forms is their suitability as dynamic views; they are basically "snap-shots," taken at selected intervals during training. Hinton and Kohonen maps are described in detail in the next section.

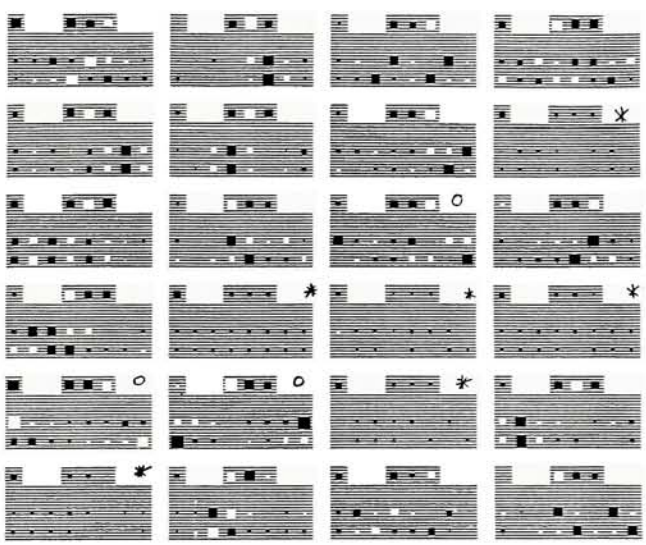


Figure 3-4(a). A Hinton map (from [Hint86:p301]).

1 The general term *dynamic visualization* is used instead of algorithm animation since the few ANN simulation systems providing dynamic visualization explore algorithm variables only.

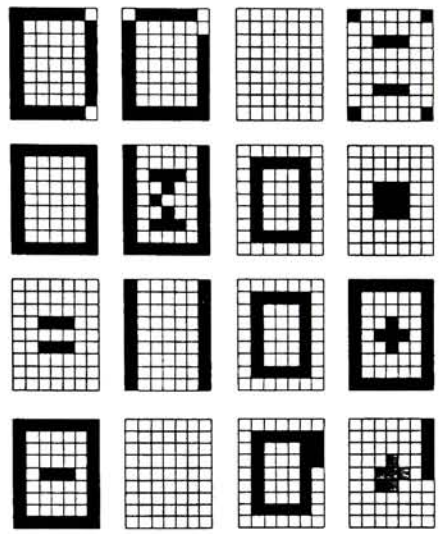


Figure 3-4(b). Hidden layer feature map (from [Dayh90:p90]).

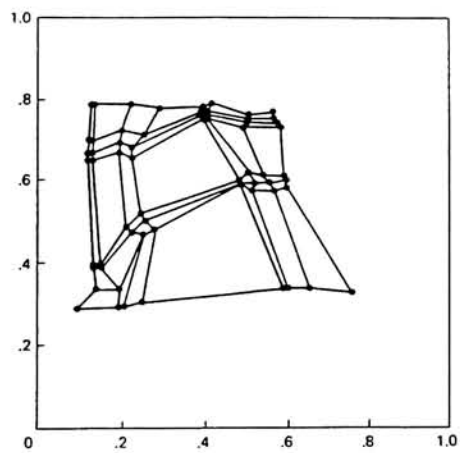


Figure 3-4(c). A 2D Kohonen map (from [Dayh90:p171]).

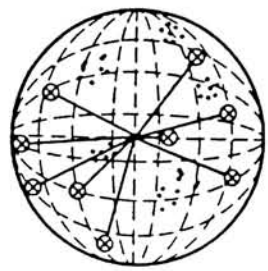


Figure 3-4(d). A vector map (from [Dayh90:p199]).

Several papers describing general purpose ANN construction and simulation tools that include some capability for dynamic visualizations have been published: The Rochester Connectionist Simulator [Feld88]; MIGRAINE [Weil88]; SunNet [Miya87]; P3 [Zips86]; and the PDP software suite [McCle88]. The latter three systems are described in more detail in the following sections; their features serving as examples of what these systems provide. The PDP software suite also served as the basis for the NetViz prototype. Two unpublished reports about dynamic visualizations experiments of ANNs have also been found and are reviewed. One of the reports is an earlier independent study conducted by the present author.

3.3.1 Hinton and Kohonen maps

A Hinton map is a compact display that shows the connection strengths of weights between layers. The example map shown in Figure 3-4(a) describes an ANN that detects a 1-bit right or left shift in the input string. It has a topology of 16 input units, 24 hidden units, and 3 output units. The 16 input units define two states: the "lower" row of eight units holds an initial binary input pattern, and the second row of eight units holds the initial binary pattern shifted either right, left, or unshifted. Each output unit represents one of three possible outcomes: shift left, no shift, shift right. In the figure, each tile containing small black and white squares represents a hidden unit. The black and white squares in the lower two rows represent the magnitudes of weights entering the hidden unit tile. The

three squares at the center-top of each tile represent the weights connecting the hidden unit tile with each of the three output units. The top-left square represents the hidden node's bias term. Black squares denote negative values and the white positive. Figure 3-4(a) is a "snap-shot" of the trained network; the hidden unit tiles constitute specialized *feature detectors* for this ANN.

Studying the map reveals that six of the hidden units are apparently "turned off," indicated by the near-zero negative weights incoming and outgoing (tiles marked with an asterisk). The remaining hidden units appear to be appropriate detectors. Tiles in which the top middle square is white (positive), consistently have squares in the lower two rows that match; these hidden units are "no-shift detectors." Tiles where the top left square is white, have squares in the lower two rows that suggest right shift detectors. Tiles suggesting left-shift detectors can also be found. Finally, certain hidden units are specialized: three tiles (marked in Figure 3-4(a) with "**o**") detect shifts that wrap-around.

A Hinton map appears to be a useful visualization tool since it is frequently encountered in the ANN literature. Although they are best suited for layered network topologies, they have been adapted for other topologies as well (e.g., Hopfield networks; refer to Section 2.2.1).

The Kohonen feature map is a visualization tool specific to Kohonen self-organizing ANNs. Basically, the Kohonen network conceptually organizes the features of a training pattern set into a topological map. This map is analogous to the classification regions defined by decision boundaries with layered, perceptron-based networks. The map corresponds to a top view of the second, "competitive" layer of the network. This layer is fully connected to a "lower" input layer and each connection is weighted. Figure 3-5 is a series of Kohonen map "snap-shots" depicting a simple ANN learning to organize a pattern set comprised of coordinate pairs that are spatially proximate. The pattern set values as a whole are a uniform random distribution between zero and one. At the start of training, the map is amorphous, as shown in Figure 3-5(a), in which the initial weight settings are equal to 0.5 (the mean of the training pattern set) ±10% random "noise." Figures 3-5(b) through (d) show the evolution of the topological map that describes the uniform distribution of the input set.

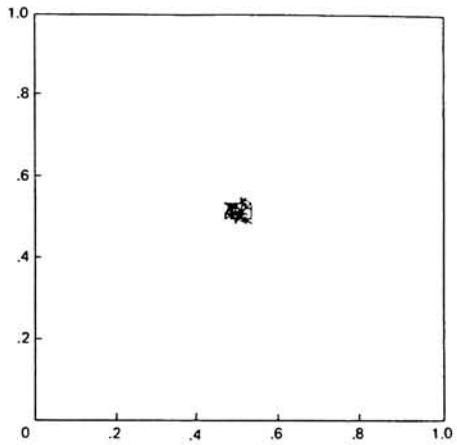
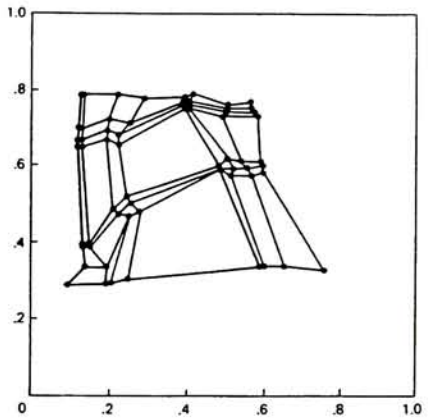
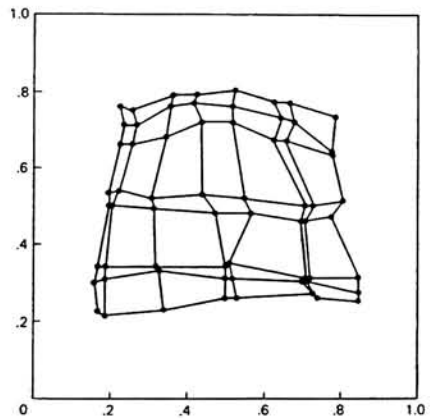


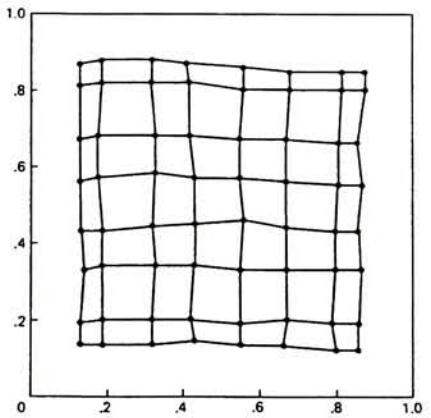
Figure 3-5(a). Starting configuration Kohonen map. (from [Dayh90: p171-2])



(b) At 1,000 iterations.



(c) At 6,000 iterations.



(d) At 20,000 iterations.

The dimensionality of a Kohonen map is equal to the number of input units. Thus a three-unit input layer would produce a 3D map. Such a map after some training resembles a bumpy weight surface of a multi-layered back-propagation network (refer to Section 2.2.2).

Both Hinton and Kohonen maps are typically encountered as static visualizations. The series in Figure 3-5 strongly suggest the possibility of animating the evolution of a Kohonen map's evolution, to observe behaviors not evident in a few stills, such as localized organizations at different periods in the training. The Kohonen map is also characteristic of the many visualization designs encountered in the published literature that are network architecture specific.

3.3.2 SunNet

SunNet is a tool for constructing, running and "looking into" ANNs on Sun workstations under the SunView windowing system [Miya87]. Its primary design objective is to allow construction of ANNs of arbitrary size and architecture using a very high-level scripting language. An additional objective is the facility for studying network states via several types of dynamic graphic views. Supported view types are: unit activation patterns, weight matrix patterns, and error/energy time-series plots.

To set up a network, the end-user describes it using SunNet's scripting language. A powerful and useful property of the system is that all components of a network are named, and so referenced in command parameters. The system also assumes various defaults in order to minimize the programming demands on the end-user.

A basic setup requires the definition of the network layers, the connection topology, the initial weights, input/target buffers, and learning rules. For special situations, the command language provides for explicit variable declarations, arithmetic statements, condition statements, procedures, and loop constructs. Vectors are a primary data type, and are supported by a simple syntax for specifying subvectors and matrix (vectors of vectors) slices.

SunNet provides an extensive set of primitives for dynamically visualizing a network: *window, graph, glist, display, dlist, plot, plotlist, screen,* and *movie.* Normally SunNet starts up in a two-paned SunView window. The top pane is a multi-line command window; the lower pane, or display area is initially empty. Issuing a window command creates one or more independent regions within the

display area in which network values can be graphically displayed. For example, "display InputHidden >1" results in the immediate display of the input-to-hidden layer weight matrix in window 1. Displays are essentially in a Hinton map format, i.e., as rectangles or squares of varying size, black squares for positive values, and white squares for negative values.

Graph provides a predefined mechanism for plotting over the number of learning epochs, the root-mean-square (RMS) error of the system, or the error for each pattern. *Plot* is analogous but allows any variable to be specified for each axis. Both plot and graph open windows implicitly.

The commands *glist, dlist,* and *plotlist* are versions of graph, display and plot that direct SunNet to dynamically update the displays of the specified variables every n epoch/frames, where n is a parameter of each command.

The final command *movie* is essentially a playback mechanism of a previously "recorded" session. *Movie take* captures the current state of the entire display area; by first using *movie frame* and the mouse, a region of the display can be specified. *Movie play* replays buffered frames. Because bitmaps are captured and replayed, SunNet restricts the number of frames captured (and held in RAM during playback) to 1024 (approx. 57 seconds at 18 fps). Buffered frames can be saved in a file to animate at a future time.

Overall, SunNet offers impressive capabilities for a public domain system (it was developed at the University of California, San Diego). Its scripting language appears to satisfy the system's objectives of extensibility. At the same time, however, its flexibility can easily lead the novice ANN explorer to design illogical (wrong) networks. The provisions for visualizing network data structures are a commendable first step, especially with respect to automatic scaling and normalizing. Its foremost shortcoming is its restriction to a type of Hinton map display for multivariate visualization. Further limiting is the absence of color, line styles, widths, and other fundamental graphic attributes. Although not purporting to be an algorithm animation environment for ANNs, its objective of flexibility falls short with respect to end-user view design.

3.3.3 P3 System

The objective of the P3 (Parallel Process Program) system, a general purpose ANN simulator, is to eliminate the tedium and time cost incurred in developing exploratory ANNs [Zips86]. In particular, P3 simplifies the construction of graphical "user interfaces" (data views), that are customized by the end-user, for the ANN to be explored.

P3 is an object-oriented LISP-based system, made up of four major components:

- *Plan language* — used to define the types of units and the unit interconnections that describe the ANN architecture. The final "object" is called a plan.
- *Method language* — used to define the internal computation behaviors of the units in the plan.
- *Constructor* — transforms the plan and set of methods into a program that simulates the network when executed.
- *Simulation environment* — provides an interactive graphics-oriented facility for observing and testing ANNs.

Typically ANN simulation environments use a simple synchronous approximation (i.e., sequentially addressing each unit in each layer) of the inherently parallel network learning and recall processes. This is a constraint imposed by uniprocessor computers. Synchronous approximations, however, can produce undesirable artifacts. Although a uniprocessor-based system, P3 allows end-users to design truer asynchronous network updating. A special control object and method compose, on each epoch cycle, a unit update schedule. Basically, the schedule describes a permuted order of unit processing.

One of the end-user efficiency aspects of P3 is the set of methods that handle access to input and target patterns (which are outside the "closed" plan). These default methods, in conjunction with a few special purpose objects, control the network (e.g., the triggering of unit updates). This approach eliminates the effort associated with constructing several special purpose routines.

Every unit in the plan has a location in Euclidean space: (x,y,z); thus a plan can also define a geometrical structure of the network (this is separable from the primary network architecture defined by the plan). This facilitates the dynamic

visualization of a P3 ANN. The geometrical structure defines unit locations that "appear...on the display as they are in the investigator's conceptual image. [Zips86: p490]" A second use of unit location information is to allow the *implicit* specification of connections between units, based on their spatial locations. This capability is useful in modeling networks of primary sensory structures (e.g., a retina).

The P3 user interface is windows-based. Windows display the network structure from the perspective of particular unit variables, e.g., activation, weight, bias, etc. The predominant representation of each network variable is a black square within a frame, essentially a type of Hinton map. Another view option is "strip charts" (time-series plots) of user selected variables. The simulation environment allows the user to interrupt training to change values or edit methods, and then continue processing from the point of interruption.

Each window is described as a simulated "instrument" (e.g., scope). Each instrument therefore, has a custom output/display format. New instruments can be added to the P3 environment and used as needed. Furthermore, multiple instances of one instrument can also be displayed.

In summary, P3 has many of the powerful construction capabilities of SunNet. It also seems to surpasses SunNet in its object-oriented visualization model and its greater repertoire of view primitives. Unfortunately, only Hinton maps, as view examples, accompany the article, and the other touted view capabilities are left to the imagination.

3.3.4 PDP Software

Rumelhart and McClelland produced with the third volume to their series: Parallel Distributed Processing, Explorations in the Microstructure of Cognition [McCle88], a suite of ANN construction and simulations programs. The book is essentially a collection of tutorials and a user's guide to the programs.

A primary objective of the program suite is to make widely available, a set of tools for casual and dedicated enthusiasts of ANN technology. The programs are written to run on two popular platforms (Intel-based PC's and Unix workstations), and to be easily transported to other platforms with C compilers.

Access to the source code also provides venturesome users the ability to experiment with new models by extending the basic programs. Finally, provisions for free-form spatial arrangement of user selectable algorithm values that can be dynamically monitored, suggests a secondary objective of rudimentary visualization.

The program suite includes simulators for a variety of ANN models, such as: interactive activation and competition model (iac; Grossberg 1976); constraint satisfaction/Boltzman machine (cs); back-propagation model (bp); and auto-associator/competitive learning model (aa). The user interface is laid out in the same way in each of the ANN programs, using direct screen location input/output provided by *curses* [1]routines. The top quarter of the screen is a command line for user input, with hierarchical, Lotus 123-style menus immediately beneath it. The remainder of the screen is used to display values of user selected algorithm variables. These variables are updated dynamically after each training iteration; some may also update during recall (e.g., current pattern name, target output, calculated output). A selective audit trail (log file) of variable updates can also be made.

The screen layout of variables and other supporting static character graphics is user definable through two types of files: *template* and *look* files. A template file has two parts: an optional layout specification, and a list of template specifications. The layout defines the placement of each variable defined by a template; the row-column position where a "tag" (i.e., a '$') is in the layout specification equates to a row-column position on the screen below the command menu. A template specification is comprised of a label, a variable name whose value appears at run-time, and other parameters, such as the "slice" of a matrix to be displayed. Each template specification in the list is sequentially associated with a tag in the layout definition (left to right; top to bottom). If a layout specification is not used, then absolute screen coordinates are required as part of each template specification. The other type of layout specification, look files, provide a user-defined method of displaying vector and matrix values. When a look file name substitutes a matrix variable name in a template specification, the look description is invoked.

---

1 Curses is a library of device independent routines for reading in and writing to the screen at specific character cells. These routines are typically part of the C language run-time definition.

The benefit of a log file is realized with the provided plotting program[1]. It provides rudimentary x-y plotting using character graphics. An example of its use is a plot of the total sum of squares error collected over the number of training epochs. The plotting program uses the first datum of each row as the abscissa of all remaining values in the line, which are plotted as ordinates. Through a separate plotting specification file, symbols can be associated with each variable.

The release of the PDP software made available tailorable ANN programs that reduced the need by casual explorers to hand craft their own software or to locate specialized ANN platforms. In fact the bare-bones visualization capabilities are characteristic of many similar (though more expensive) programs available at the time of release. With respect to dynamic visualization capabilities, the flashing of numbers updated during training is hardly adequate. The provision for customizing views is noteworthy. Overall, it manages to provide one level better information presentation over reading a log file.

3.3.5 NNANIM

An earlier project by the present author was the creation of a program specifically for animating the back-propagation learning algorithm. It was developed for the Amiga color-graphics PC [Bubi88].

NNANIM provides only one basic layout; however, various graphics attributes in the view have several possible settings, such as line width, color, and basic shape (square or circle). The layout is shown in Figure 3-6, which represents the network topology of a binary string symmetry classifier. Symbols representing processing units are of fixed size. With larger topologies the layout expands towards the right and up. The program accommodates network topologies with any number of processing units, provided that all unit symbols appear on the screen.

---

1 A separate program provided, *colex* , enables the user to subset the log file, extracting only the value stream of variables desired to be plotted.

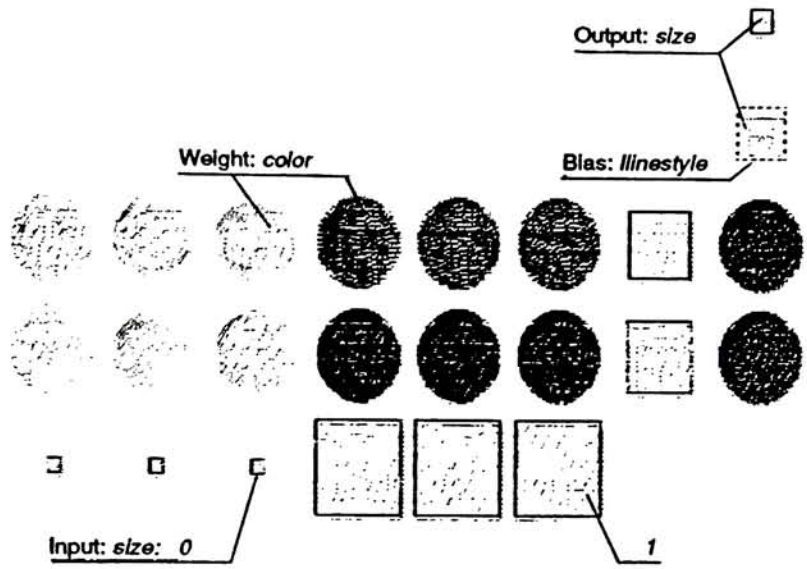


Figure 3-6. Symmetry Classifier Network in NNANIM.

The algorithm variables mapped to the graphic symbols are fixed in the system code. The user is restricted to visualizing activation values (including the input pattern) as squares, weights or delta weights as circles, and biases as line weights. The user can select between mapping activation value changes to size and weight value changes to color, or the reverse.

Depending on the specific graphical attributes defined, visualization entails discrete changes in the size and color of the unit symbols for each training iteration, at every n training cycles, where n is user defined. For weights mapped to size changes, the size of the circle depicts the magnitude of the value and an outer ring of maximum unit size depicts the sign of the value by its color (blue for positive, red for negative). Weights and activation values depicted by color changes follow one of two fixed color maps of 14 colors. One map depicts negative values in reds and positive values in blues. For activation values mapped to size, the smallest square represents a value of zero and the maximum size a value of one. Bias values affect the line weight of the border on the hidden layer and output layer units; solid lines depicted positive values, with the

"thickness" representing the value's magnitude. Negative biases appear as dashed lines; the more negative a value, the sparser the dashed pattern.

Although restrictive, the system provides a vehicle for evaluating the effectiveness of certain graphic attributes in an interactive and dynamic context of back-propagation learning. Reported observations include network convergence, formation of feature maps in the weights, and dampening effects of biases. Visualizing delta weights "swinging" between small negative and positive values suggested a view of the "error point" traversing the rough weight surface. Problems included the form of bias mapping and the use of color. The system limitation of 14 colors was a confounding element.

3.3.6 Back-Propagation Dynamics

Lehar [Leha88], in a poster session paper describes a series of informal explorations using a form of dynamic visualization. His work was motivated by earlier difficulties he encountered while debugging an ANN by reading printouts of algorithm values. A graphical view of a three-layer ANN was subsequently devised, to recognize hand printed digits (0 to 9) when trained.

The network consisted of a three layers: a 10 by 10 retinotopic[1] input layer, laid out as a single vector in the graphical view, a 100 unit hidden layer, and a 10 unit output layer (Figure 3-7). Each output node represented a single class, i.e., one of 10 digits. All layers were fully connected to the next layer "up."

Input patterns and activation values in hidden and output layer units were mapped to a grey scale, white being "active" and black "inactive." Weights, shown as the two large two-dimensional arrays, were mapped to two color scales. A grey scale was used for positive weights: white for high excitory values, black for non-excitory values (equal to zero), and increasing greyness paralleling decreasing weight values (also decreasing value in an HSV color scale). A red-to-black graduation was used for negative weights: red for strongly inhibitory weights, black for non-inhibitory (equal to zero), and increasing color value (black to red) paralleling decreasing weight values.

1 A 2D, regular layout of stimulus points modeling the way a viewed image is laid on an eye's retina.

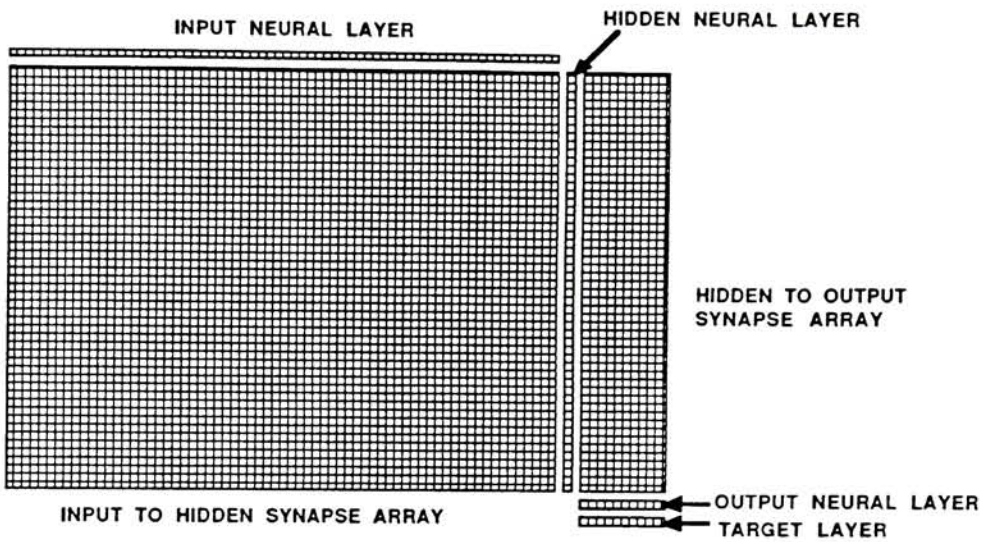


Figure 3-7. Graphic layout used in the experiments (from [Leha88]).

Several training investigations using back-propagation, were run in which the order that the stimuli of hand sketched numbers were presented was varied. The network in the investigation for which most observations was described, was trained to classify presented numbers as even or odd. The evolution of weights and activation values were observed as color changes of the squares in the graphic display. A second view, monitoring ANN performance over time (performance based on target-output error), was also used to follow network convergence.

In one investigation, the author used the resultant pattern to formulate a compression-based modification to the learning algorithm (pre-processing). The network performance degradations that resulted led to the conclusion that weights close to zero are significant. In yet another investigation, weight patterns were used to "discretize" the weight matrices (i.e., weights were forced to 1, 0 or -1). Network performance was determined to be essentially unaffected.

Lehar reports that the weight patterns provided the most interesting information. For example, salient inhibitory connections emerged early and quickly in the

hidden-output matrix at unassigned output nodes, followed by development of "inhibitory-excitory banding patterns" over the output class being learned. As such patterns became stronger, the input-hidden matrix weights for the corresponding hidden node became stronger.

Lehar does not reach any specific conclusions from his investigations about the behavior of the back-propagation algorithm. Instead he concludes that a graphical representation of the network state combined with an interactive interface "facilitated free experimentation with immediate feedback...valuable insights were gained by observing patterns in the synaptic weights." No investigations involving adjustments of learning and momentum constants were reported. One potentially problematic aspect of Lehar's setup is the color mapping used. According to Ware [Ware88], color maps of monotonically varying color (both maps only varied the color value) are highly susceptible to simultaneous contrast effects, which then lead to erroneous interpretation of the colored image. The likelihood of problems is further supported in that the first weight grid alone contained a thousand colored tiles, forming a region large enough to have large color gradients.

3.4 Empirical Work on Visualizing Algorithms

No empirical studies, to this author's knowledge, have been conducted to investigate those design elements that constitute effective animated graphics of the form used in current algorithm animation systems. Various studies have examined discrete related aspects. Reviewed here are studies that examine the general question of the effectiveness of graphics in algorithm instruction, the effectiveness of animation (motion), and the experience level of the algorithm viewer.

3.4.1 Static Graphics

Two reports investigating the effectiveness of using "graphics for algorithm understanding" are based on the use of static graphics [Scan88], [Cros89]. Scanlan's study compared the preferential use of flowcharts (a graphical medium) and pseudocode (a verbal medium) by introductory Computer Science

(CS) students for learning short, complex algorithms, typical of those taught in a data structures course. The global hypothesis of the study was that flowcharts are the preferred tool because both the verbal, logic-dominant "left-brain" and graphical, spatially-dominant "right-brain" are used to comprehend the algorithm presented. Pseudocode in contrast used only the left-brain's logic processing. The study employed a questionnaire to collect data on users' preferences. Scanlan proposed that preference was a sensitive measure of a medium's possible effectiveness of algorithm comprehension. Questionnaire results indicated a significant preference for flowcharts (for short, complex algorithms) by students; however, many students also preferred pseudocode. Scanlan suggests that there are two classes of learners: those with stronger verbal and logical skills and those with stronger spatial skills. The results of this study supposedly concur with other similar studies [Rue85: cited in Scan88]; however, [Gilm84: cited in Cros89] reports mixed results using flowcharts, due to interactions between flowchart size and programmer experience.

An experiment conducted by Crosby and Stenlovsky [Cros89] investigated the interaction between type of algorithm, presentation media, and viewers' programming experience in the comprehension of an algorithm. The media were Pascal code and a graphic representation of the same algorithm. An illustration in the paper of the graphics used for one of the algorithms tested (sorting) was a "sticks" view. Sticks views have been animated with most of the algorithm animation systems described in Section 2.2. In this experiment, however, it and the code were presented statically (from what the present author can tell). View experience was examined because results from other research suggests that meaningful text and patterns (e.g. chess board patterns [Chas74: cited in Cros89]) rather than random patterns are better assimilated by individuals. Consequently, domain experts recognize a larger repertoire of meaningful situations. The dependent measures of the experiment were comprehension and time spent viewing. Generally, the results indicated a significantly longer time spent viewing Pascal code than the graphic counterpart. Comprehension was better when viewing the graphical representation, though the difference was not statistically significant. More important were the findings of significant interactions between algorithm, viewers' experience, and presentation medium, suggesting that conditions such as the amount of detail in an algorithm text and the viewer's expertise influence the media's contribution to understanding.

Another key finding was the effect of presentation order: comprehension scores were significantly higher when graphical representations were presented after exposure to the code of the same algorithm. The reverse did not lead to improvement in code comprehension.

3.4.2 Dynamic Graphics

Two studies examined the effectiveness of dynamic "graphics" for algorithm comprehension [Hays88] and data behavior comprehension [Mezr84] (data visualization as described in Section 2.1). As with the previous two studies, the reference to "graphics" is generalized. Hays used the term "dynamic graphics" by which she means the dynamic display of code and of data structure internals on a computer screen. The objectives of her experiment were to verify if the problem solving performance of CS students would improve using dynamic code/data presentation. Her premise was that using graphics may improve performance by dynamically illustrating algorithm concept/ behavior, by allowing a student to test predicted solutions, and by allowing a student to "discover" solutions by viewing and manipulating (given an interactive capability) the presented data structures. Discovery was defined as students devising their own solutions and diagnosing their own errors. The experiment compared each student's own performance with and without the use of the "graphics." Prior to the experiment, students were thoroughly tutored to use the code/data visualization system. Performance was a combined measure of "correctness points" of solutions to given problems, the time spent on each problem, and the score on a final quiz. The results indicated a significant performance improvement for students who at least viewed dynamic data changes and code execution sequences over a control group who worked all problems on paper. A second experiment found further improvement for the students that were also able to directly modify data structures.

Mezrich, et. al. [Mezr84] investigated the dynamic representation of multivariate data of economic trends[1] as a means of presenting "data evolution properties." Their premise concerned behaviors characteristic of trend type data that exhibit

1 According the definition in section 3.1, economic trend data is a natural phenomenon not based on a general algorithm. Hence this investigation does not examine graphics in the context of algorithm understanding.

"dynamic outliers;" such outliers "differ from other data strictly in [their] dynamics and not necessarily in their values." Animating a trend data stream could provide a "temporal neighborhood" such that the behavior of the data can be learned. Their study was not designed for generalization, rather to examine one specific task. This was the performance of perceiving the set of four variables with a positive correlation from two sets, presented in three different static and in one dynamic form. The static forms were: four tiled, four stacked, and four overlaid plots each on a page. The dynamic form was a modified meter-type display: five vertical "needles" were arranged cascaded from the left screen edge and screen bottom and then mirrored along the center vertical screen axis. The outer most bars mapped the most current data values, and each successive bar mapped the data "grandfathered" by the previous bar, thereby forming a temporal neighborhood. The experimental procedure constrained the viewer to a fixed five second view of the data in each form. Viewers were "sophisticated in the use of conventional computer graphics"; each was trained in the dynamic view only. The results, averaged for the six subjects, suggested that viewer performance was superior with the dynamic presentation form; overlaid plots were second best. Performance with stacked and tiled formats was significantly worse. Mezrich conjectures from the results that the stacked and tiled forms involved a "local" perceptual processing, entailing a detailed feature-by-feature examination of the data. The dynamic and overlaid forms involved a "global" perceptual processing (as defined in [Jule71: cited in Mezr84]) in which the viewer "grasps" a pattern that is not evident by detailed scrutiny.

3.4.3 Discussion

These studies (and certain references therein) generally suggest that some type of complementary representation to code or pseudocode can heighten a learner's comprehension. In these cases the complementary representation is broadly in the form of annotated graphics and simulated (single-step) code execution. This latter form was available in the BALSA system and reported to have been very useful [Brow85]. Similarly, the observations of Crosby et al. (1989), regarding presentation order effects (i.e., code before graphics), appear to correlate with Brown's findings in BALSA, that students comprehended an algorithm better when presented with both the code and animated graphics. The work by

Mezrich, et al. (1984) is essentially dynamic DV. The experimental procedure he used could have confounded the results, particularly in that subjects were "trained" to read the dynamic display but none of the others. Similarly, Hays' (1988) findings may be debatable, since tutoring the subjects prior the study exposed them to a novel tool that might have then biased many of them. Finally, Scanlan's (1988) use of preference to infer effectiveness seems simplistic; key factors such as previous exposure, experience, and course requirements of each student involved were not factored in.

In general, these empirical studies investigate only portions of some of the issues raised in this thesis; clearly the factors that need to be controlled for such investigations can be overwhelming.

4

Project Description

4.1 Motives, Objectives, and Challenges

This project presents a relatively novel application of algorithm animation (AA) for the exploration of problems involving three-layered ANN topologies and the back-propagation learning algorithm. The primary objective was to investigate issues surrounding the use of algorithm animation for learning about ANN learning algorithms. There were three interrelated subordinate objectives:

- determine preliminary guidelines for the selection of graphic attributes and layouts suitable for animating data and operations of a neural learning algorithm
- determine which algorithm data structures best portray the algorithm's behavior, and in what ways
- determine an algorithm animation system architecture that provides interactivity and extensibility needed to address the previous two objectives

Basically, the project sought to answer these questions:

- What combination of algorithm data structures characterize correct and faulty learning behavior?
- How should selected algorithm data structures – several of them multidimensional – be coded on a two-dimensional screen?
- In what ways can lines, circles, rectangles, color, and related attribute changes be used to portray stalled learning?
- Finally, what should constitute a system for animating ANN algorithms, that can be easily extended to handle new learning algorithms, network architectures, new algorithm data structures, and new views?

Similar questions and issues have been summarized by Myers [Myer86] in his review of graphical programming systems.

The project was driven by four key motives:

- no work specifically using algorithm animation techniques for the exploration of learning algorithms has been published
- there are no known guidelines for the design of effective animated graphics
- the reported benefits of algorithm animation have been informally arrived at and are based on algorithms for manipulating data structures (e.g., sorting, searching, hashing, etc.)
- ANN learning processes are commonly described using strong imagery references

This project was partly inspired by BALSA and similar algorithm animation systems. Understanding how BALSA's many graphic views were determined, sparked initial interest. The question of why static visualizations of learning algorithms have not led to more prevalent use of animated visualizations, was also a motive.

Part of the project's scope is bounded by three assumptions. First, static visualizations of ANN data structures - usually portraying learning in progress - can provide useful information for the learning-algorithm researcher. Support for this assertion is the regular accompaniment of various graphics to describe certain interesting behavior of an algorithm (Hinton maps, Vornoi tessellations plots, etc.). Second, ANNs, in the process of learning, are difficult to follow and analyze, either for instructional or diagnostic purposes, when using direct means such as reviewing printouts of algorithm data structure values. This problem has been reported in [Leha88], [Tour89], and in [Koho88]; in all cases the difficulties were "overcome" by using static or dynamic visualizations. Third, viewers of the animations created for this project would not be expected to interpret them without a prerequisite understanding of back-propagation fundamentals first, and then an understanding of the graphic elements of each view. The experiences with BALSA reported by [Brow86], and the experimental results of [Cros89] suggest using animated graphics for teaching only after understanding a written description of the problem.

Static visualizations of ANNs appear frequently in the published literature. Are animated visualizations any better? This author proposes three reasons for animation. First, the data structures involved in learning algorithms are mostly evolutionary; a direct way to capture this is dynamically. Second, fundamental principles of graphic data analysis recommend using *relevant* data display

techniques to expose *trends* in the information of interest [Cham83]. Animation is an example of the Gestalt principle of continuity contributing to the perception of "emergent features," i.e., visual features not evident viewing the static "frames" of the animation. Finally, by mapping the time component of the algorithm (e.g., forward/backward propagation cycles) to a view's temporal dimension (i.e., frames), a spatial or symbolic channel can be "freed," either to reduce visual clutter, or to provide a channel for another key data structure. One final important aspect of dynamic visualizations is the ability of the viewer to react and control the learning process at any stage, provided that the environment supports real-time user interaction.

4.2 Project Overview

4.2.1 Approach

Design principles from three diverse domains were merged to structure an approach to the problem:

- graphic design techniques for data analysis
- perception theories: Gestalt, emergent feature, and texture mapping
- algorithm animation systems design

Theories, methods, and techniques from each of these domains provided practical ideas towards solving the project's challenges. Graphical data analysis provided the most pertinent foundation for the project's requirements of graphically mapping program variables. Particularly applicable are techniques for the two-dimensional display of multidimensional data, which are inherent in ANN algorithms. Of the techniques researched, generalized draftsman displays, and multi-code symbol plots were explored for the project's view designs. Principles and guidelines for producing effective displays with high visual impact were also adopted from [Cham83] and [Bert81] (see Section 3.1). Issues such as color, contrast, and gridding were also addressed [Tuft90], [Bert81], [Marc82], [Ware88].

Visual perception theory provided suggestions towards "effective displays." A challenge here was to abstract applicable design guidelines from several theories.

Properties such as closure, symmetry, regularity, and complexity were adopted since both theoretical and empirical studies support their contribution to good form. The display of gridded multi-code symbols in a way that produced global features in a texture map was also explored. A texture map can develop global features through the formation of contours and gradients produced by the underlying elements/symbols. At any instant, the viewer can also examine details of symbols of specific interest. As an example, one application of global feature development was to explore whether the feed forward/thresholding behavior of units could be displayed as a global feature.

The domain of algorithm animation provided much of the framework for implementing a prototype system, NetViz. The use of unobtrusive algorithm code annotation, the separation of algorithm code and data structures from the animation code and data structures, and the provision of run-time user control over the algorithm and views were predominant aims. The cubic taxonomy of algorithm animation systems (Section 2.1.2), originally used to classify AA displays in the published literature, was adapted to profile the views in this project. The profiles thus established a framework for formally evaluating the view designs, and further defined the scope of the project.

In conjunction with the principles from these domains, initial view designs goals included:

- clearly depicting differences between correct (convergent) and incorrect (non-convergent) learning behavior
- portraying oscillations of the error point in weight-space
- portraying the formation of feature maps in the hidden layer
- depicting the presence of features such as "basins of attraction" in weight-space

To explore the view designs, three ANN classifiers were constructed and simulated:

1. an XOR operator
2. a symmetry classifier of one-dimensional, even-length binary-encoded strings
3. a classifier differentiating horizontal and vertical lines

These classifiers were chosen with the following criteria:

- The networks should range in architectural complexity to explore the impact of scaling network architectures to the views.
- Networks of both relatively low dimensionality (mostly defined by the weights) and very high dimensionality should be used to provide a means to explore the *transition* component of the AA taxonomy.
- Different network architectures can provide a basis for "viewing the problem from different angles."
- Classifiers that have been shown to converge should be used. This is to avoid debugging an ANN configuration while trying to study view effectiveness. (Unsuccessful training can also be studied with these classifiers)

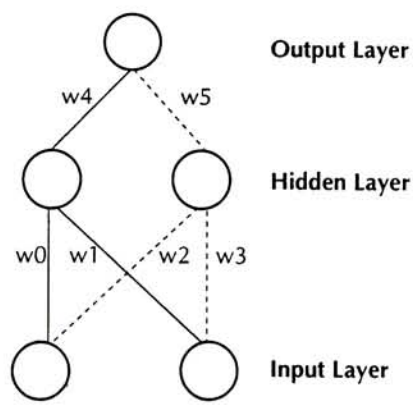
4.2.2 Neural Network Classifiers Explored

The XOR Operation

The XOR is a boolean operation defined in Figure 4-1(a). Although apparently simple, the operation is a challenging problem for ANN learning, because two decision boundaries are involved, that, by design, can only be formed by a multi-layered network architecture. The network topology used is depicted in Figure 4-1(b). This is the simplest network topology for this problem (apart from connecting the input layer directly to the output node which is an atypical form). The complete set of training patterns presented at the input layer are those in the left column in Figure 4-1(a); corresponding target patterns are in the right column.

Input		Output
1	1	0
1	0	1
0	1	1
0	0	0

Figure 4-1(a). XOR operation.



(b). Network topology.

Symmetry Classifier

The network for the symmetry classifier is comprised of a six-unit linear input layer, a two-unit hidden layer, and a single output unit, as shown in Figure 4-2. All layers are fully interconnected: input to hidden, and hidden to output. Input patterns are binary strings; strings are symmetric if the left three digits are mirrored in the right three digits, otherwise the string is asymmetric. For example, "100001" is symmetric, whereas "110001" is asymmetric. Symmetry is classified as a '1' at the output unit, and asymmetry as a '0'. This network's total weight dimension is 14[1].

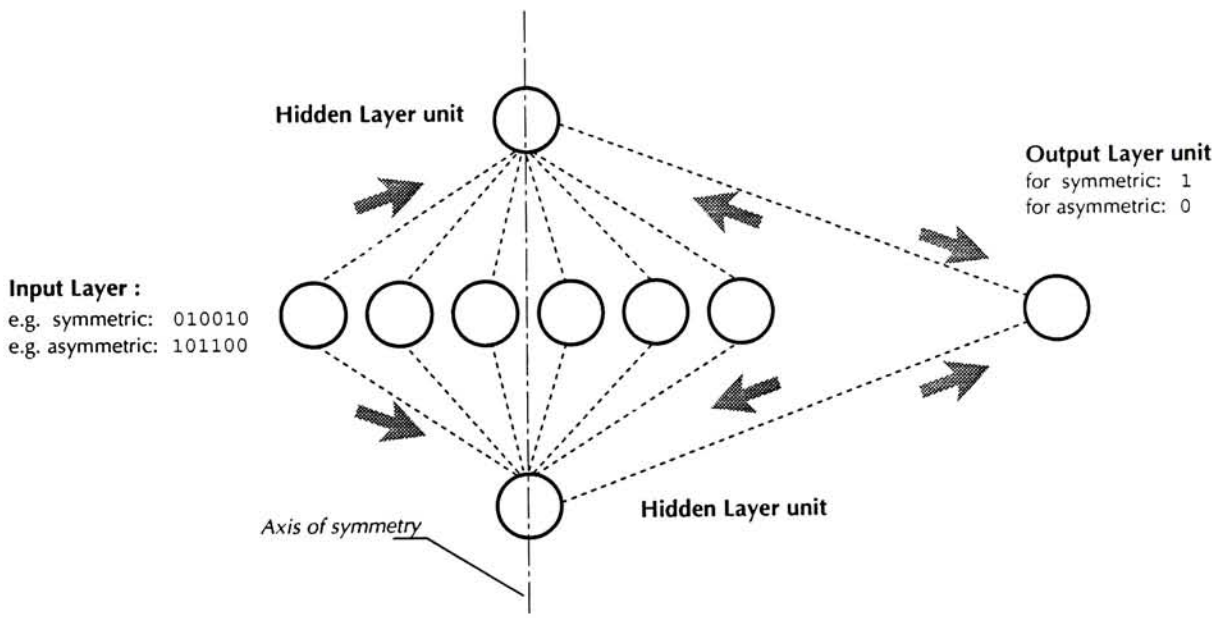


Figure 4-2. Symmetry Classifier.

Orientation Classifier

One of two network topologies for the orientation classifier includes a retinotopic, 4 by 4 input layer, fully connected to a 2 by 2 hidden layer. The hidden layer is fully connected to a two-unit output layer (Figure 4-3(a)). The

1 Derivation: number of input units (6) x number of hidden units (2) + number of hidden units (2) x number of output units (1) = 14.

second topology differs only in the hidden layer (and number of connecting weights) size: 3 by 3 units (Figure 4-3(b)). "Horizontal" input patterns (for training) have any four horizontally adjacent bits of the 16 bit field,"on"; "vertical input patterns have the composite, vertically adjacent bits (Figure 4-3(c)). For a 16-unit input layer, there are at most four perfect horizontal patterns and four vertical patterns. Horizontal patterns are classified by "01" and vertical patterns by "10." The weight dimension of the smaller topology is 72, and the larger is 162[1].

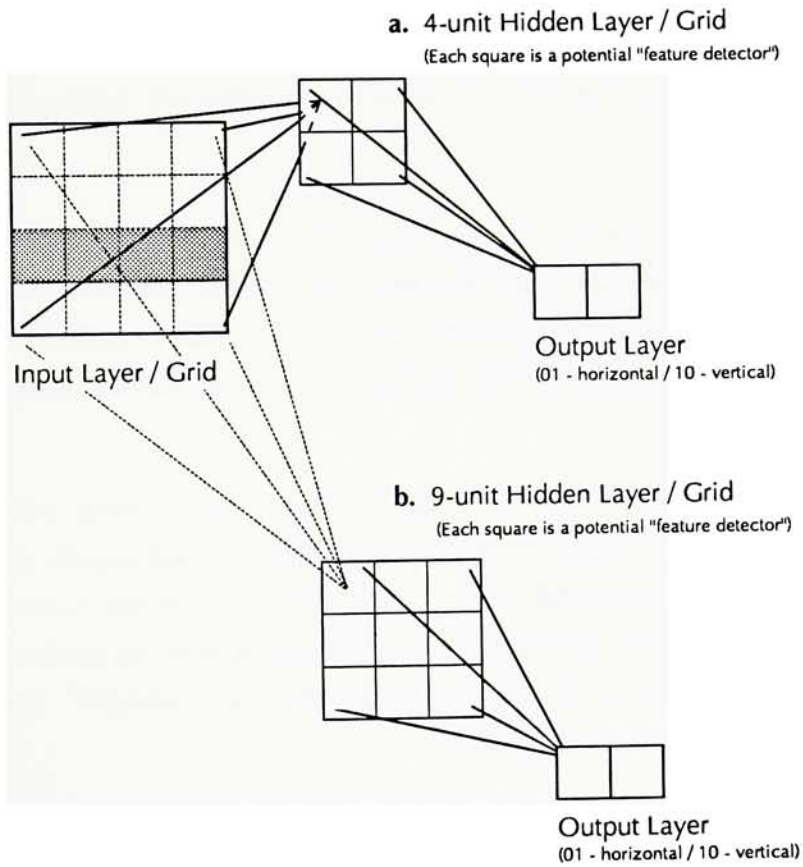


Figure 4-3(a). Orientation classifier topology: 4 hidden units.
(b). Same classifier with 9 hidden units.

1 Small topology derivation: 16 x 4 + 4 x 2 = 72. Large topology derivation: 16 x 9 + 9 x 2 = 162.

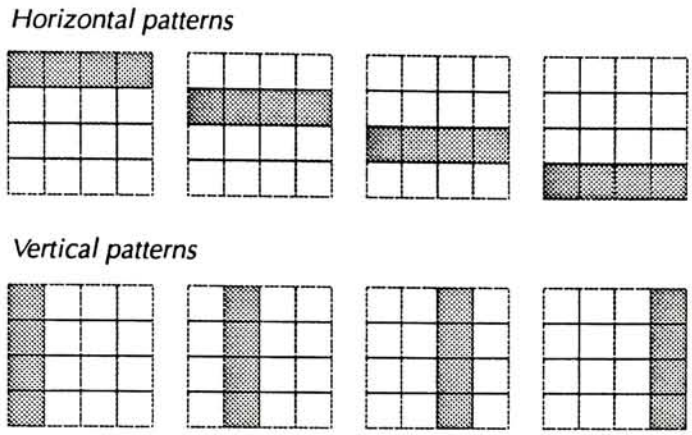


Figure 4-3(c). The training set of horizontal and vertical input patterns.

Although a single output unit would suffice for this classifier (i.e., '0' for horizontal and '1' for vertical), an objective with this ANN was to compare the types of feature maps produced by output units explicitly mapping classes, with feature maps produced by a "binary" class detector (in the symmetry classifier).

In summary, the "glue" of the project's objectives is the design of multiple views, each depicting unique and complementary data combinations or views isomorphic to each other. As Brown describes in [Brow85], multiple views of the same data structure provide a "thread" connecting different graphic representations. Together they "provide a Gestalt" of the algorithm's behavior over time.

4.3 Project Design

4.3.1 Animation View Design

Designing appropriate animation views was one of the central activities at the start of the project. The objective, "to create effective views," required the synergistic application of graphical data analysis principles, principles of good form, and concepts developed in existing algorithm animation systems. Principles of human factors were involved in the display and user interface

design. One of the requirements for an effective view was the ability to expose patterns, through animated algorithm data, that would not otherwise be perceived by: viewing the back-propagation algorithm code alone; or by examining the (final) data of the trained network by a means such as a Hinton map. A pattern in this case is determined through a "global" perceptual process. This design objective was coupled with the need to display the multidimensional data structures inherent in the algorithm. Although principles from statistical data display were applied, the focus was on qualitative (trend) information in the algorithm data, rather than quantitative information.

In this thesis an *animation view* or "view" refers to both the static and dynamic layout of graphic objects and the associated graphic attributes that appear during animation. A *display* is the contents of a window from the perspective of the workstation windowing system (UIS). A *viewport* is used synonymously with *window* to refer to a framed region of the screen containing a view[1]. A view that is animated by rendering a new view over top the previous view is called a *movie*. Finally, a reference to a specific frame of an animation is a *snap-shot*.

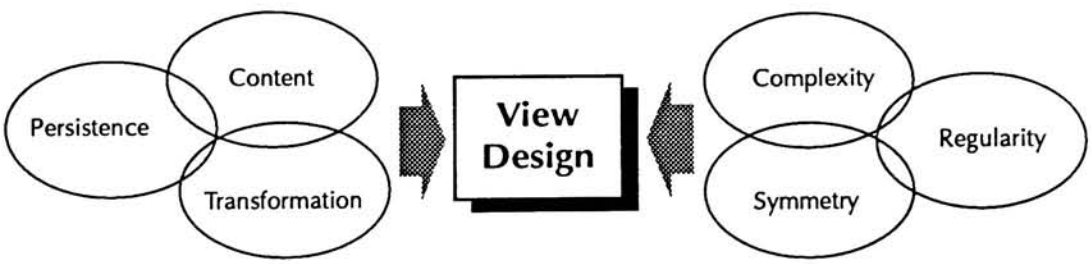


Figure 4-4. Elements contributing to view design.

Figure 4-4 depicts the key elements which contributed towards the view design objectives. One purpose of integrating these diverse concepts was to define the scope of the project. Furthermore, it helped to establish a framework within

1 At the detailed level of the workstation windowing system, a viewport is the framed region seen on the screen and a window is a conceptual frame whose dimensions determine what appears in a viewport.

which informal judgements could be made on each view's effectiveness, based on specific design elements, rather than a "style."

The views were designed specifically for the back-propagation algorithm (both feed-forward and backward phases). Therefore, the set of potential variables and data structures (from here on simply "variables") to map to graphics was constrained. The first step in the view design was to: (a) identify those variables that would best portray algorithm behavior, and (b) determine the appropriate associations between selected variables. Table 4(A) summarizes the final set.

TABLE 4(A) Algorithm Variables to Monitor

Variable/ Data structure	*Co-variables*	*Behavior portrayed*
• weight	bias, activation, RMS, current epoch	a. "feature maps/detectors" describing each hidden node b. surface contours and path of error-point through weight hyperspace
• delta weight	RMS, TSS, current epoch	a. error point oscillations b. a "current" view of weight change c. (unknown)
• weight error derivative.	current epoch	(unknown)
• bias	weight	overriding activation behavior of hidden and output units
• gradient correlation coefficient (GCOR)	current epoch, weight	change of direction of the "error point"
• total sum of squares error (TSS)	current epoch, weight	a. depicts system energy level; "height" from the global min. b. trend over time exposes convergence; oscillations
• root mean squares (RMS) of error	current epoch, weight	a. depicts system energy level; "height" from the global min. b. trend over time exposes convergence; oscillations
• error signal	weight, current epoch	(unknown)
• activation	weight, bias	which feature maps participate in classification (during a feed-forward phase)

Note that for a few of the variables in the table, their portrayal of algorithm behavior was not predictable; instead, their behavior would be determined through explorations using algorithm animations.

TABLE 4(B)	Operational Definitions	
Attribute	*Definition*	*Scale*
Image:		
Symmetry	• The number of horizontal, vertical and radial axes of symmetry of a view as a whole.	• Low, MedLow, MedHigh, High (Low: no global axes of symmetry; High: 4 or more axes of sym.)
Regularity	• The number of horizontal, vertical and radial axes of symmetry affecting a unit of a view (i.e., a view component that is repeated in the view).	• Low, MedLow, MedHigh, High (Low: no local axes of sym.; High: 4 or more axes of sym.)
Complexity	• The number of different components, angles, nested enclosed areas, and direction changes.	• Low, Medium, High (Low: 10 or fewer defined components; High: 30 or more defined components.)
Motion:		
Complexity	• The proportion of a view that is updated (changes), and the visible number of elements changed per update.	• Low, Medium, High (Low: localized updating; little visible activity; High: global updating in view; many visible changes.)
Grid Size	• The number of view units and functional units.	• Small, Medium, Large
Persistence (history — current)	• The duration of which past data remains in the display.	• History, Bound Record, Current (History views: display view transformations from the start of animation; Current views: display only the most recent information.)
Transform (discrete — continuous)	• The detail of change in the animation (of data).	• Discrete, Interpolated, Continuous (Discrete trans: one large update of symbols; Continuous: small, incremental updates, possibly through interpolation)
Content (direct — synthetic)	• The style of algorithm data representation.	• Direct, Combined, Synthetic (Direct: data mapped isomorphically to graphics; Synthetic: graphics representing data not of algorithm, but caused by algorithm execution).

The second step (actually performed in parallel with the first) involved establishing view design profile attributes that would guide the design of

different views, and then serve to somewhat formally categorize the final views. The final profile attributes and definitions were abstracted from the three main domains researched, i.e., algorithm animation, good form, and principles of graphical data display. Table 4(B) summarizes operational definitions applied in view design.

Simple scales were used for each of the parameters. For the last three view attributes, the scale reflected the transitions between anchors shown in parenthesis. The remaining attributes were scaled from "low/small" to "high/large."

The actual view conceptualization process involved describing the ANN system's behavior verbally and concurrently sketching layouts and symbols to map variables. From the verbal descriptions, analogies and connotations ultimately contributed to visual ideas; examples of this are the "traversal of the system error-point in weight-space," and the "evolution of feature detectors in the weight matrices." The view design concepts were then audited against the algorithm variable constraints and profile parameters. Exploiting spatial mapping was of highest priority; symbol size, and angle were second. Color mapping was of lower priority. For example, to fulfill elements of good form, the view was "graded" for symmetry; to fulfill visual impact of data display, graphical attributes that might over depict a potential relationship (destabilize the variability) were separated. This process also resulted in some adjustments of *a priori* constraints. The final view designs reserved certain aspects violating good form such that a comparative analysis could be performed. Such aspects were made user selectable, so views could span a range of "goodness" of form.

The ultimate goal was to fully construct three views, each with unique profiles. In addition, each view would depict a subset of the selected variables, so that animations, isomorphic with respect to the data displayed, could be comparatively explored. The results of this process were (many rejected views and) the following final selections:

- the Draftsman Plot (DP)
- the Texture Map (TM)
- the Receptive Field Map (RF)

One other view not designed as part of this group, but implemented as a complementary view is the Energy Plot (ERG) view. It was implemented after initial trials with the DP and TM views as a view that users of NetViz would find more familiar and easier to interpret. It also seemed to portray convergence and non-convergence better. Each of these views is described in more detail below.

4.3.2 Final Views: Details

Draftsman Plot

The Draftsman Plot (DP) view is a conceptual and functional analog of the draftsman scatter plots described in [Cham83]. The main function of this view is the clearer, two-dimensional depiction of an ANN's weight-space (also: hyperspace), a potentially large multidimensional data space. As illustrated in Figure 4-5, the predominant objects are square tiles arranged in "blocks." Each tile is a distinct plotting field of two weight values over time. The particular weight pair is determined by row and column; tiles in the same column plot the same weight in the abscissa, and tiles in the same row plot the same weight in the ordinate.

Each block defines the weight-weight pairing by layer. The three blocks in Figure 4-5(a) represent a network with a single hidden layer. Tiles in the first block map paired weights of the input-hidden weight matrix. Tiles in the second block map weights paired between input-hidden and hidden-output matrices. Finally, tiles in the third block map paired weights of the hidden-output matrix. Figure 4-5(c) further illustrates the convention for labeling weights and the weight associations between conventional topological network views and the DP view. Basically, consecutively counted weights are those leading *into* a network unit.

The DP view deviates from a classic draftsman plot in that the main diagonal of tiles (mapping the same weight on both abscissa and ordinate) and the mirror image of tiles above the diagonal are not drawn for the first and third blocks (actually modulo 2 when there are more blocks).

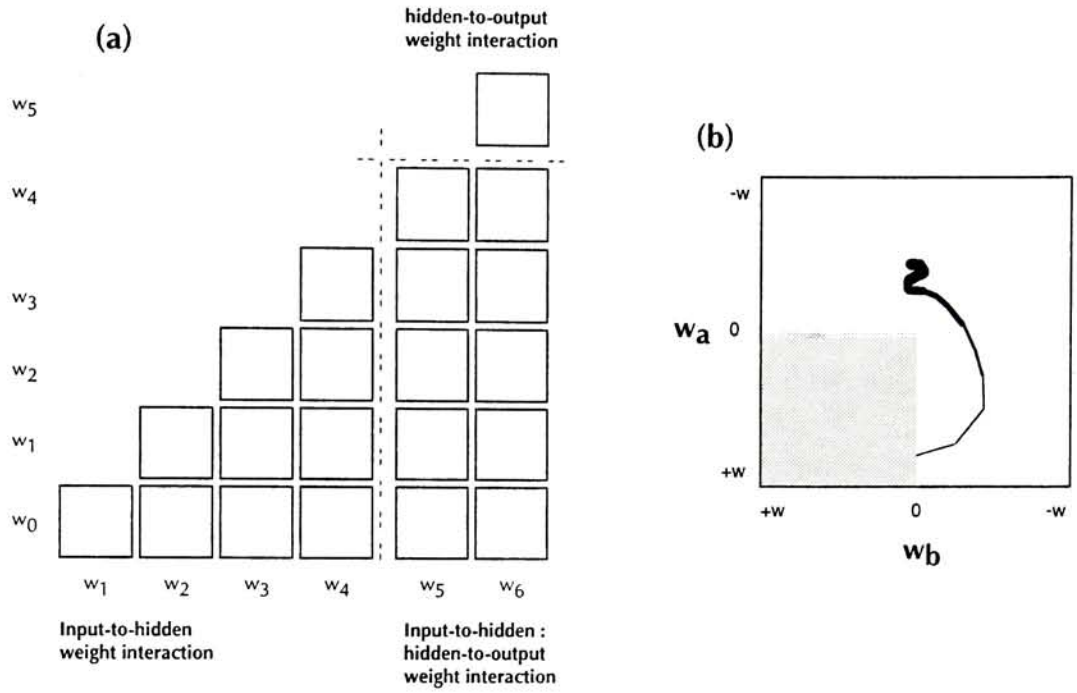


Figure 4-5(a). Draftsman plot.

(b). Detail of a tile.

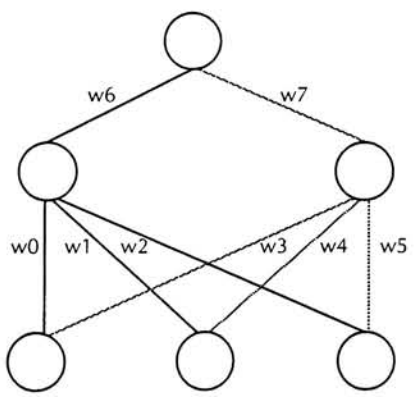


Figure 4-5(c). Weight labeling convention in a topological view.

Figure 4-5(b) is a detailed view of a single tile. The actual scale range of both axes is user defined. If the scale set includes (0,0), then the negative region is rendered in a light shade of magenta for reference. Within a tile, one of three

primary variables mapped are plotted in the spatial dimensions per rendering cycle: weights, delta weights, and weight error derivatives (WED). Each tile then is a trace of the conceptual error-point along a two-dimensional error surface in weight-space; collectively, the tiles depict the traversal through weight "hyperspace."

The Draftsman Plot can optionally map two of three secondary variables: total sum-of-squares (TSS) error, root mean sum-of-squares (RMS) error, and gradient correlation coefficient (GCOR), using two of three coding schemes: pulse width, line width, and color. For example, RMS error can be represented by a varying pulse width while the color of the pulse line (and weight-weight segment) varies according to the GCOR value. (The implementation of this view enables the user to also map one secondary variable to two different coding schemes.)

The pulse width coding scheme involves a separate line segment drawn perpendicular and bisecting a weight-weight segment (i.e., the line drawn between $(w_x,w_y)(t_n)$ and $(w_x,w_y)(t_{n+1})$). The pulse width correlates with the value of the mapped variable; pulses are widest with large mapped values, and narrowest (the smallest is a "dot" (pixel)) with small mapped values (depending on the exact scaling factors). Figure 4-6(a) illustrates the details of the scheme. Note that the pulse width (using an appropriate scale) is wider, more visible in a view than a weight-weight segment may be, and therefore affords a more salient image of the error point's *direction*.

The line width scheme maps the selected secondary variable to a set of graduated line widths and line styles. These line attributes are applied to a weight-weight segment. A scale can be set up such that high values map to wider lines and smaller values to narrower lines. Smaller values, for example (but not restricted to) negative values, can be mapped to single pixel width line styles, that range from "long-dashes"[1] (the gaps in an otherwise solid line are few and small), to "long-short dashes" (gaps are slightly larger and more in number), to, finally, "dots" (solid line segments are very short).

1 The names of the line styles used here are approximate. Users can define as many as needed of their own line styles in one of the system files.

The color scheme is similar to the line width scheme in that the weight-weight segment is affected. The particular color mapped to the secondary variable's value depends on the color scale specified by the user.

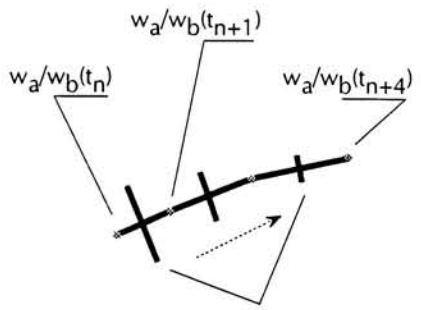
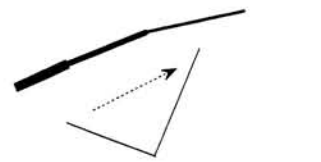


Figure 4-6(a). Pulse width encoding.



(b). Line width/style encoding.

Texture Map View

The Texture Map (TM) view involves the following hierarchy of elements:

1. a symbol cell
2. an epoch column
3. a TM plot or movie

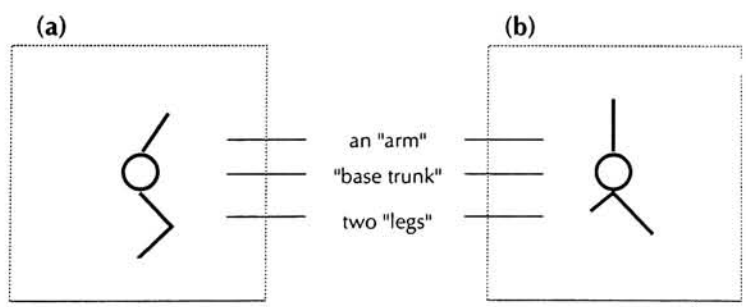


Figure 4-7(a). A hidden symbol cell, growth configuration.

(b). Same symbol cell, spikes configuration.

determine the size of the symbol. The key part is the symbol itself, which can map a set of algorithm variables. Globally, each symbol correlates with a network processing unit. Specific symbol components map separate variables; there are three key ones:

- a set of arms
- a set of legs
- a base trunk

Both arms and legs can map to one of three possible variables: weights, delta weights, and weight error derivatives (WED). There are two possible configurations, both user-definable: *growth* (Figure 4-8(a)) and *spikes* (Figure 4-8(b)). Each arm segment of a growth configuration correlates to the connection weight (from here on "weight" also implies the other valid weight-related variables) leaving the unit. Figure 4-8(a) clarifies this mapping; each arm segment, starting with the closest to the base trunk, corresponds to the weight starting with the left-most (for the particular processing unit) in the network's topological view. Variation in weight is mapped to changes in arm and leg segment angles; angle was considered as providing the most compelling differentiation. Zero weight values are depicted by a vertical arm segment. Positive weights pivot the arm left of vertical; negative values, to the right[1]. An analogous mapping exists for leg segments. Leg segments correspond to connection weights entering the unit. The leg segment nearest the base trunk corresponds to the topologically left-most weight entering the unit; the lowest leg segment corresponds to the topologically right-most weight. Zero weights produce a vertical leg segment, positive weights pivot it left of vertical, and negative weights, to the right of vertical. The unit length of both arm and leg segments is determined by the size of the symbol cell and the total number of input and output units in the network.

The spike symbol configuration maps the same variables as the growth. Basic differences between configurations are the positions of segments and the depiction of values. Figure 4-8(b) describes the mapping between spike symbol and network topology. As with arms and legs, there are two classes: "top spikes"

---

1 The region left of the vertical central axis of the symbol was designated "positive" early on from the perspective of arm rotation. Positive rotation for arms is counter-clockwise, following a polar coordinate system.

and "bottom spikes." Spikes in each class are arranged uniformly within a horizontal 180 degree arc. Spikes depict current weight values in proportion to their length: smaller, more negative weight values produce a correspondingly shorter spike segment. Larger, positive weight values produce correspondingly longer segments. Zero weights produce a spike length in-between.

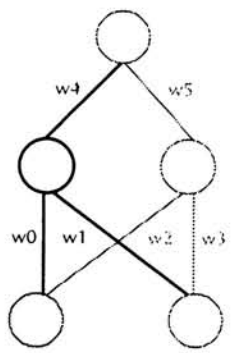
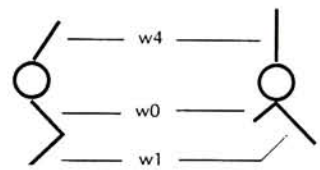


Figure 4-8(a). Details of a growth symbol configuration.



(b). Details of a spike symbol configuration.

Since a symbol cell corresponds to a network processing unit, there are three types of symbol cells: input layer, hidden layer, and output layer cells. The difference between them is the potential of having both a set of arms and a set of legs. Only hidden layer symbols can have both; input layer symbols have only arms, and output layer symbols have only legs.

The base trunk defines the center of the symbol and its cell. As a circle of a varying diameter, it can map to one of two variables: activation and incoming bias of the corresponding processing unit. Direct values are mapped to the circle diameter; small, negative values produce a small circle (3 pixel square minimum), and larger positive values produce large circles.

The user/viewer setting up a TM view can select to not display legs, arms or the base trunk. Such a view emphasizing variables differently reduces the view's complexity and regularity. From one perspective, it also stabilizes variability (e.g., weights are only shown once, as arms or legs).

An *epoch column* is formed by stacking appropriate symbol cells. Each symbol cell in an epoch column correlates to a processing unit in the corresponding network. Figure 4-9 illustrates the convention used to arrange appropriate symbol cells within a column. By default, symbol cells are adjoined top-to-bottom; however, a parameter, "intercellgap" can be user specified to either create a gap between cells or force cells to overlap (using a negative value).

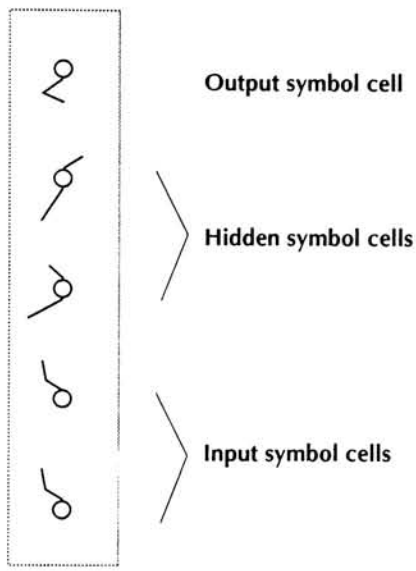


Figure 4-9. Symbol cell arrangement in an epoch column.

The epoch column is the key integrating element of both a texture map plot and movie. As its name suggests, an epoch column describes the set of weight, bias or activation values at a specific epoch. In forming a texture map plot, epoch columns are rendered left to right as a time-series. Over many epochs the evolution of weights and other variables is recorded. By default, columns are placed adjoining one another; however, using the parameter "intercolgap," a viewer/user can separate columns or make them overlap.

Two implementation related provisions are the rendering by pattern and the stacking of texture map plots. An epoch column can be rendered at the finer unit of a pattern update. At this setting, each column depicts the input patterns (as input layer "activation" values), and activation or bias values for the particular input pattern presented. Some weight adjustment may also be perceived, unless

the learning algorithm is set to accumulate weight changes for the entire pattern set and adjust weights at the end of the epoch (i.e., *follow* mode; see Appendix B). In that case, changes in growth angle or spike length appear in the column of the last epoch pattern. Given the finite window width, a texture map can ultimately be plotted beyond the visible bounds of the window. There are two ways to compensate for this: (i) to pan the window's field of view and realign the current column to the left window edge; (ii) to "reset" the view so that a new plotting row is started at the left window edge and above the previous plot. The latter method has the advantage of keeping more (if not all) of the plot within view.

A TM *movie* is a variation whereby each epoch column is rendered "in place," erasing the previously rendered epoch column. When the view is updated, typically once per second, the arms, legs, or spikes appear to swing and undulate in accordance with the rate and direction of changes in mapped variables. The contents of a movie frame is otherwise identical to what is plotted in time-series fashion in a TM plot. Whereas the plot view depicts a "historical" persistence view (see Section 4.3.1), the movie view is at the opposite end of the range, as a "current" persistence view. An advantage of a TM movie is the ability to fill a window with one epoch column, thereby increasing the resolution of segment angle changes.

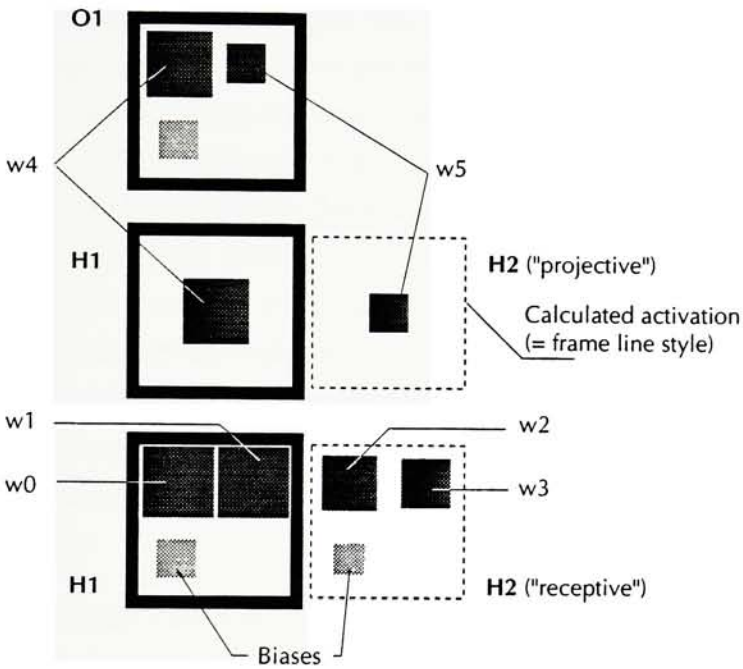
Receptive Field View

The Receptive Field (RF) view is a movie-type view designed for two main functions: (i) to clearly depict hidden and output layer feature detectors, and (ii) to depict the flow of the learning algorithm. The hierarchy of view elements are:

1. data variable symbols
2. "receptive" unit tiles and "projective" unit tiles
3. data variable frames
4. a color tile
5. an RF movie

Data variable symbols are simply squares and circles mapped to weights and biases; the user/viewer can decide, for example, to map weights to squares and biases to circles. Weights are colored dark grey and biases blue, so that they are

distinguishable when both are encoded by the same symbol. Variation in weights and biases are mapped to a continuous change in symbol size (area). Given a scale that spans a positive to negative range, symbol size would correlate from a large to a small symbol; no special coding is used to indicate a zero value.



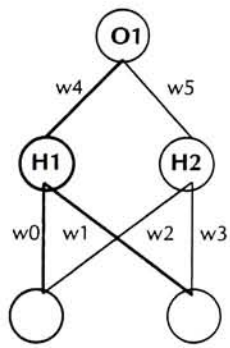


Figure 4-10(a). RF view of the XOR classifier. (b). Mapping convention.

Data variable objects always appear within the boundaries of unit tiles. There are three types of unit tiles: *hidden receptive-field, hidden projective-field,* and *output receptive-field unit tiles.* As the names suggest, hidden unit tiles represent the hidden units of a network and the output unit tiles represent the output units. A receptive-field refers to the view perspective of weights "entering" a specific hidden unit from the input layer. Correspondingly, a projective-field is the view of weights "leaving" a hidden unit (connecting to output units). Although these two perspectives of weights are named here, the TM plot view's arms and legs are isomorphic analogs. The size of tiles and the spacing between them is user-defined, through the view description file (see Section 5.2.4 and the Appendix B). The layout of data variable symbols is as well. Weight squares may therefore

either be arranged in a single row, in a block or diagonal, which ever produces the most recognizable feature detector pattern.

Figure 4-10(a) is an example RF view. Figure 4-10(b) illustrates the mapping convention between a topological network view and the RF view.

Both hidden and output tiles have an additional encoding whereby activation values are mapped to tile frame thickness. *Activation frames* are rendered in a light blue. A normal scale maps low activation values (near zero) to the thinnest line width and larger activation values to thicker lines. The user can alternatively include line styles in the scale, as described in the section on DP views. Using line styles, the maximum of the activation's sigmoid curve (0.5) could be mapped to the thinnest (1 pixel) solid line; values below this inflection point would be depicted as dashed lines. Output tile frames have one more encoding for output unit signal error. This frame is rendered in red, after the tile's activation frame has been rendered. The mapping scale is analogous to that of activation values: thick to thin (including line styles) corresponding to large to small signal error levels.

The color tile is an optional "symbol" through which RMS error changes can be monitored as color changes. The system global color map is used. An option in defining this mapping is to either render an RMS error update as a discrete change or as a "sliding" change (interpolated).

Variable mapping in an RF view is flexible. A view may map only weights, only biases or only activation values; or a view with any two, three or all supported variables can be specified by the user. Two other view specification parameters affect data variable symbols: whether to render symbols as filled or hollow, and whether to render their updated size in a discrete (transform) or interpolated step.

An RF movie can run in one of two modes, determined by the value of the *stage delay* variable. A stage delay of one or more causes the RF movie to effectively "single-step" through the major phases of the back-propagation algorithm. Each stage is depicted by rendering those algorithm variables calculated at that stage. For example, during the feed-forward stage, hidden unit tile activation frames (if specified by the user) are rendered, displayed for one second (corresponding to a stage delay of one) and then erased. Next, the output unit tile activation frames

are rendered in the same way. The back-propagation phase begins with the rendering of output unit tile error frames, followed by the updates in weight-square/circle sizes. Before the learning cycle begins again, all tile frames are reset (erased).

A stage delay of zero provides a constant view of all mapped variables (except output error frame). Frames, weight, and bias sizes are updated every rendering cycle, so that frames that grow thicker and thinner, and squares that shrink and expand are perceived over time.

Energy Plot

The Energy plot (ERG) view is a simple "time" plot of one of two variables: TSS and RMS error. The term "energy" is used generally, as a synonym to the two forms of system error plotted; the goal of an ANN system is to decrease its energy level to a lower, equilibrium, by decreasing the total error.

The ERG view was created as an ancillary view to monitor convergence. Nonetheless, its setup is consistent with the other three views. It can be of any size (on the screen) and its *render cycle* can be varied.

RMS was expected to be the most commonly plotted ordinate variable. The only recognized abscissa variable is *epoch number*. When the learning algorithm is set to update weights at the end of an epoch (*epoch update mode*), a TSS plot provides essentially the same curve as an RMS plot, except based on a different ordinate, since TSS is a larger value (accumulated total; RMS is an average). A more appreciable use of TSS is in *pattern update mode* . Then at each render cycle, the sum of squares error for *each pattern* is plotted. Consequently, over time, the error decrease of the highest and the lowest error-committing units are reflected in the plot.

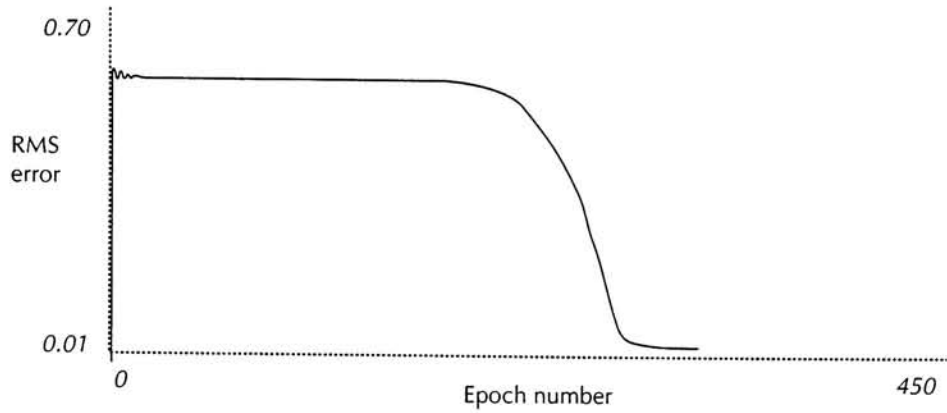


Figure 4-11. An Energy plot example.

Discussion

Symbols and mapping schemes of each of the views are summarized in Table 4(C). Profiles of each view are summarized in Table 4(D).

TABLE 4(C)	**Summary of View Mapping Options**	
View Type	*Algorithm Variable*	*Variable Mapping Options*
DP	weight/dweight/WED	coordinates (distance)
	RMS / GCOR	pulse (length), color (color)
TM	weight/dweight/WED	growth (angles), spikes (length)
	activation / bias	base (size)
RF	weight/dweight	grey squares (size), grey circles (size)
	bias	blue squares (size), blue circles (size)
	activation	blue frame (thickness/line style)
	output error	red frame (thickness/line style)
	RMS	color (color)

TABLE 4(D) View Profiles								
View name	*Image*				*Motion*	*AA Taxonomic Profile*		
	Grid Size	*Sym/try*	*Regul/ty*	*Complex.*	*Complex.*	*Persist/e.*	*Transit/n*	*Content*
DP	(varies)	MedHigh	MedLow	Low	Low	History	Discrete	Synthetic
TM								
- plot	Large	MedLow	MedHigh	High	Low	History	Discrete	Combined
- movie	Small	MedLow	MedLow	Med	High	Current	Discrete	Combined
RF								
- movie with stage delay	MedLow	MedLow	MedHigh	MedLow	High	Current	Discrete	Direct
- movie w/out stage delay	MedLow	MedLow	MedHigh	MedLow	Medium	Current	Discrete	Direct
-vars set to trans-ition smooth	MedLow	MedLow	MedHigh	MedLow	Medium	Current	Inter-polated (explicitly)	Direct

Deciding whether a view has an interpolated or discrete transformation profile was difficult. Weight changes are "naturally" very small (over the majority of a learning period) and therefore generally render smoothly. However, if the render cycle is, for example, once every 20 epoch cycles (set by the user), the resultant accumulated weight change would appear as a discrete update, since there is no mechanism ensuring interpolated updating.

Similarly, deciding whether to label the TM plot view and the RF view as having either a direct or combined content was determined to depend on how the view was "read." A direct content profile for these views seemed appropriate while users studied the separate elements of the view (local perception). On the other hand, a combined content profile seemed appropriate while users studied the "whole" view, e.g., the weights in the function of feature detectors.

The movies of delta weights are clearly current views (because the values eventually become zero at convergence); however, movies of weights project a

weight *evolution*. The final view is the accumulation of weight adjustments showing a form of history, although, not as discretely as a time-series plot.

In summary, these difficulties suggest two aspects of the three axis AA taxonomy: (i) that additional definition details are required for it to apply beyond the set of systems that it was developed on; and (ii) that to apply the three AA characteristics more "robustly" to the views designed for this project requires lower-level definitions of the views, in which specific variables and view configurations are considered.

4.3.3 Implementation Environment

NetViz is implemented in C for a VAX/GPX II graphics workstation, operating under a proprietary[1] VMS Workstation Graphics Software system (VWS). The VWS system provides a window-based user interface environment, UIS (User Interface System).

This environment was selected for several reasons:

- It provided the computational power deemed necessary for the problem.
- It provided a robust development environment, with powerful development tools
- The VWS graphics were "high-level" and supported by special purpose hardware.
- The required color capability was integrally supported.
- It provided a familiar environment and was easily accessible to this author.

A previous project of the same problem domain [Bubi88] concluded that a personal computer platform was computationally inadequate, as well as a challenging development environment. Experience on other projects suggested that the VMS operating system gracefully handles address faults characteristic of programs under development. Another useful VMS feature, the UIS windowing environment, enables multiple concurrent terminal sessions to be invoked

1 VAX, VMS, VWS, and UIS are trademarks of Digital Equipment Corporation.

facilitating modular development and testing. Consequently, tools, such as the integral symbolic debugger operate in their own window, which for this project, enabled the NetViz control screen to function normally in its own window. Finally, the workstation's large, high resolution screen was deemed well suited for the graphics needs of NetViz.

VWS is a library of object-oriented graphics routines, similar to CORE and GKS [Fole84]. Routines are provided for: windowing, drawing geometric objects, text, and images, modifying object attributes, color operations, interrupt event handling (e.g., mouse events), and display list control. Relative to X Windows and similar environments (Intuition on the Amiga), VWS windowing is more automatic, e.g., window "damage" cause by window overlap is handled by UIS, and the scaling/zooming of window contents is supported at a high level. Because VWS routines are not portable to other environments (such as to Unix operating with OSF/Motif), routines referencing VWS procedures have been encapsulated in order to facilitate an optional, smooth conversion of NetViz. Details are provided in Appendix B.

An original objective was to provide NetViz with a direct manipulation, menu based user interface; however, the effort required to develop this was determined to be a project in itself.

NetViz was implemented in C because of its portability, this author's familiarity with it, and its powerful "struct" data type, which facilitated an object-oriented implementation.

5

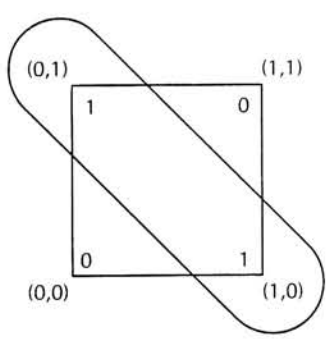
The NetViz Prototype System

5.1 A Basic Tour: Exploring the XOR Operation

The XOR operation is a common introductory problem studied with multi-layer networks trained with the back-propagation algorithm [Rume86a] [Dayh90]. Its operation is summarized in Figure 5-1. The solution classes to the problem are defined by two hyperplanes, as illustrated in Figure 5-1(b). Theoretically, a network with two hidden units and a non-linear activation function are needed to correctly learn the solution.

Input		Output
1	1	0
1	0	1
0	1	1
0	0	0

Figure 5-1(a). XOR Operation.



(b). Solution class mapping.

NetViz is invoked with the control file *xor.str* as the only parameter. The control file contains NetViz commands that define the XOR network structure, pattern-target pairs, initial key learning parameters, and two animated views: an Energy plot (ERG) and a Draftsman plot (DP) (details of the control file and other files are in Section 5.2.4, and in the Appendix). As NetViz starts up, a total of three

graphics windows appear on the screen and the NetViz control screen appears in the terminal window; this setup is shown in Photo 5-1.

The three overlapping graphics windows constitute components of the AA system. Windows can be repositioned on the screen, and can overlap each other (refer to Appendix B about window operation details). The DP and ERG windows are presently algorithm data views in their initial state. The Color Bar window is a static display of the user-defined system-wide color scale. In this example, color is mapped to the global correlation coefficient (GCOR) variable, in the DP view.

The control screen is sectioned in three zones: a command entry line at the top (preceded by the prompt "NetViz/BP"), a hierarchical command menu, and a data-form. The data-form is comprised of constants, variables, and data structures that are central to the learning algorithm. Many of the data structure values displayed in the form are also depicted in the animations. The primary use of the data-form is to examine specific data values at any time during learning, and to correlate the details of an animated view. The control screen window always receives keyboard input, regardless of its stacking position with respect to the other windows. The control screen layout is shown in Photo 5-2.

Training is initiated with the *seq_train* command (*seq_* meaning "sequential"). Note that the NetViz command parser accepts the minimum number of initial command letters that disambiguate the command. In sequential training, input patterns are presented to the network input units in the order they are stored in the pattern file; the pattern presentation order is the same each epoch. NetViz also provides training in which patterns are presented in a permuted order each epoch, using the command *ran_train* (*ran_* meaning "random"). Since the XOR network's training pattern set is small, the type of pattern presentation does not significantly affect the training outcome.

As training proceeds, changing algorithm data structures are depicted on the screen. In the control screen, weights, delta weights, biases, presented pattern names, and current epoch are some of the data structures continuously updated. Concurrently, the root mean square (RMS) of the total network error is plotted

against the current epoch (number) in the ERG view[1] window. In the DP view, small overlapping line segments are plotted in the center of tile (Photo 5-3). Although not discernable at this stage of learning (and at this standard scaling[2]), there are actually two line segments being plotted, representing three algorithm variables. The main, "weight-weight" line segment (and least detectable) is a time-series plot of two weights from the weight matrix connecting the input to hidden layer. In each tile a different pair of weights is plotted. The second, more noticeable *pulse line* segment, is plotted as a centered perpendicular bisector of a weight-weight segment; this line segment varies in width in direct proportion to the current RMS error value to which it is mapped. During the early learning stage, a patch of predominantly redish pulses develops, though periodically, some blue pulses also appear. This color change represents the mapping of the third data variable, the global correlation coefficient (GCOR) of the system.

For approximately the first 170 epochs, the RMS error level is flat in the ERG view. Correspondingly, the pulse line segments in the DP view appear to be maintaining a constant width, and are largely unvarying spatially, though over time their orientation changes, forming in some tiles a patch in the shape of an asterisk. The present DP view is depicting the infinitesimal movement of a conceptual error point in weight-space. In tiles developing circular patches, the error point is especially undirected in the representative plane, as the error point seeks a gradient to descend. Tiles in which the pulse line patch is in one dominant orientation depict the error point moving along a "z-axis" (i.e., basically outside the screen surface, since there are several axes in this weight hyperspace).

Around 180 epochs, weight-weight segments begin to lengthen, producing a trace that begins to take a general heading, different in each tile. Gradually at first, the RMS error curve in the ERG view begins to descend; correspondingly, the pulse line segments become shorter and spaced further apart. Some weight-weight traces change headings. As the training session reaches the termination criterion, the ERG trace descends sharply as the DP traces in each tile come to a

---

1 The term "energy" refers to the energy level of the system which defines the gradient decent function that the back-propagation algorithm is derived from. The energy of the network system is directly proportional to the RMS error.

2 A "standard scale" used in this thesis is the setting of plotting scale factors that enable the full plot to appear in view without being clipped, e.g., by the tile edges, over the course of learning.

tapered point. Traces in both views also develop rapidly when viewed dynamically. Frames of the final views are shown in Photo 5-4.

Both the dynamic and the final ERG view depict a characteristic convergence profile of the algorithm learning the XOR operation. The DP view does as well in this case, by way of the RMS error mapping. The DP view is primarily a visualization of the gradient descent function. Traces in each of the tiles depict the error point traversing over/through uneven error surfaces of the multi-dimensional weight-space. Each tile is essentially a "slice" of the hyperspace, enabling its visualization in two dimensions. The traces in each tile give clues to the texture of the local surface: crowded patches of lines, such as noted near the the center indicate a flat region where the error point is also seeking a direction with some downward gradient; the directed, more developed traces represent movement "down" the lowest gradient in the local plane. Changes in gradient "steepness" are represented by increased distances between perpendicular bisectors as well as by their decreasing width. Many of these cues are enhanced with the temporal component of the animation.

The ERG plot portrays an "averaged" profile of descent through weight-space. In the animation, DP weight-weight traces develop in different directions, while the ERG plot remains "flat." This is an example of how the views complement each other.

A single training session maps only one solution path to global minimum. Other solution paths can be explored by varying one of several algorithm variables, as well as the pattern presentation order. By changing, for example, the starting weights, the error point starts from a different location in weight-space (though relatively "near by"; investigations nonetheless show that very different learning outcomes result). The chance then exists for the error point to "find" another path to a global minimum.

Changing the learning rate and momentum rate can also lead to a different path; a higher learning rate translates to greater steps over the weight surface, raising the probability of finding a downward slope. Photo 5-5 shows compound ERG and DP views, showing the result of a repeated training session with a learning rate twice that followed above (earlier learning rate: 0.6; the second trial: 1.2). The second trace in each view was achieved by first issuing the command *reset no* to reset the network to its pre-trained state ("no" tells the system not to erase the

present animation view graphics). The next command, *get weights xor.wts,* restored the original starting weights (we explore the effects of changing only the learning rate). The learning rate is changed using *set param learn_rate 1.2.* The new rate and initial weights are displayed in the control screen. Training is started with the command, *seq_train.*

The second RMS error curve in the ERG plot begins an descending around the same time as the previous trial, but the descent is steeper. This is consistent with the property of an increased learning rate in the back-propagation algorithm. Each DP tile depicts two different error point paths, their end points in completely different regions of the weight-space. The presence of two solution paths to two different global minima is also expected, due to the multidimensionality of the weight-space imparted by the hidden layer units.

Photo 5-6 illustrates the capability in NetViz of magnifying a particular view. Shown is a single DP tile trace after 200 epochs of the previous problem setup. This view is magnified in two ways. First, the *view navigate* command was used to zoom in and pan through the "display space" of the view (see Section 5.2.2: *User Interface* for an explanation) to frame a single tile in the window. Second, the scaling factors for the weight axes and the width[1] of the RMS pulse were increased eight-fold by issuing respectively: *dp chg_xrange, dp chg_yrange,* and *view dp pulsecoef* commands. Before training was restarted, views were reset and cleared, and the command, *set nepochs 20* set the ending epoch number.

One of the strengths of NetViz is the support for animating several views concurrently. This is already demonstrated in the first examples of the basic tour. Photo 5-7 emphasizes this point, showing a Texture Map (TM) movie view as a fourth window. This view depicts the same network problem as the other views, but is portraying the evolution of weight error derivatives (WED) through arm and leg angle changes in each of the symbols. Activation values are portrayed through diameter changes in the symbol bases (circles). The TM view was activated with the command: *get animation tmtour.ani.* This invoked the animation view manager to process the window and view definitions contained in the file *tmtour.ani.*

---

1 The *pulse coefficient* determines the maximum width of the pulse line segment as a percentage of the tile width (world coordinates). Changing the z-axis scale, as the pulse is referred to, only sets the granularity of possible pulse line widths.

The TM movie view is especially susceptible to degradation of animation caused by three active views. Some improvement is achieved by turning off (*view win_onoff*) the continuous updating of the control screen. Setting the *render cycle* (i.e., number of epochs between frame updates) value of each view so that the TM movie is updated most frequently also helps.

When a view is no longer needed, it is closed, for example, using *dp close* to close the DP view. The process also removes the branch of DP view related commands. Should another DP view be subsequently opened, the DP commands are reinstalled. Ending a NetViz session is done with the base menu command, *quit*.

5.2 System Details

5.2.1 Objectives

Three major design objectives guided the development of NetViz:

- Adopt proven design elements of algorithm animation systems, such as structured algorithms, unobtrusive algorithm code annotation, algorithm and animation system code separation, and perceptually satisfying animation speed.
- Adopt a data-oriented/object-oriented design and implementation approach, mainly, data encapsulation, and data oriented functions.
- Provide the end user with interactive control over the data-graphics mapping and run-time view control.

These objectives were considered paramount in producing a tool suitable for exploratory work: one that provides adequate run-time control, and that is easy to extend. They encompass recommended strategies found in the published literature on algorithm animation systems and this author's experiences from prior system development work.

Algorithm Animation Specific Design Elements

NetViz takes the preferred approach for annotating algorithm code: a minimum of simple, animation directives strategically placed in the algorithm code. Consider the following code fragment from the module *bp.c* shown in Figure 5-2. The statements *Visualize* mark "interesting events" with respect to the algorithm's temporal flow. They are calls to a control module which determines

those views that are active and calls them to update their views. A view is active if its window is visible on the monitor screen. The parameter *displayflag* is the algorithm-code global mask of bit-flags associated with data structures and variables that are recently updated. In the code fragment (Figure 5-2), the flags WEDMASK and WEIGHTMASK are only appropriate at specific points, hence are passed to active views explicitly.

```
...
if(learnflag) {
   compute_wed();
   if(grain_string[0] == 'p') {
      Visualize (displayflag, frame);

      if(followmode) {
         change_weights_follow();
         Visualize ((displayflag|WEDMASK), frame);
      }
      else {
         change_weights();
         Visualize((displayflag|WEIGHTMASK), frame);
      }
   }
}
...
```

Figure 5-2. An annotated fragment of algorithm code.

Unobtrusive algorithm code annotation is greatly facilitated by a *structured algorithm*, i.e., an algorithm which encapsulates its major functions as discrete function calls. This affords the most direct means of annotating the algorithm, that of placing interesting event calls in the beginning of each module. Another benefit is that interesting event calls can be relocated in the code more easily during the phase of determining the best points that algorithm behavior is portrayed.

In BALSA and Tango, simple interesting event markers imply a communication path to a separate animation program running concurrently with the algorithm program. Such separation and asynchronicity is central to those systems for various reasons. One key reason is the separation of the algorithm code from the animation system code so that one system can be extended without "disturbing"

the other. By adhering to basic principles of modular architecture, NetViz achieves a functionally equivalent separation, and maintains an attribute of simplicity and compactness. Instead of passing interprocess message packets of visualization data, NetViz "passes" data through designated global structures.

Data Design

NetViz makes extensive use of "object-oriented" data structures, especially, the algorithm animation component. Although the language used (C) precluded true object-orientedness, view related units are described by "struct" (compound data structure) definitions, and data structure instances are manipulated by specific purpose functions. For example, symbols, cells, epoch columns, and a *tminfo* type, in the Texture Map view, are defined as a hierarchy of structures. Object-oriented data structures are also part of the user interface subsystem. Extensive use of dynamically allocated memory to create structure instances allows networks of arbitrary size and topology, and multiple views to be established dynamically. The primary benefit of an object-oriented design was the affordance of quick extensions to the system with new views. Existing structures served as templates for the data structures of new views.

A key design element in NetViz is the access to algorithm variables by the animation view routines. Since NetViz is a single program, the addresses of designated (by the user, through a view definition) algorithm variables are stored in designated fields of each view's data structure. Functions in the animation system treat these mapped variables strictly as "read-only." A second key element is a semi-global flag-mask (it is global to the algorithm program, and passed to the animation system monitor) that describes those algorithm variables that have changed since the last animation frame. The use of address mapping and update flags ensures that the most current algorithm values are accessed. More important is that copies of algorithm data structures in the animation system and their maintenance is avoided, thus maintaining the robustness and extensibility of NetViz.

User Control

Two key elements (which are also examples of the object-oriented design in NetViz) of the user interface subsystem are the *command table* and the *variable table.* Both are the basis for several routines that constitute the menu-based command line interface. The command table is primarily used to invoke user

specified commands by referencing the corresponding function address. The command table is also used for the run-time construction of the command menus displayed in the control screen below the prompt. The variable table provides interactive access to and manipulation of registered system variables. By design, the access name of the variable can be different than the variable's name in the program code. Associated functions enable commands and variables to be dynamically added or deleted.

5.2.2 System Architecture

NetViz is comprised of four primary components:

- user interface
- algorithm generator
- animation view generator
- playback controllers

These components are built together into a single executable process. Named memory regions are used for limited inter-component communication, and to a lesser extent for intra-component communication. The interrelationship of the three is shown in Figure 5-3.

Each of these three components have more specialized subcomponents. Generally, the subcomponents involve operations transparent to the user, such as initialization, memory management, window management, and error handling. The following sections detail the primary system components and their interactions.

User Interface

The NetViz user interface is a command line driven, graphical interface. It has three major components:

- a command line interface
- the control screen data-form
- the windowing interface (VWS/UIS)

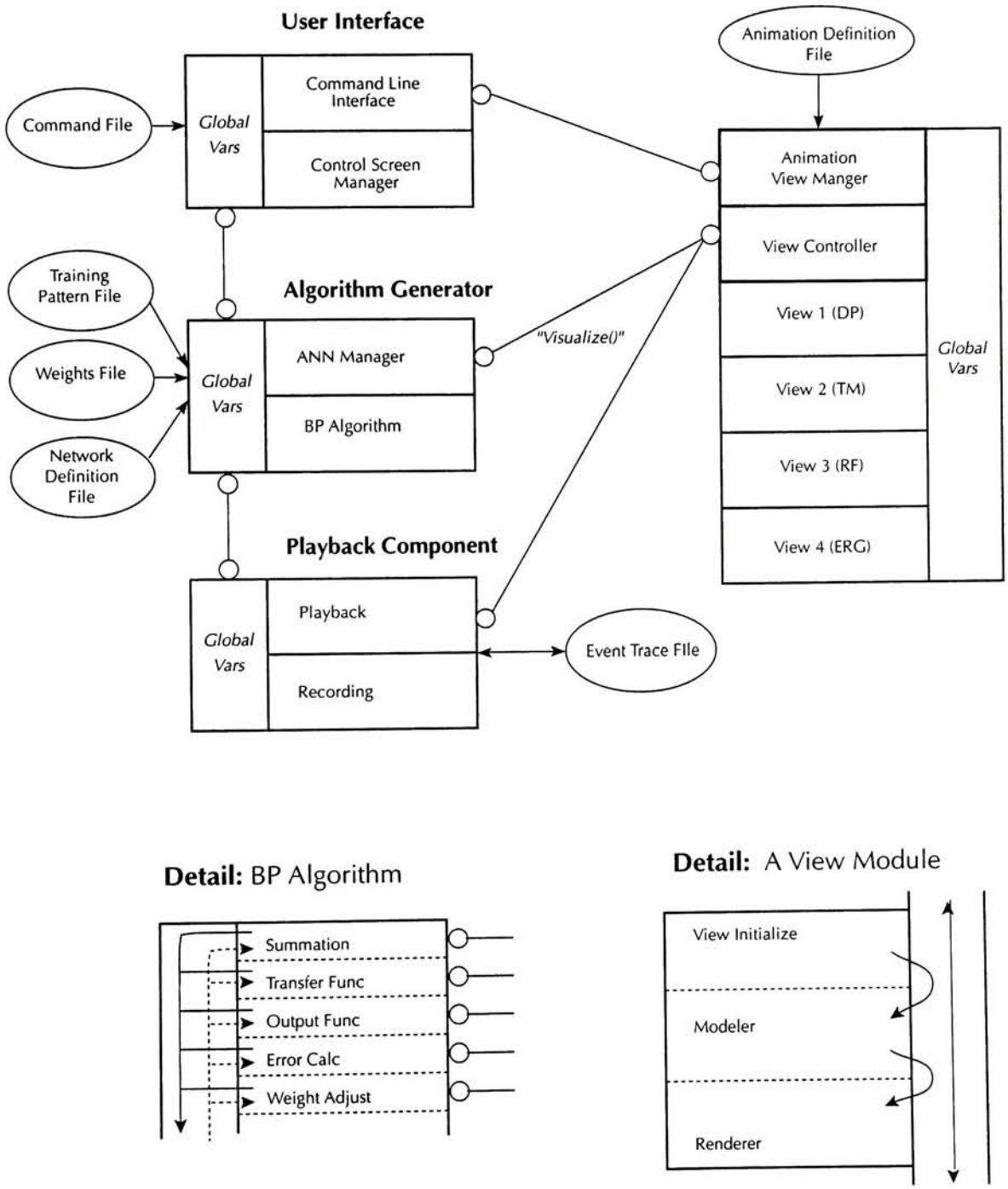


Figure 5-3. NetViz architecture.

Command Line Interface

The command line interface is derived from a part of the PDP software suite. The single command input line in the top most row of the control screen, together with the command menu below it, constitute the command line interface. This and the rest of the control screen is controlled by curses screen input/output routines (allowing direct screen coordinate I/O). The command set is hierarchical; normally, base level commands are displayed in the command menu. Entering the name of a sub-menu (represented as commands with a "/" suffix) causes the commands of that sub-menu level to be displayed. The command line parser can act on partial commands, provided they can be disambiguated. A command line can specify a full or partial navigation path, with or without parameter values. The user is prompted for missing command parameters. When a command or parameter is unrecognized, a standard formatted error message is displayed that includes the name of the system module where the failure occurred .

The NetViz command set involves these general categories:

- *get* — provides routines for installing animation views, networks, pattern sets, and weight sets from description files.
- *save* — provides two commands, one for saving a snap-shot of weights to a file, and another saving the prevailing image of the active view in a file.
- *set* — encompasses several sub-menus that categorize parameters and algorithm variables that can be altered at run-time. These include: *set/mode/* where the weight update mode is changed; *set/state/* where the values of variables such as epoch number, GCOR, and RMS can be altered; and *set/weightstuff/* where values of variables associated with unit connections can be altered.
- *view* — provides functions that affect the active view and window, such as window cycling, view zooming and navigating, and window background color.
- *tm/, dp/, rf/,* and *erg/* — are sub-menus of commands for controlling the Texture Map, Draftsman Plot, Receptive Field, and Energy plot views, respectively. These menus appear only if the corresponding view is open. Common commands within these menus allow plotting scales and render cycle changes, view resetting, and view closing.

All NetViz commands are covered in detail in Appendix B. Four fundamental commands and interaction mechanisms that provide important user control are summarized below.

- *Interrupt processing* — allows the user to stop a lengthy learning or playback process by typing "Control-C," where "control" refers to pressing the control key on the keyboard. The system pauses processing at the end of the ongoing processing cycle, and prompts the user to either, continue from pause-point, to fully stop the process, to single-step through the algorithm or to temporarily begin a new command level (this is a recursive entry). Interrupt processing is very useful for changing network parameters, such as learning rate, in the middle of a training session.
- *Single stepping* — is a setable mode that allows the user to control the pace of the algorithm, one epoch cycle at a time. Essentially, when this mode is on, the system interrupts itself; therefore, the same prompt that appears during *interrupt processing* appear here, and the same options for continued action apply.
- *Seq_train* and *ran_train* — invoke sequential and random (permuted) training respectively. Sequential training presents the training patterns to the input layer in the order that they exist in the file, every epoch. Random training permutes the order of training patterns each epoch. Training proceeds for the number of epochs set (using *set epoch number*) or until the system RMS reaches error criterion.
- *Reset and newstart* — reset the weight and bias data structures to initial conditions of basic noise values. Newstart seeds the pseudo-random number generator to produce a different set of values each time. Reset initializes weights and biases to the same random values determined at the invocation of NetViz or the last newstart. Reset also sets the epoch cycle counter to zero and optionally erases the active views (user confirmation is required). When the desired starting weights are in a file, a reset is done first, followed by a *get weights* and the file name. Loading known weights allows replicated testing.
- *Test_all* and *test_one* — allow all or one (by name) pattern to be tested for classification using the prevailing network weights. Test_all effectively performs a test_one in a single-step mode, reading each pattern in the file in sequence. The TM and RF views provide the necessary feedback to verify if a pattern is correctly classified; the data-form in the control screen can be used as well, by examining the target output values and the corresponding calculated activation values.
- *Refresh* and *screen_onoff* — control the display of information in the control screen. Refresh repaints the data-form with the prevailing algorithm values and certain system flags. Screen_onoff toggles between normal, full data-form updating and limited data-form updating (the update of multidimensional data structures is circumvented). When the data-form

update if "off," a refresh command invoked in the middle of a training session, will display the latest data-form information.

Control Screen Data-Form

Below the command menu lines in the control screen is the a *data-form* of static labels and dynamically updated values. Most of the dynamically updated values are usually also animated in one or more views. Thus, the user can optionally correlate (in most circumstances) actual values with the contents of animated views. The data-form primarily evolved as a means of debugging views in development.

The top half of the data-form is a layout of simple algorithm constants/ coefficients and variables, such as learning rate, error criterion, RMS error, pattern name and epoch number. Generally, the algorithm constants, listed on the left side, are not updated during learning, unless a user changes a value. The other variables in this area are updated each epoch, and so are not affected by any view's render cycle setting. On the right-most side of the top half are the key algorithm data structures, biases, activations, and target patterns. The last two data structures provide a means for testing the trained network, by comparing computed output unit activation with the target value. (This information can also be viewed in TM and RF views.) In the lower half of the data-form are full layouts of weights, delta weights and WEDs. Their layout (i.e., the number of rows and columns taken up by each) is determined when the control screen is initialized (this is why the network needs to be defined before control screen initialization). The layout of these data structures with respect to a topological network view is illustrated in Figure 5-4. An understanding of the mapping is necessary to make changes to specific weight values during run-time. Data structure related variables are updated in conjunction with a render cycle of an animated view, in order to reduce the overhead of refreshing a potentially large layout. If there is no interest in the data-form, the updating of all variables, except epoch number, pattern name, and RMS can be disabled.

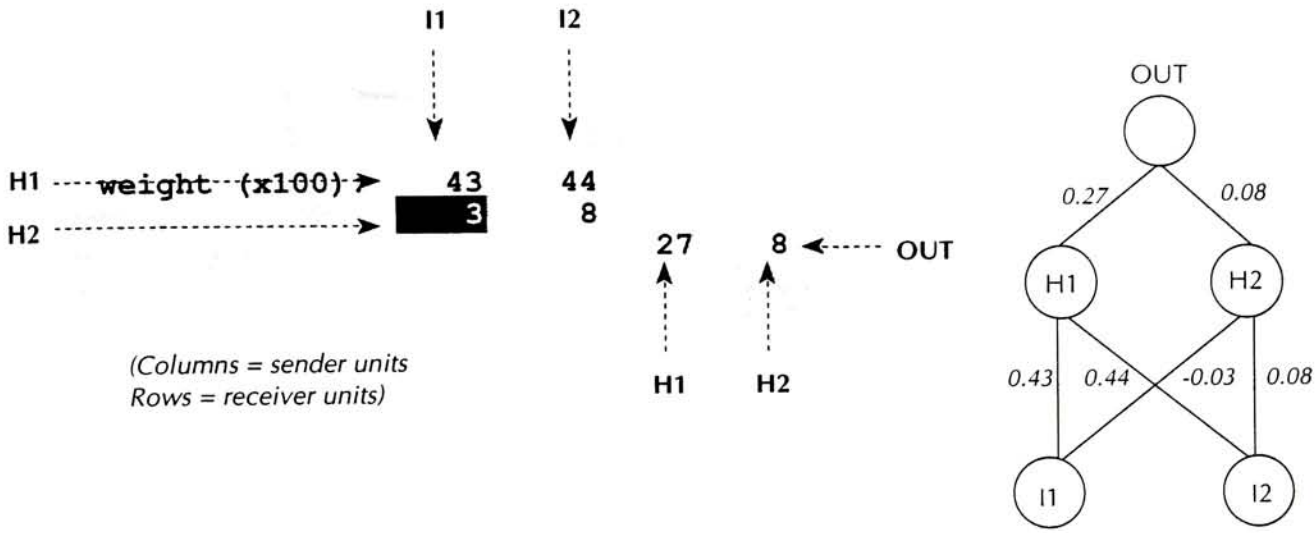


Figure 5-4. Mapping between control screen weight layout and a topological view.

Windowing Interface (UIS/VWS)

NetViz displays animated views in windows opened by VWS routines (VAX Workstation System), that are subsequently managed by the UIS (User Interface System) window manager. UIS windows can arbitrarily overlap one another, can be repositioned on the screen, and can be resized, using the workstation mouse. Resizing a window does not scale its contents.

UIS recognizes only one *active window* at anytime, that is, with respect to recognizing keyboard entry. In a workstation set up with multiple terminal windows, the window in the foreground (also showing a highlighted "KB" icon in its title bar) gains control of keyboard input. In a typical NetViz setup, in which only one terminal window is required, the NetViz control screen always recognizes keyboard input, even when overlapped by view windows. Subsequently opened view displays do not vie for the keyboard. Internal to NetViz, is the concept of a *current animation window*. This window is the "recipient" of commands such as *navigate* (zoom, scroll), *replay* (re-execute the display list), and *wincycle*. The last command is the means of transferring the current animation window status to other open views.

At the system level, creating an animated view involves a hierarchy of "spaces," each with dimensions specified in the animation definition file. The hierarchy

starts with a *display space*, specified in "world coordinates," one or more *window frames*, and a corresponding number of *viewports*. Details of this hierarchy are described in Appendix B. The hierarchy of spaces and frames is the basis of displaying a view on the workstation screen, scrolling/panning and zooming in and out of a view, and of displaying the same animated view through more than one window (to UIS a window is a *viewport*), each potentially sized differently.

An important concept associated with the display space is the *display list*. A display list is a memory resident instruction stream of all rendering commands issued for a particular display space. It is the means by which UIS refreshes the contents of a window affected by zooming, scrolling, and window resizing. Saving a view equates to writing the display list, between header and trailer data, to a file. For long and complex animations, several long display lists can significantly degrade system performance. A NetViz command thus allows the display list of a specific view to be "turned off," whenever visualizations do not require view manipulation (it can be turned on again). Animation proceeds normally, with the only difference being that anything rendered after the display list was last on, is erased, if (say) zooming in on the view is attempted.

Since a display list is in a way a memory-based "record file," (see the section on Playback, below) a command is provided to force its "replay," one or more times. The resultant animation is very quick and cannot be controlled (it is of modeled data). The depiction of evolutionary trends, such as acceleration and deceleration is nevertheless compelling.

Algorithm Generator

The algorithm generator is made up of two subcomponents: (a) the ANN manager, and (b) the learning/recognition algorithm. A major task of the neural network manager is the set up of a network described in a *network definition file*. An ANN is described by several parameters, predominantly by: the number of units in each layer; their interconnection pattern; the weight update constraints; the inclusion or exclusion of biases, and the set of training patterns. Several other parameters that define training details are usually also contained in the network definition file. Proper set up of the network precludes the initialization of the

other two components; once the prerequisite setup is complete, the *network_defined* global state flag is set, enabling the rest of the system to initialize.

Secondary neural network manager tasks essentially occur during run-time. Included are the resetting of weights to the same initial values, generating a new random set, saving a snap-shot of weights to a file, and initializing weights from a saved set.

The feed-forward and back-propagation algorithms are main parts in the learning/recognition algorithm subcomponent. This is the generator of data that is modeled and animated. As shown in Figure 5-3, "Detail: BP Algorithm," the algorithm is structured into discrete modules representing the fundamental back-propagation learning functions. The modules *train*, *test_one*, and *test_all* encapsulate the back-propagation learning and input pattern testing, respectively. They are the only modules that are annotated and thus, communicate with the animation view generator.

Animation View Generator

The animation view generator is the largest and most complex component of NetViz. Its hierarchy of three subcomponents is:

1. an animation view manager
2. a view update controller
3. one or more views

Animation view manager — is analogous to the neural network manager of the algorithm generator. The animation view manager's function is to set up data structures that are used by subsequently opened views. Three key data structures involved are: the window list, the color table, and the lines table. These are described in more detail in Section 5.2.5. The animation view manager's initialization is directed by the first *animation control file* (see Section 6.2.4) processed. Once the prerequisite setup is complete, the system wide *view_defined* state flag is set, and the installation of requested views is enabled.

View update controller — is the procedure called through the interesting event statements placed in the algorithm code. It first determines which views are

active, and then whether a view update is scheduled, according to the render cycle setting for the particular view. The render cycle is a user-defined value that specifies the number of epoch learning cycles between view updates. The view update controller then calls the modelers and renderers of the views that are active and scheduled for update. As part of this procedure, the view update controller also causes the control screen to be refreshed. When a view is playing back a record (from a record file) of a previous animation, the view update controller directly handles modeling and rendering of the view (i.e., interesting events are not involved).

Views — Each view is comprised of three basic subcomponents: view initialization, view modeler, and view renderer. Figure 5-3, "Detail: View Module," depicts this breakdown and the subcomponents' interrelationships. A view initialization routine is an extension of the animation view manager, that deals with view specific setup tasks. A view is *opened* according to commands and parameters specified in an animation control file. This involves first opening a standard workstation window (empty), of user-designated dimensions. Then, the view initialization module initializes relevant data structures, in part by establishing links to algorithm variables specified to by visualized. Other initialization tasks include: establishing view element dimensions, requisite colors and line weights for static view parts, and computing the plotting scale coefficients. Finally, the initial view is modeled, and rendered in the window.

During learning or playback, a view modeler transforms "current" values of algorithm variables. Transformations involve calculating plotting coordinates, symbol size, angle, color, and line weight. The result is essentially a memory-based view update-frame. The corresponding view renderer subsequently "draws" the update-frame in the view's window.

Playback Controller

Two of the modelers in NetViz, DP and RF, are capable of recording the pre-modeled algorithm variables in a *record file* for future playback. Recording involves writing view-specific pre-modeled algorithm values to a file, structured as update-frames. Capturing pre-modeled values provides greater visualization flexibility during playback, such as using different plotting scales and different

symbol mappings. Only those update-frames originally modeled and rendered can be recorded. The user can begin a recording session at anytime during a normal algorithm animation session, and can either specify a number of update-frames to capture or can interrupt the learning process and manually terminate the record session.

Playback is invoked by the user to drive the animation view generator, as an alternative to the algorithm generator. The playback capabilities in NetViz are basic; two playback controllers, one for each of the supported views, operate by initializing the data structures required by the view modeler, with data read in from a record file, and calling the animation view controller. This operation is repeated for each update-frame contained in the file. A playback animation proceeds at the render cycle set during recording, as long as the render cycle during playback is set to one. At higher render cycles an animation can be played back "faster," though update-frame transformations are more discrete. A recorded animation can be designated to playback the entire file or a specific number of frames; in either case the user can also interrupt the animation and terminate it. Through the use of NetViz control files (see section below), playback animation loops can also be set up.

Playback by-passes the overhead incurred by the algorithm generator. Since it requires a record file, this implies that the played back view was once visualized (though potentially unattended) through the algorithm generator. The advantage of record/playback is in animating long learning processes (>1000 epochs), in which the render cycle was set for sparse updating during recording. ("Sparse updating" describes animation in which views are updated very infrequently throughout the course of training.)

5.2.3 Implementation Overview

Figure 5-5 is a structure chart showing the major modules of the NetViz prototype system.

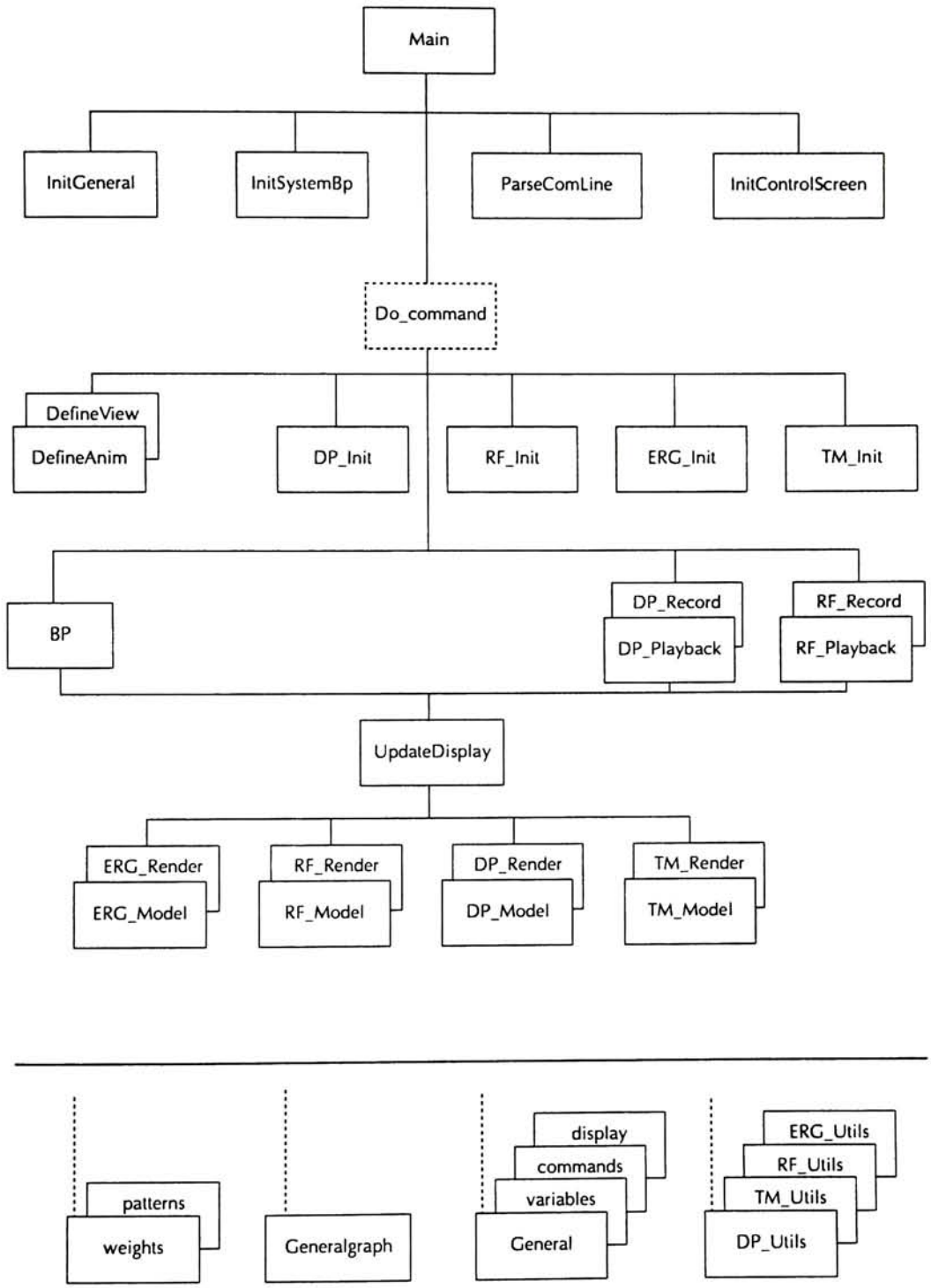


Figure 5-5. NetViz System Structure Chart.

In Figure 5-5, closely related modules are shown overlapping each other. The topmost module, *main*, is the system root module. In the second row are three main initialization commands, that are called once at system startup. The third row of modules also depicts initialization routines; however, ones that are called at run-time, either through user commands or the control files. The row below the module *UpdateDisplay* comprises the heart of the system: the main algorithm module, *bp*, four view modeler/renderer module groups, and two record/playback module groups. At the bottom of the figure are various modules containing general purpose procedures and functions called by the higher-level modules as well as from the lower-level modules. Overviews of each of these modules are included in Appendix C.

5.2.4 External Interfaces: Control Files

As is shown in Figure 5-3, there are a number of initialization, control and save files in the NetViz architecture. Highlights of each follow; templates and other details are provided in Appendix B.

Command File

A command file is an editable file containing complete NetViz commands. The commands must be valid for the prevailing context when the command file is processed. For example, a command file containing DP view specific commands will not run unless a DP view is already open.

A special command file is the start-up file, which is included as a parameter when invoking NetViz. The only command that makes this a required file is network initialization (specifying a network description file). Typically, however, other start-up commands are included, such as the installation of training patterns, installation of one or more views, and the initialization of various algorithm parameters.

Network Definition File

A network definition file has two required sections: (i) *definitions*, where the number of units in each layer is stated, and (ii) *network*, where the network interconnection pattern is defined, together with the type of connection

weighting (e.g., random, ±0.5; unmodifiable at +1.0, etc.). An optional *constraints* section allows special character encoding of specific weight values and ranges to be used in the network section (hence it must precede that section if used). A fourth section, also optional, is *biases*, where bias to unit links are made, and bias constraints are assigned.

Pattern File

A pattern file is a list of input-target pattern records. Each pattern record's first field is a label that is displayed in the data-form when the pattern is presented at the input layer. It is also used as a parameter when testing the network with a specific pattern. The second record is a series of input values separated by white space, and equal in count to the number of input layer units. The last field is a similar series of target values, equal in count to the number of output layer units. Both input and target values are handled as real numbers. Fields are separated by white space and by the number of values. Labels are alphanumeric without intervening white space; the first character must be alphabetic.

Animation Definition File

An animation definition file has five sections, the first two required before the very first view is created. (i) *Colors*, where the system-wide color table is defined by specifying the table size and hue, saturation, and value ranges. (ii) *Lines*, where a list of line style patterns is defined, as part of the system-wide lines table. The system builds the table with an equal number of increasingly thicker line styles (iii) *Layout*, where primary and secondary view windows are defined. A primary window subsequently contains a new animation view. A secondary window becomes associated with the display space of the current animation window, and hence, displays the same view; one intended use is to display a magnified portion of the view. (iv) *Animation*, which encompasses three different definition templates, one for each of the main views (DP, TM, and RF). In general, each of the templates contains commands for defining tile/cell dimensions, render cycle, and variable-to-symbol mappings. The final section, *ergplot*, is where an ERG plot view is defined, using a template analogous to those under the *animation* section.

An animation definition file can contain any number of sections. For example, when a file containing only the *color* section is processed, the existing color table

is superseded by the new definition. A new view is created using a file containing a *layout* section and an *animation* section that defines the new view.

Weights File

A weights file is an editable list of real values representing weights and biases. Such a file can have "hand crafted" values that subsequently initialize an ANN under study. A file can also be created from within a NetViz session, through the *save weights* command. In this way, snap shots of the weights and biases at different epochs can be captured, modified, and reloaded in a later session.

Record File

There are two types of record files in NetViz, one for DP views and one for RF views. The files have in common: a header followed by a list of records with values that constitute pre-modeled view update-frames. The header contains information used mainly to verify that the data structure dimensions of the playback view match the size of data records in the file, and that the recorded values are of the same variables set up in the playback view.

5.2.5 Major Data Structures

A central implementation strategy of NetViz is object-oriented design, primarily through data encapsulation. This section provides highlights of major system data structures, from both the algorithm generator and animation view generator components. Refer to Appendix B for details of these and other data structures.

Command Table

The command table is an array of records constituting the user-accessible command set. A command record includes: the name of the command, a pointer to the function that is called (when the user enters it on the command line), the menu level (in the hierarchy), and a pointer to the parameter passed to the command function. Commands that invoke sub-menus, are records with a pointer to the *do_command* function, the same function that waits for user input from the command line, and the menu level code as its parameter. The command table is generally built/appended to in initialization routines. Commands with the same name and different parent menu are allowed (e.g., *dp close* and *tm close*). Some key functions that operate on and with the command

table include: *do_command*, which parses user input and, (recursively) invokes the requested function(s); *install_command* and *remove_command*, which appends new commands to and removes exiting commands from the table, respectively; and *do_help*, which produces a sorted and formatted command list of the current menu level, below the command line.

Variable Table

The variable table is a linked-list of records that provides run-time user access of internal variables, as well as access by functions that are not in the local or global scope of the desired variable. Each variable table record contains an access name that can be different than the symbolic name used internally. This is primarily useful for accessing specific elements of an array by a mnemonic name. Each record also contains a variable type code, the pointer to the variable, two fields defining the lower and upper bounds of vector and matrix type variables, and a field for the menu level code of the variable. Variable type codes include scalar, string, vector, and matrix variables. These codes are used by the function, *change_variable*, a control function that is called in a way analogous to *do_command*, to modify the value of the variable. *Change_variable* generally prompts the user for any required input (typically showing the existing value and requesting a new value in the case of simple variables), such as the valid sequence and range of indices when the variable being changed is a matrix. Other primary routines that operate on and with the variable table include: *install_var*, used to register a new variable in the table; *remove_var*, used to remove a table entry (such as when closing an animation view); and *change_variable_length*, used to change the upper and lower bounds of vector type variables.

Network Structures

The key network structures are vector arrays for: *weight*, *delta_weight*, *WED*, *posi_contraints*, *neg_constraints*, *epsilon*, and vectors for *activation*, and *bias*. Fundamental to the manipulation of these data structures is the internal perspective of the network. Each network unit is labeled by its distance from the first input unit, as shown in Figure 5-6. A vector of weights, delta weights, etc., is associated with each "receiving" unit. Receiving units are the hidden units, which receive weights (etc.) from the input units, and output units which receive weights (etc.) from the hidden units. The length of each vector is equal to the

number of "sending" units. Thus, following the illustration in Figure 5-6, *weight[3][0]*, is the weight of the connection from the first input unit to the first hidden unit; the complete weight vector to the first hidden unit is *weight[3][0..2]*. Since there are no "senders" to input units, and array variable declarations by default begin with element "0," the weight (etc.) vectors from "0" to the "number of input units" are null.

The activation and bias vector indices correspond to the unit counting convention shown in Figure 5-6. At the start of the feed-forward phase, input patterns are actually loaded into the activation vector, from index "0" to "number of input units." However, correspondingly indexed biases are ignored, since input units do not have associated biases.

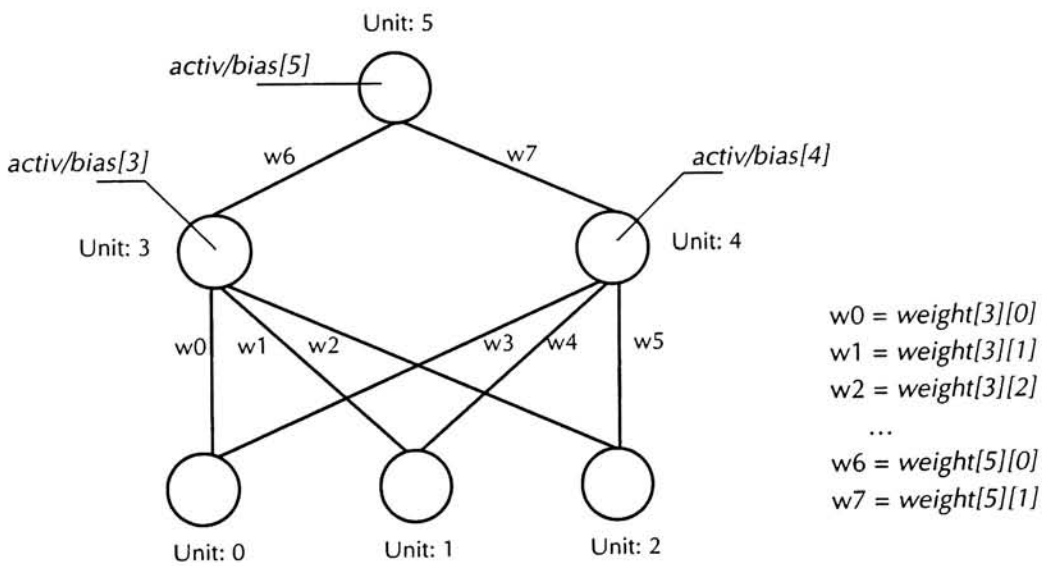


Figure 5-6. Internal structure of a network.

Color Table

The color table is defined by two structures, one being an instantiation of the *colortab_type* record, and the other being a VWS/hardware table of color "registers." The *colortab_type* record contains fields for: the maximum number of available registers, the color table name (used by VWS), the actual table size (specified by the user), "floor" and "ceiling" color register indices, a pointer to the

hardware table (the registers are actually a block of memory), and a set of nine fields that define the color map. The floor register isolates several registers reserved for system global colors, such as the background color in register 0, and the main foreground color in register 1. Colors used for general view layout by DP, TM, RF and ERG views are allocated above the ceiling register. Six of the nine color definition fields contain the lower and upper range information for hue, saturation, and value, that are read-in from the *colors* section in an animation definition file. The hue ranges from 1° to 360° (red is at 0°/360°). Saturation and value ranges are percentages from 0 to 100%. The three remaining fields are the increments of each range, computed by dividing each component's absolute range by the specified table size. Increments can be either positive or negative, depending on the mapping desired for the associated variable (e.g., red to blue versus blue to red mapped to positive to negative).

The color table builder performs a modulo operation on ranges that exceed the normal upper bound. Therefore, a color map can be defined that cycles twice through the hue range (0° to 720°), while it decreases gradually in saturation (e.g., 100% to 30%).

The system can process a new color definition at any point during run-time. The direct change in the color registers is immediately seen in traces already rendered. A caveat to this capability is that subsequent color definitions should have the same table size as the original definition to avoid partial mappings or infringing on static color definitions (i.e., those above the original ceiling index).

Windows and the Window List

Each viewport on the workstation screen containing an animation view is associated with a NetViz *window record*. Window records form nodes in a form of binary tree, illustrated in Figure 5-7. A header structure, *window_list*, maintains pointers to the first and "last" window records, and a count of window records. Each window record contains many fields that can be categorized as: pointers to VWS/UIS structures (e.g., the display space), window identification (name, type, etc.), dimension, scaling, and origin information, algorithm variable-symbol mapping information, and pointers to next and previous windows.

The window list recognizes two types of windows, "different" and "same" windows. A "different" window is the first instance of a particular view on the screen; hence, a window list can have at most four "different" window records. A "same" window is one that shares the same display space with the first, "different" window (i.e., parent window) of the same view. A "different" window can therefore have any number of "same" windows.

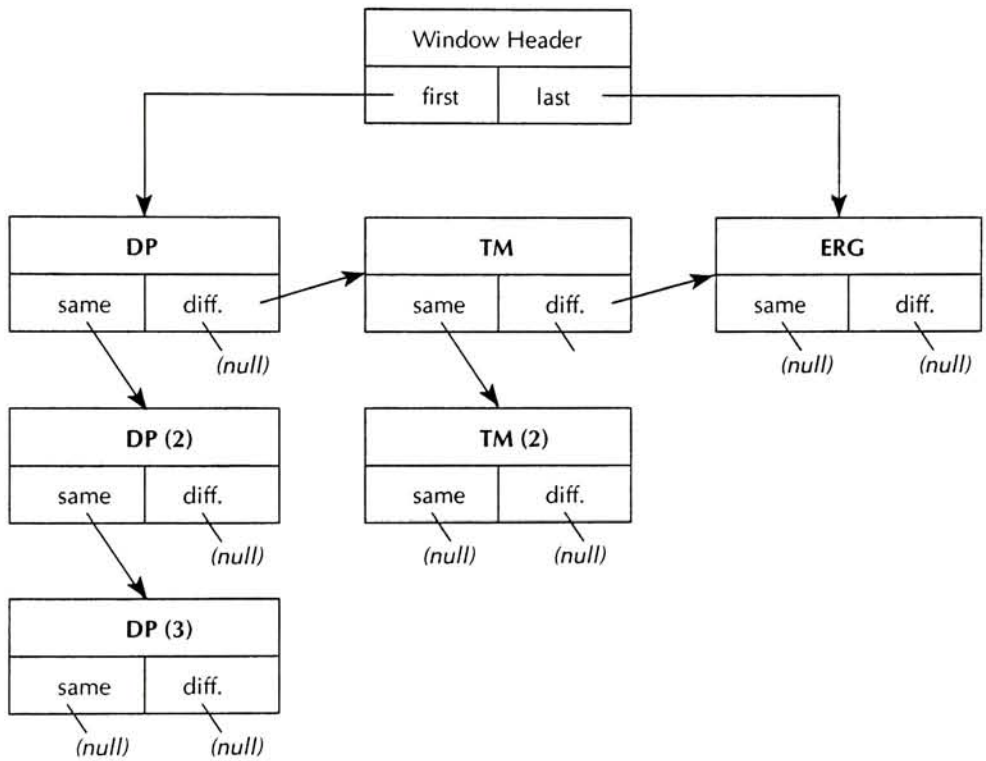


Figure 5-7. Structure of the NetViz window list.

There are many functions that operate on and with the window list. Central to them is the global pointer, *cur_anim_window* which indicates the foremost window of a stack (the control screen is not part of the window list). The function *read_layout* instantiates and initializes new "different" and "same" windows, using data from an animation definition file. With the help of *add_win_at_end*, it also adds the new window record accordingly to the window list. The of routines *CloseWindow*, and *rem_same_win* remove all windows of a specific view (a constraint of the system is that a *view* is closed, i.e., the parent window, rather than an arbitrary "same" window). *Wincycle* moves the

cur_anim_win pointer to the "next" window, either a "same" or "different" one, depending on the argument provided. Finally, the function *navigate* provides the means to "zoom in," "zoom out," and scroll left/right, up/down in the cur_anim_window.

View "info" Structures

The root structure of each of the four views is generically an *info* structure. Although each is optimized for the peculiarities of its view, there are five groups of fields in common. Each structure has a group of color and line style attribute definitions used for rendering the static "frame" of the view. A second category encompasses count, dimension, and location information of the view components (e.g., tile count, base tile size, location, etc.). A third category encompasses variable-symbol mapping and re-expression state information. A pointer to the view's parent window in the window list constitutes the fourth category and the fifth category is a pointer to view structures defining more view specific detail (e.g., *DPtile* structure for the DP view, *TMcolumn* in the TM view, etc.). Details of all view structures are in Appendix B.

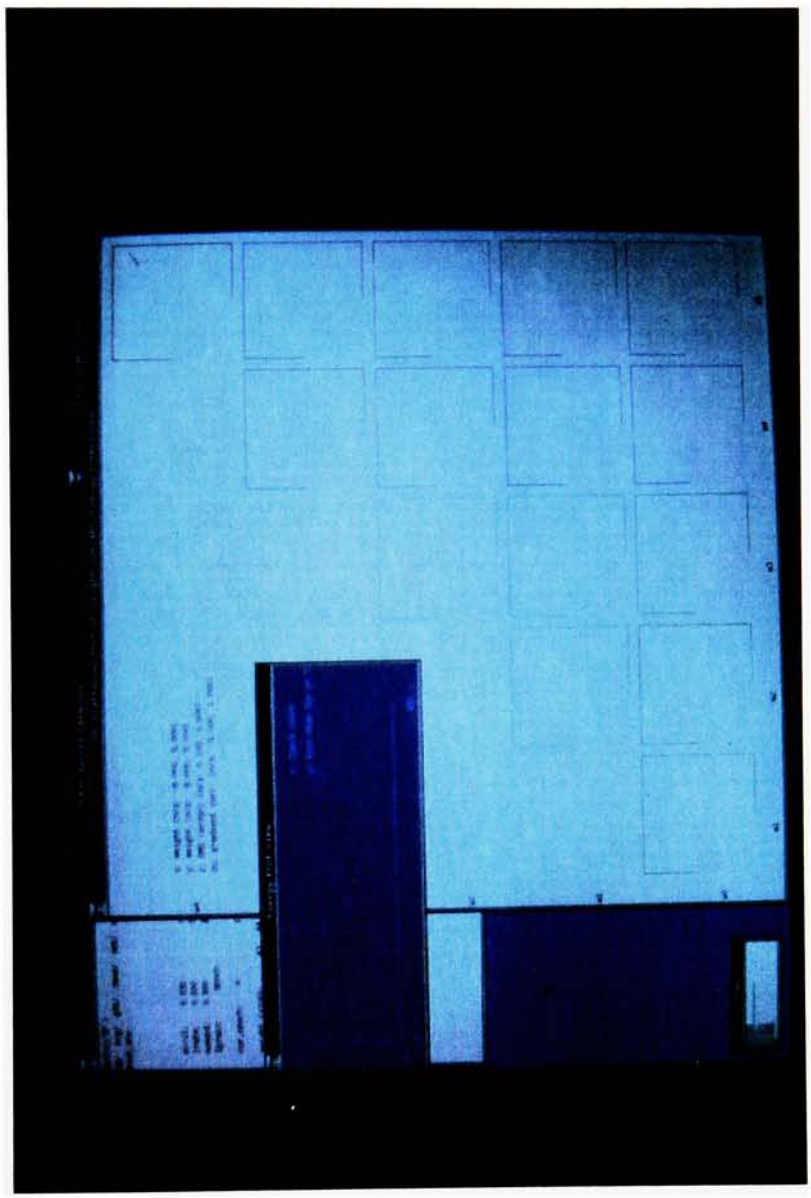


Photo 5-1. Example NetViz startup screen.

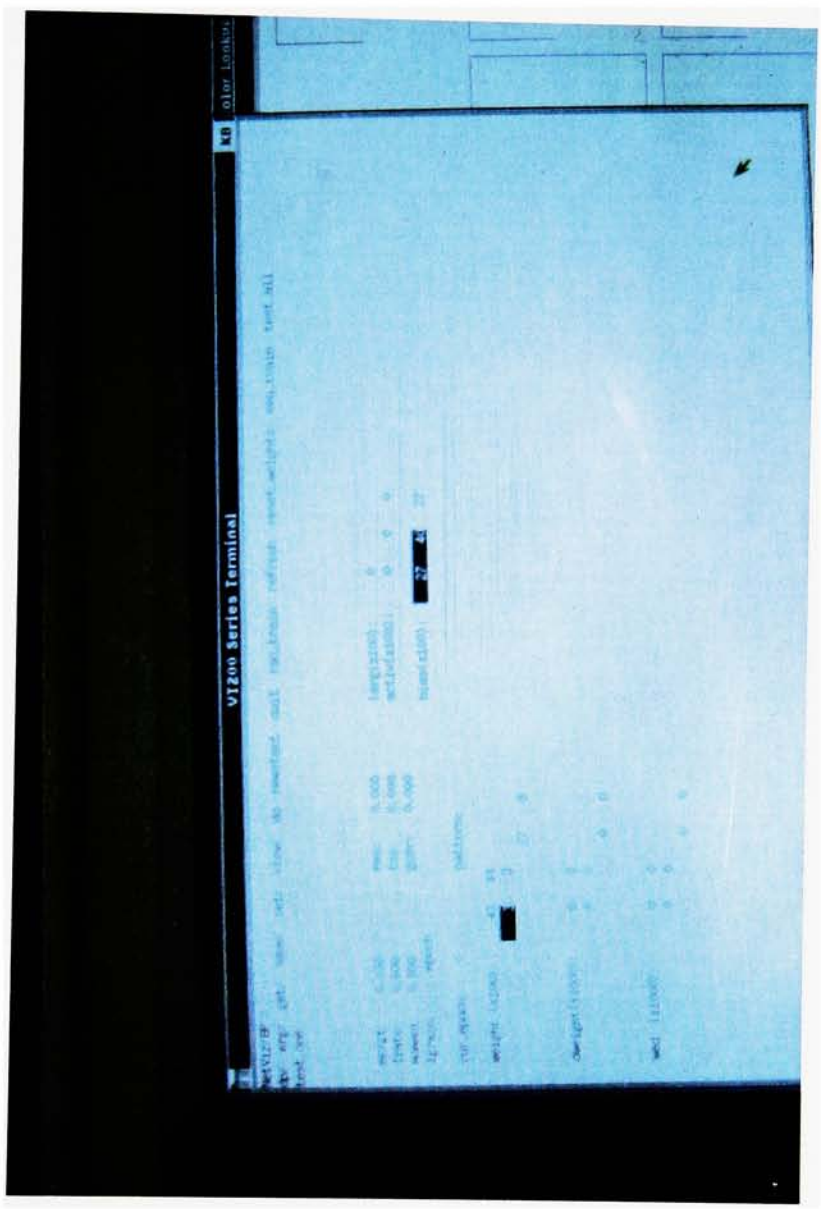


Photo 5-2. Control screen layout.

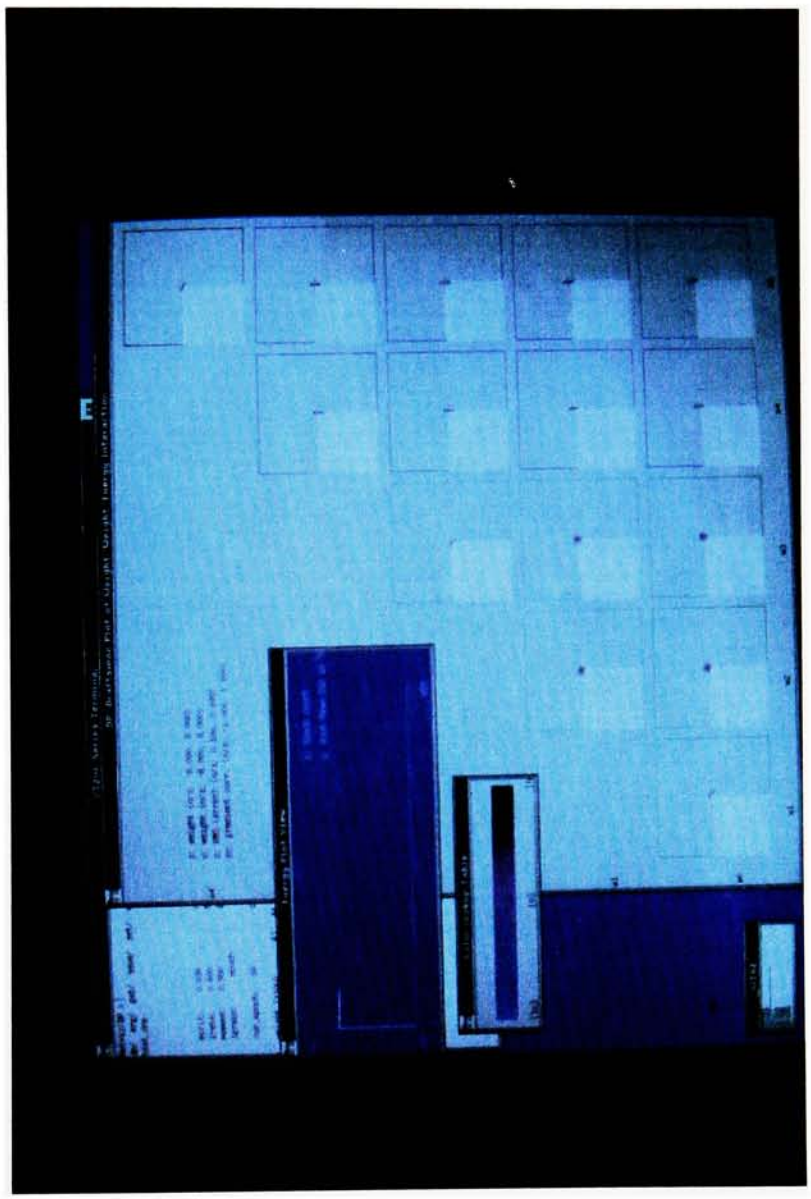


Photo 5-3. XOR learning at 50 epochs.

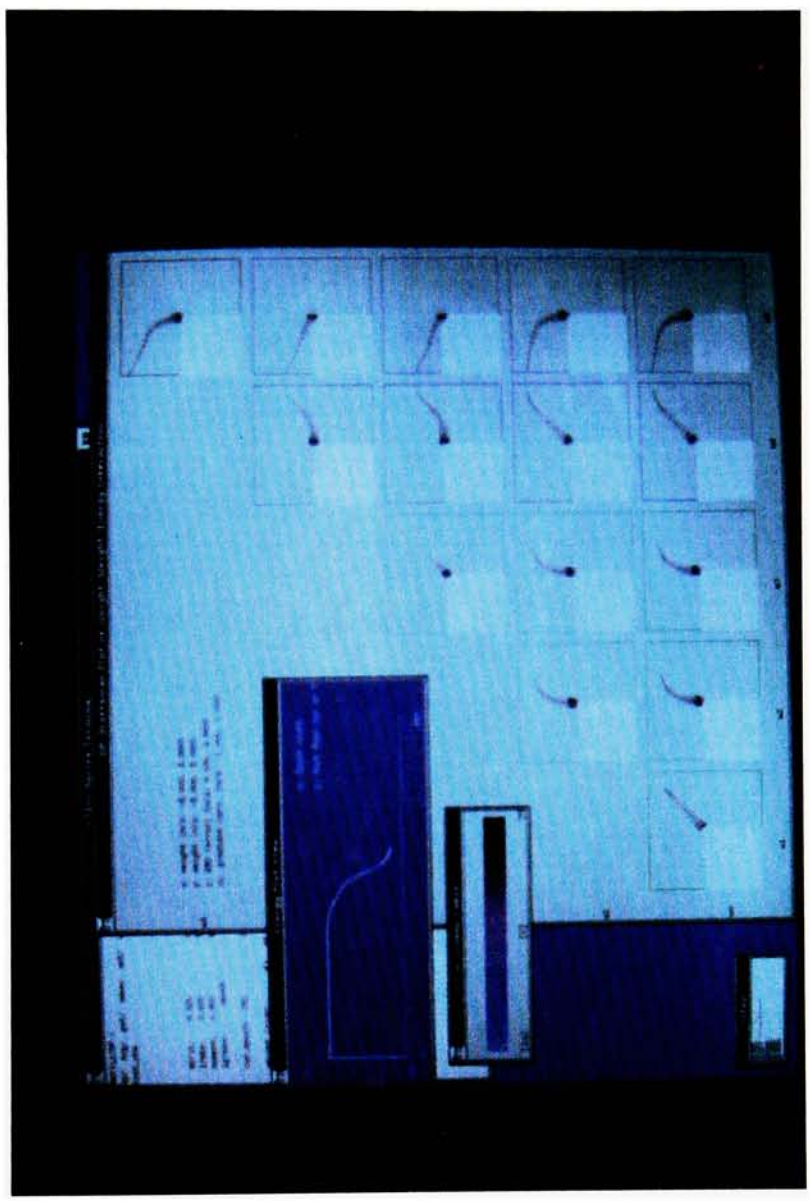


Photo 5-4. Convergence at 268 epochs.

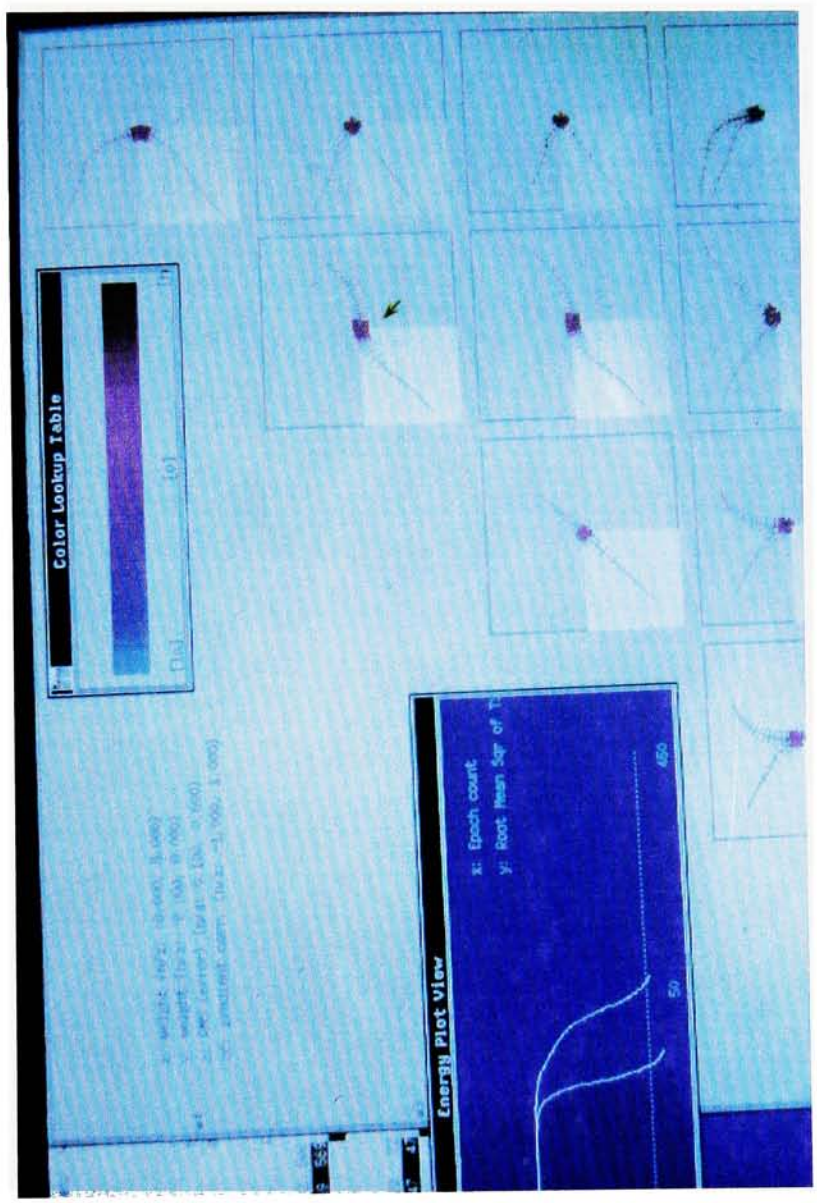


Photo 5-5. The DP and Energy plot of XOR learning traced with a learning rates 0.6 and 1.2.

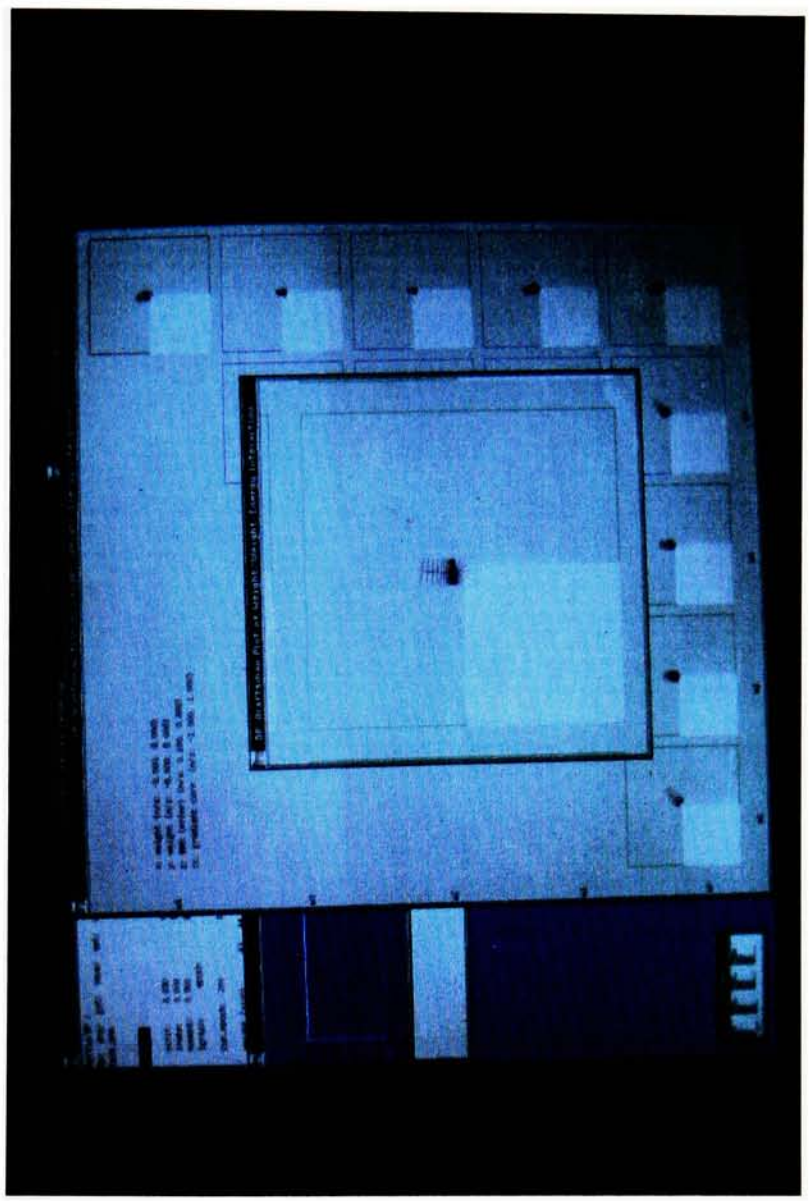


Photo 5-6. Close up view of a Draftsman plot tile (at 200 epochs).

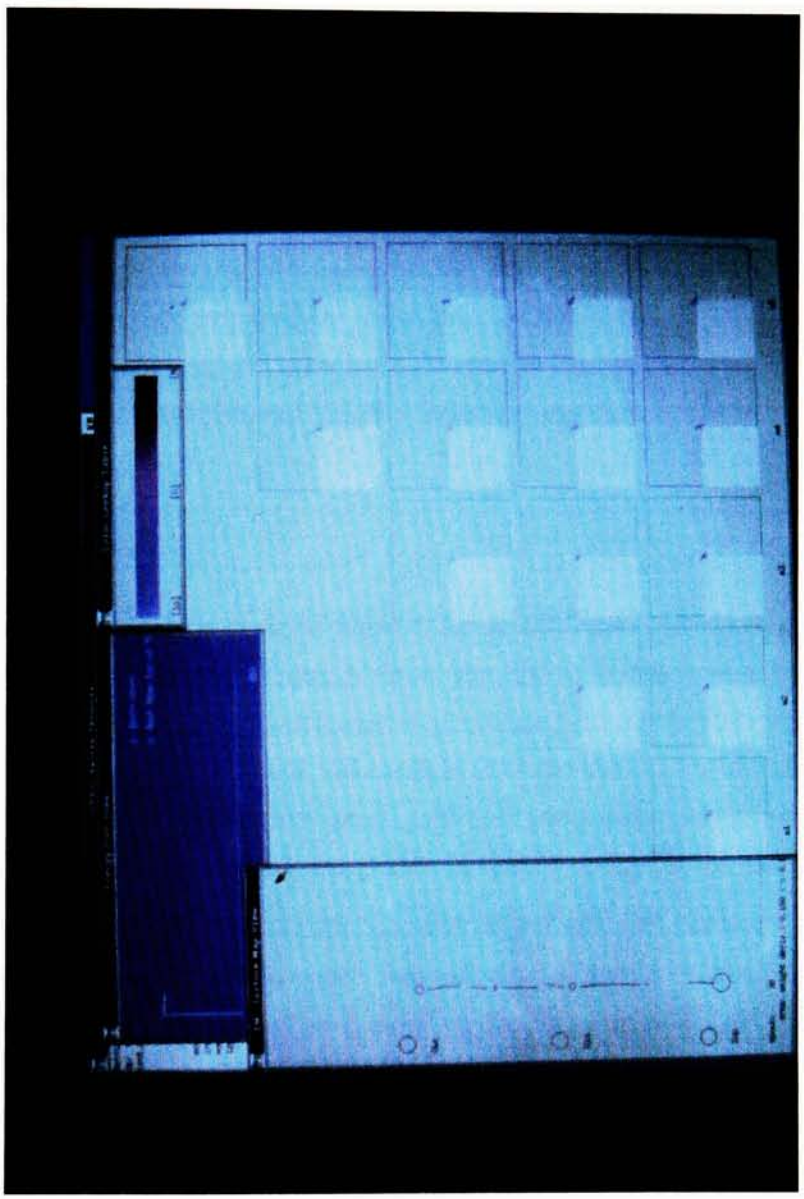


Photo 5-7. Three concurrent animated views.

6

Investigations and Observations

This section reports two types of observations: observations made while conducting a structured set of animated learning investigations, and observations made by invited users of NetViz. Figures accompany the former, though the dynamics implied by them cannot, unfortunately, be represented in this document. The section concludes with the lessons learned from observations and this author's thoughts collected while writing this thesis.

6.1 Investigations

Nine structured investigations were conducted, based on the three network problems and the three animated views described in Section 4.3. Table 7(A) summarizes each of the investigations according to the ANN classifier explored and views used. A more detailed table is in Appendix D which lists the specific purpose, network architecture used, and corresponding view details for each investigation. The overriding purposes of the investigations was to explore:

- general input pattern set feature development / evolution
- behavior descriptive information extracted from a "global" view
- behavior descriptive information extracted from a "local" view
- the compatibility of data structures in each view
- contributions of animation in interpreting network behavior
- contributions of a static view-frame in interpreting network behavior

Many of the investigations explore XOR learning. The reason is that the XOR network has a small architecture suitable for producing simple views (i.e., views with fewer instances of a view component, such as tiles, symbols, etc.) that are

easier for the viewer to learn the view properties from initially. In addition, a simpler view can be explored at a higher resolution in the same screen space than a more complex view, thus displaying unique behaviors through view details. Exploring elements unique to more complex views is attained by the remaining investigations, which involve training the symmetry and horizontal/vertical line classifiers. Variations of some investigations were also studied, in general to explore hypotheses derived from initial observations.

Each investigation training session was started using random weight values between ±0.5. These weights were either generated by the neural network system, or read from a weight file when training for replication was required. A momentum constant of 0.9 was used in all investigations. Furthermore, all cases used epoch updating, where weight error derivatives are accumulated over an epoch and then applied towards the delta weight calculation and weight adjustments. View plotting scales were set to render, as best as possible, the entire range of a variable's values during training, while maximizing the available space in which to create the plot. In the remainder of this section, major observations and findings of these and each original investigation are reviewed.

TABLE 7(A) Summary of Investigations

Investigation/ *Network*	*View Descriptions*	*Mapped Variables*
1a. XOR (learn rate: 0.6)	DP — All (3) blocks	weight, RMS, GCOR
b.	TM — Plot, Grow	weight, bias
c.	RF — All tiles, Stage delay: 1 sec.	weight, bias, activ, error, RMS
2a. XOR (Same as 1a.)	DP — All blocks	delta weight, RMS
b.	TM — Movie, Growth	delta weight, activation
3. XOR (Same as 1. except: varied initial weights)	DP — First + third blocks	(Same setup as 1a.)

Investigation/ Network	*View Descriptions*	*Mapped Variables*
4. XOR (Same as 1. except: varied learn rate)	TM — Plot, Growth	(Same setup as 1b.)
5. Symmetry (learn rate: 0.3)	TM — Movie, Growth DP — First block only	WED, bias weight, RMS
6. Symmetry (Same as 5.)	RF — All tiles, No stage delays. TM — Movie, Spikes	weight, bias, activ, error, RMS, weight, activation
7a. Horiz/Vert (16 in/4 hid/2ou learnrate: 0.25)	TM — Plot, Spikes	weight, activation
b.	RF— All tiles, No stage, delays	weight, bias, activation
8. Horiz/Vert (Same as 7 except: 9 hidden units)	RF — All tiles, No stage delays	(Same as 7a)

6.2 Observations

For each investigation, observations begin with an overview of the network and a description of the view setup, highlighting special settings. Following that are two or three tables of descriptive observations. The tables categorize early, middle, and late training phases. Table columns separate observations made for each of the relevant views. This affords the comparison of parallel developments in each view. Each section concludes with a brief discussion of noteworthy observations, categorized by view type.

Many interesting aspects of the animated graphics are not easily or vividly describable. A video tape of a few of the investigations is available for interested readers.

Investigation 1

This investigation served to "introduce" each of the views using a simple network. The network was set up for replicable training: initial weights were read from a file, and training patterns were presented in a sequential order. All

four NetViz view types were explored. Each of the views visualized algorithm variables unique to the particular view. Weights were visualized in all views except the Energy plot. Biases were visualized in the TM and RF views. The RF view also visualized the hidden and output units activations, and the output unit error[1]. The Energy plot, DP, and RF plot modeled RMS error.

For DP and TM views, variable values were captured and modeled once in a combined feed-forward and back-propagation phase. This update policy was extended slightly in the RF view in two ways. First, weight, bias, and RMS symbols were set to be rendered in a smooth, interpolated method (AA taxonomy:*incremental transition*). The second way was a one-second stage delay setting that caused the flow of the forward and backward propagation phases to be made explicit, by rendering in stages the modeled variables in the phase in which they were calculated. The goal of this setting was to study the benefits and effectiveness of a view with greater *direct content*.

Early Learning (0 - 50 epoch)

DP View —	TM View —	RF View —	Energy View —
In each DP tile, pulse segments were plotted at different orientations, within a very narrow area in each tile's center. Pulses in the two "end" tiles (w0/w1 and w2/w3; see Photo 6-1) in the first block appeared to twist significantly less than those in the inner four tile-plots. Occasionally pulses would also be rendered in contrasting colors (i.e., colors at opposing ends of the prevailing color map).	In the TM plot, arms and legs of the output symbol and first hidden symbols, were observed swinging very slightly, alternating between each symbol's vertical axis over several epoch columns.	Under careful observation (in order follow symbols rendered that were effectively "flashed"), only the output error-frame was seen to alternate between a predominantly 1-pixel and 2-pixel line width. Activation-frames for the first hidden cell (containing w0 & w1) and output cell maintained a 1-pixel line width ; the second hidden cell displayed a long-dash frame.	Over the first 20 epochs, the energy plot formed small (relative to the plot scale) bumps in its profile. Thereafter, it maintained a flat course around 0.5 RMS.

1 The evolution of activation and error values of only the the last pattern in the training set were visualized, since an "every n epochs" modeling/rendering policy was set (n is the view's "frame rate" value).

Middle Learning (51 - 175 epochs)

DP View —	TM View —	RF View —	Energy View —
During most of this period, the "twisting" of pulse lines developed into circular blobs in each of the inner four tiles' centers. The end-tiles, in contrast, maintained a relatively constant angle, perpendicular to the main positive sloping diagonal of the tile. Near the end of this learning period, the beginnings of directed weight-weight traces appeared in all tiles except (w2/w3).	Starting at 131 epochs the first hidden weight was barely beginning some transformation. The arm (outgoing weight) of the first hidden symbol began a positive angle shift accompanied by an increasing positive swing of the symbol's two legs. The second hidden symbol started shifting its arms and legs later, around 166 epochs.	At 82 epochs the first hidden cell activation-frame increased to a 2-pixel width. At 137 epoch it increased again, accompanied by a very slight increase in the cell's receptive-field (rf) weight squares and projective field (pf) weight square. Starting at 158 epochs, the first hidden cell's activation-frame and rf & pf weight square begin significant growth. Also towards the period's end, the RMS-color tile changed from red to magenta (a slight value decrease)	Over the first 170 epochs the energy plot curve was flat, depicting a constant error RMS around 0.5.

Late Learning (176 - 272:(convergence))

DP View —	TM View —	RF View —	Energy View —
This final segment of training displayed a rapid evolution of directed DP weight-weight traces. The end tiles evolved straight weight-weight traces along their tile's main diagonal; slightly curved traces evolved in the inner tiles. Pulses within the middle of the traces were mostly uniform, though some irregularity was detectable upon closer examination. The pulse widths narrowed sharply between 260 and convergence.	The first hidden symbol cell continued is strong positive twisting of legs, stopping at 200 epochs. Around 190 epochs, the bias of the same symbol began a slight decrease. Over this period, the second hidden symbol's legs continued a positive twist, stopping around 210 epochs. From around 185 epochs, this symbol's bias began markedly decreasing, while its outgoing weight began a sharp twist in the negative direction. Photo 6-2 is a monitor screen photograph of the final view.	Overall changes in weight and bias-square sizes were most noticeable during this period. Large jumps in size were noticed in the first hidden cell's rf & pf weight-squares. Smaller size increases were observed in the second hidden tile's rf-weight squares, concurrent with a more rapid size decrease in it's pf-weight. By period's end, all of the first hidden cell's weight squares were large; the second hidden cell's pf-weight squares were very small in contrast. The output error-frame decreased rapidly over the last 40 epochs, from a 1-pixel solid line to a becoming dotted line style. The activation-frames of the first hidden unit and of the output unit continued to thicken all the while. Early in the period, the RMS-color tile steadily changed from magenta to light blue. Over the last 20 epochs the color changed suddenly to a deep blue (smallest mapped color). Photo 6-3 is a monitor screen photograph of the final view.	The RMS error curve began descending around 185 epochs, with a slope of approximately -1.0. Near the error criterion, the curve turned horizontal, and continued asymptotically for 10 or so epochs.

DP — The DP view's pronounced twisting evidenced in the inner four tile-plots corresponds to more disparate initial weight adjustments between weights leaving each input unit (tiles: w0/w2 and w1/w3) than of the weights entering each hidden unit (tiles: w0/w1 and w2/w3). This level of detail was not observed through any of the other plots.

DP tiles traces mainly represented the traversal of the network error point through the uneven weight hyperspace; "unevenness" was depicted by

oscillations in orientation and length, and fluctuations in inter-pulse spacing between adjacent pulses. End-points of the traces define the location of a global minimum in hyperspace. The tile traces also portray the relative "learning rate" of the two hidden units. According to the inner four tiles shown in Photo 6-1, the weight pair w2 and w3 (in-going to the second hidden unit) developed after the weight pair w0 and w1 (in-going to the first hidden unit).

RF — The stage-delay setting of the RF view resulted in the "flashing" of activation and error frames once per epoch. Assessing the evolution of these variables turned out to be difficult. The one-second delay setting eventually severely degraded the continuity of the desired animation, in spite of the setting for frequent RF frame updates. The concept of modeling explicit algorithm flow has merit though. If the delay could be set in fractions of a second, then the continuity disruption might be reduced; a better solution, however, lies in an alternative design.

The incremental updating did not provide any noticeable benefit for the weight and bias symbols. For a significant part of the learning period, these values are already changing in very small increments. Furthermore, for optimal animation, the view's render cycle must be kept small; therefore, the situation where frame updates would involve large, accumulated weight changes is unlikely.

Incremental updating did benefit the RMS-color tile, by depicting the direction of color change, that could be matched to the color bar. Overall, the mapping of color to RMS error provided a depiction of convergence within the primary field of view (i.e., near the other symbols being watched). However, it did not function as integrally as the rest of the RF symbols. Furthermore, it was most effective with the color bar (reference) placed adjacent to the color tile.

All — During early learning, the energy plot, DP, and TM views depicted oscillations resulting from the "smoothing" of the initial, random weight values. All visualizations during the final learning period conveyed the large-scale weight changes of the network. Only the DP and energy plot views truly conveyed convergence, through RMS values. The energy plot view portrayed convergence most clearly. The diminishing error-frame in the RF view also portrayed convergence, though without observing the error caused by the other, unseen training patterns, a false inference of convergence is possible. In the DP

plot, pulse length represented RMS, so as the system converged, weight-weight traces became tapered.

It was common as an observer to alternate between the two concurrently evolving views (e.g., energy plot and DP views). Each view provided a unique perspective of algorithm behavior. Following the energy plot while the weight-weight traces appeared unchanging provided the desired feedback. The pulses on the weight-weight traces provided a compelling synthetic image of a local contour, both by inter-spacing and in width changes near convergence.

Investigation 2

This investigation explored the evolution of delta weights during the training of the XOR network used in investigation 1. Delta weights are the absolute *change* applied to weights, and therefore, provide a temporally local perspective of the evolution of weights.

One of two views used was a TM movie, a variation of the TM plot in which each new epoch column is rendered over its predecessor epoch column. The result is a compelling illusion of freely moving arms and legs (refer to Section 4.3.2: *Texture Map* for further view details). The second view was a DP view using a configuration similar to investigation 1. Since delta weights are the small increments applied to adjust weights, the plotting scales of both views were set to plot the range of -0.2 and 0.2 (2.5% of the scale used for weight-weight mapping). This range was the best compromise for undistorted TM symbols and unclipped DP tiles, and adequate detail in the views. Finally, a frequent TM render cycle was used to effect a reasonably pleasing animation speed, without unduly slowing the algorithm.

Early Learning (0 - 50 epoch)

DP View —	TM View —
The first block of tiles developed only a red dot in the center of each tile. In the tiles of the second and third block, a significantly large, overlapping "zig-zag" red line formed, over the first 20 epochs. Thereafter, no activity was perceptible.	Legs of the output symbol cell and arms of the first hidden symbol cell exhibited slight swinging about the vertical axis. This swinging was mimicked by the swinging hidden symbol cell arms. This activity was dampened by epoch 20.

Middle Learning (51 - 170 epoch)

DP View —	TM View —
Around 145 epochs (late in this period) a blue trace began forming in the lower end-tile and the inner four tiles of the first block. Straight traces also began forming in tiles in blocks 2 and 3.	All legs and arms of the first hidden symbol cell began gradually adjusting in the positive direction. Corresponding arms, closest to the base, of input symbol cells and the inner leg of the output symbol began a gradual adjustment in the positive direction. The second hidden symbol cell did not exhibit any perceptible change.

Late Learning (171 - 317 (convergence))

DP View —	TM View —
The start of this period continued with the growth of traces started at the end of middle learning. Around 180 epochs a directed trace began in the upper end-tile of the first tile block. A that point, the trace in the inner four blocks began turning right. Up to epoch 210, the upper end-tile trace evolved as a line with slope +1.0. Concurrently (starting at 171 epochs) a directed trace evolved in the tile in block 3 (w4/w5 - the two weights connecting the output unit to hidden units). Around 210 epochs, most traces evolved the first of small cusps and bends (the two end-tiles in block 1 were exceptions) in the d/weight-d/weight traces. The greatest activity was in w5. Around 260 epochs, all tiles developed traces directed towards the center of each tile. These trace segments became increasingly blue as they neared their destination.	Around 190 epochs, the base diameter of the first hidden symbol began expanding. By the end of the period, its diameter was nearly the size of the reference base in the left margin. This was the only symbol base that grew during learning. The legs in the first hidden symbol continued twisting strongly in the positive direction, stopping at 200 epochs. Starting at 200 epochs, the second hidden symbol's legs began twisting more noticeably in the positive direction, and its arm began a sharp twist in the negative direction. In general, the symbols evolved into shapes closely matching those observed in investigation 1. Around 210 epochs, the "tensed" arms and legs of the first hidden symbol began gradually unfolding. During that period, the arm of the second symbol continued an aggressive positive twist. By the 255 epoch, it began a rapid unfolding, following the first hidden symbol and ending in a nearly vertical position.

DP — The final DP view of the delta weight interactions is shown in Photo 6-4. In this DP view, cusps and bends represent a significant weight-rate change[1] between the weights paired in each tile; a "correction" of the weight evolution between weight-pairs. For example, the first bend in the w4/w5 tile's trace portrays the shift of larger weight changes from w4 to w5. Similarly, in the w1/w2 tile's trace, the top cusp (top of the "B") indicates a "slowing down," of, first, w1 changes, then w2 changes, followed many epochs later by larger changes in w2. Traces start out in red, representative of high RMS error, and gradually become dark blue as network error decreases. Traces in the first block of tiles best illustrate that the end of a trace concludes near its start; more accurately, traces conclude near the tile origin (0,0), since initial weights (which describe the start of the trace) are approximately zero.

A playback of the display list (see Section 5.2.2: Windowing Interface) of this view portrayed a compelling delayed acceleration of the traces' formations, their "signature," and a final deceleration of the trace to its starting point.

TM — The TM movie was overall more engaging than the DP view, particularly when there was no noticeable activity in the DP view. Like the RF view in investigation 1, the TM movie depicts in a compelling way the first hints of change of a variable. This impression is more marked in the TM movie, because smaller changes in angles were more noticeable than square-size changes. A significant shortcoming that the RF view also exhibited (in Investigation 1), was the slow animation speed. When the TM view's display list was replayed (an attempt to observe the movie at a speed closer to original expectations), the global staggered evolution between weights to and from hidden unit 1 and hidden unit 2 was more noticeable. Because this form of animation is too fast, several consecutive replays were necessary.

The evolution of arms and legs in the TM view that closely matched the corresponding symbols modeling weights (in investigation 1), illustrated the property that weights are the accumulation of delta weights. The final TM view of the delta weight interactions is shown in Photo 6-5.

---

1 The weight-rate is the magnitude of the delta weight per epoch.

All — The narrow plotting scale required by the delta weights accentuated the visualization of initial oscillations associated with the output unit (to which weights 4 and 5 lead into). Such initial oscillations are characteristic of the back-propagation behavior. The largest weight change is made to weights directly associated with the source of the largest error signal, namely the output unit.

Investigation 3

The purpose of this investigation was to explore the XOR solution weight-space. A script was used that reinitialized the network with new random values and then invoked a sequential training, a total of 14 trials. An upper limit of 450 training epochs was set to force the start of a new trial in the event of a trial reaching a local minimum.

Network learning behavior was the same, overall, as described for the DP view in investigation 1. The final, composite view is shown in Photo 6-6. Of the total training trials one reached a local minimum, and another did not break from its starting point. The remaining traces reached global minima, mostly within 160 and 250 training epochs; a few trials required more than 250 epochs.

The training trials were animated at a low render cycle since observing network learning details was not an objective. Evident across the 12 final views of successfully trained networks are two classes of solutions[1]. They are best defined by characteristics in the end tiles: the first class generally has parallel traces of slope -1.0, and the second class generally has parallel traces of slope +1.0. The views defined by these classes suggest two pairs (4) of global minima. Supporting this inference is the final view depicting training to a local minimum shown in the Energy plot. At the end of 450 epochs, and still at 0.5 RMS, the trace in the upper end-tile approximates end-tiles of views in the first solution class, while the lower end-tile approximates end-tiles of views in the second solution class.

From this investigation it appears that the class that a final view matches is independent of the number of training epochs. Noteworthy are the variations in size and style of traces in final views that reach the same global minimum. These

1 This finding was more obvious when studying each trial plotted separately.

variations most likely arise solely from the differences in initial (random) starting weights, since these comparative training trials were each run identically apart from their initial weights.

Investigation 4

Like investigation 3, this investigation sought to create a composite image of learning using different learning rate constants. The main view was the TM plot view, modeling weights in the growth style of legs and arms, and biases as symbol bases. A view render cycle of 20 (resulting in very infrequent view updates) was set to allow up to 600 epochs to be plotted in one row, within the window frame. An energy plot view was also used to follow the RMS error reaching the convergence criterion.

A secondary purpose of this investigation was to further explore the utility of NetViz scripts. In this case, the script was comprised of four repeats of commands to: perform a sequential training, reset the view to start a new row (in TM view), and reload same initial weights. The command to change the learning rate was invoked before invoking the *seq_train* command. The learning schedule started with a learning rate of 0.4 for the first row, 0.8 for the second, 1.2 for the third, and 1.8 for the fourth row. A command that turned off labeling[1] was set in the file to execute before the second training session. For each trial, the only difference in training was the learning rate.

The final views are shown in Photos 6-7 and 6-8. An approximate linear relationship between learning rate and number of epochs to convergence is suggested by global view of the plot. Fewer training epochs are required as the learning rate is increased. A local view of each plot reveals two things. First, one common network solution is arrived at with the two lower learning rates (0.4 and 0.8), determined by the matching shapes of the final epoch columns, and a different common solution is arrived at with the higher learning rates (1.2 and 1.8). The second revelation, involves a comparison of the number of epochs from

1 The epoch count, the current plotting scales, and the horizontal dotted line, all of which are normally rendered at the bottom of the window are not rendered for each new row started when labeling is "off."

which marked changes in at least one symbol begin to convergence. According to the composite view, at a learning rate of:

- 0.4, large weight evolutions take approximately: 440 epochs
- 0.8, large weight evolutions take approximately: 120 epochs
- 1.2, large weight evolutions take approximately: 80 epochs, and
- 1.8, large weight evolutions take approximately: 80 epochs

This gain in convergence at the higher learning rates involves a mechanism that is not portrayable in the TM plot. The mechanism may either be general to learning processes above a certain learning rate (possibly 1.0), or different mechanisms may be involved at each each learning rate. Further exploration is warranted.

Investigation 5

The fifth investigation was one of two that explored learning with the symmetry classifier (see Section 4.2.2:*Symmetry Classifier* for a network description), an ANN of "medium" complexity among the set of ANNs selected for this project. Three concurrent views were used in a way previously not explored, and each view modeled a unique set of variables. A DP view modeled weights as traces and the RMS error as color. A TM movie view modeled WEDs as legs and arms in a growth configuration. Finally, an energy plot modeled the TSS error over epoch number.

The rationale for view selection and the variable-to-view assignments was based in part on displaying information of interest, and specifically to:

- see through a DP view the "suggested locations" of global minima
- see how WED values indicate where weight adjustment activity is high and low
- see if a TM movie view can accentuate the activity of WEDs in a relatively narrow viewport, thereby displaying a sense of network activity not possible in the DP view with optimal scaling.

The last rationale refers to the DP view's practical tile-size limitation in this investigation, caused by fitting 66 tiles[1] (in the first block alone) within a 30 cm^2 viewport. For the same reason, RMS error was mapped to color changes on a 2 pixel line, instead of pulses as used in investigation 1.

The pattern set used for training included all eight symmetrical patterns, and seventeen asymmetrical patterns. Because of the large training set size and no objectives for ensuring learning reproducibility, training patterns were presented in random order for each epoch. Render cycles for each view were set such that the TM movie would be updated frequently; three times more within the DP view update period and five times within the energy plot update period.

Early Learning (epoch 0 - 30)

DP View —	TM View —	Energy Plot View —
Around 25 epochs, significant parts of tile traces emerged; tiles that had w6 as an x-axis variable, showed the longest initial traces. The tiles adjacent to w6 developed the next longest traces, and the remainder have even smaller ones.	Initially, straight arms of the input symbol cells swung widely at the base, in apparent unison with each other; the corresponding legs in the two hidden symbols (six legs per symbol) moved similarly. At 20 epochs the same arms began to bend (each input symbol has two arms, one to each hidden symbol cell). Soon after, the swinging changed into a pattern that separated the bottom 3 hidden symbols from the top 3. One set of WED arms moved/flexed significantly, while the other set remained "relaxed" (near vertical); the reverse then occurred the next epoch cycle. Over time, the end-most segments of legs in the hidden symbols folded while the inner segments remained straight. Near this period's end, the arms of the hidden symbol cells were bending/swinging dramatically.	The plot started out with a bumpy profile that gradually tapered off to a smoother flat line at around 30 TSS error.

1 The number of tiles for the first block is the permutation, P(n,2)/2 = n(n-1)/2, where n= the number of input-hidden weights. The permutation is halved, because the mirror-image of the tiles along the diagonal are not plotted.

Late Learning (epoch 31 - 77 (convergence))

DP View —	TM View —	Energy Plot View —
At 46 epochs, a "burst" of weight-weight trace growth occurred in all tiles with weights w8, w9, w2 and w3. By 60 epochs, the traces appeared to have reached the global minimum.	The asynchronous behavior from the previous phase continued at a steady pace. The hidden symbols' legs were becoming significantly distorted. Towards network convergence, the hidden symbol legs unwound into their resting orientation (straight vertical). Easier to follow was the rapid expansion of the output symbol's base, and simultaneous contraction of the 2 hidden symbol bases.	At around 50 epochs, the plot's flat curve broke into a sharp / steep decent. At 65 epochs, the TSS curve reached near the 0.0 TSS baseline, leveled off, and then continued an asymptotic course parallel to the x-axis for another 12 epoch cycles.

TM — The TM movie provided greater viewer engagement; however, no general meaning could be made of the arm and leg distortions, apart from identifying those units receiving the greatest adjustments, and the "direction" of those adjustments. Although the view render cycle was set for frequent updating, the animation appeared very jerky. Furthermore, the longer legs of each hidden symbol cell eventually became too distorted. Relative to weights and delta weights, the WED values appear to be less continuous over three or more epochs.

DP — The final DP view shown in Photo 6-9 has a salient global symmetry among the tile-traces. Two major mirror axes are noted: one between w8 and w9, and another between w2 and w3. Figure 6-1 illustrates the noted symmetry to the network topological view. These four weights are on connections between hidden units and the two input units that surround the network's conceptual mirror axis (as defined by the input patterns). The final DP view suggests that the network's solution involves an internalization of symmetry. However, exactly how this is used in classifying input stimuli, is not determinable in the DP view.

All — Finally, a surprise of this investigation was the consistently quick learning time, relative to the architecturally simpler XOR network. Training trial replicates involving new random weights and random training, converged within a mean of 80 epochs (6 trials). The brevity in training was evident in the energy plot profile which had a very small "flat" period prior to the decent

period. This suggests that there are many points in this weight hyperspace, close to the "origin" that lead to a descending path.

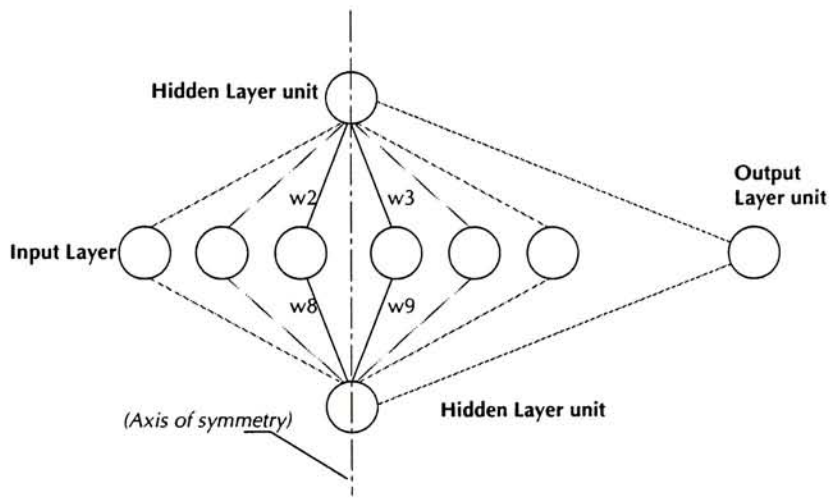


Figure 6-1. Symmetry network topological view.

Investigation 6

Symmetry network learning was again investigated using two different views to visualize behaviors that were unattained through the views used in investigation 5. Two movie type views were used. The first was an RF view modeling weight and bias as squares, activation as a blue frame, and output error as a red frame. A stage delay of zero was set, so that the activation and error mappings would not be erased between forward and backward propagation phases, and that faster animation would be perceived. The second view was a TM movie modeling weight through spikes and activation through each symbol base. Spikes were chosen to investigate their informativeness, relative to the growth style, with a large number of weights. Only leg-spikes were modeled to reduce the information content of the view.

Network training was conducted the same way as done in the previous investigation. Render cycles were set so that both views updated frequently, since both were animations showing current images only.

Early Learning (epochs: 0 - 30)

RF View —	TM View —
Initially, weight squares changed in very small increments over both hidden and output fields. The activation frames of hidden tiles started out with a dotted line style and maintained this through out this phase. The output error-frame started out with a 5-pixel thick frame. By epoch 30 it had decreased slightly. By the end of this period, rf-weight squares within each hidden tile began changing asymmetrically. Changes were largest in the second tile: the group of three weight-squares on the left grew smaller, while the three on the right grew steadily larger. Concurrently, the same unit's pf-weight square began to contract. By 30 epochs, the color tile changed from purple to pale blue (a "decreased" along the color scale).	The bases of both hidden symbols began as dots (very small circles). In the beginning, only the second hidden symbol's spikes showed any activity (i.e., slight changes in length). At 30 epochs, the same symbol's left group of 3 spikes began shortening, while the right 3 spikes began lengthening. The output symbol base diameter fluctuated throughout this period. Near 15 epochs the symbol's spikes gradually began getting shorter, symmetrically at first. By the end of this period, the right spike was noticeably shorter than the left one.

Late Learning (epochs: 31 - 80 (convergence))

RF View —	TM View —
At 55 epochs the output tile's activation-frame thickened from a long-dash to a 3-pixel width. Within 10 epochs thereafter, the error-frame thinned to a 1-pixel width. The second hidden tile's weight-squares (rf and pf) continued to receive the largest adjustments (the first hidden tile's weights were also being adjusted). Globally, the hidden tile weights continued to follow the earlier adjustment trend. At 60 epochs, the second hidden tile's rf-weight squares appeared to reach an equilibrium, at which point, the first hidden tile's weight squares began more noticeable adjustments. Also at 60 epochs, the output tile's activation frame reached its maximum line thickness; the error-frame thinned to a long-dashed line style. Beyond 65 epochs, few symbols (i.e., weight-squares, frames, etc.) changed significantly. By convergence, each hidden unit developed a uniquely symmetrical weight pattern. This is shown in Photo 6-10.	The output symbol's base began a steady growth. It's spikes continued to of uneven length, as the right spike continued to shorten. The second hidden symbol's spikes continued to follow the earlier trend, with the right group of 3 spikes lengthening considerably. Near 60 epochs, the first hidden symbol's spikes began changing length more noticeably; the two spikes bordering a conceptual vertical center axis were being adjusted in opposite directions (the left spike was getting longer, the right was getting shorter). Similar, though smaller, pair-wise adjustments (i.e., across the center axis) were affecting the other spikes. By 70 epochs, the output symbol's left spike had shortened to a length matching the right spike. Over the final epochs to convergence, these spikes continued to slowly decrease in unison.

Concurrently viewing the two movie views to study details of feature evolution was fatiguing. In order to document the dynamic details in the tables above, two replicate training sessions were necessary, one for each view studied. Individually, however, each movie provided a compelling display of algorithm variable evolution.

Each view modeled a common subset of algorithm variables. The TM view was a more engaging and informative dynamic view for this author. Two reasons are proposed. One is that the TM movie has comparatively fewer dynamic components that required attention. The other is a stronger percept provided by the extending and compressing spike lengths, compared with the size changes of the weight squares.

On the other hand, the final RF view provided a stronger image of the final hidden unit feature detectors. In addition, the dynamic feedback when testing the final trained weights was more informative.

Both views failed to satisfactorily convey when weight values were positive or negative. This information was desired more so for this network than for the XOR, because one of the hidden unit feature maps suggested a pattern that was not confirmable until the actual values were read from the control screen.

After training, the TM movie was "replayed" again several times. This high speed animation made clearer the weight adjustments of the first hidden node around 60 epochs. Weights were adjusted pair-wise about the symbol's center vertical axis (symmetry axis also); in one frame-update the left weight of the pair would adjust, followed in the next frame-update, by right weight.

Investigation 7

This investigation was the first of three that explored learning of a horizontal and vertical line classifier network. The first trial used a TM plot view modeling weight through spikes (legs only) and activation through the symbol base. The second trial used an RF view modeling weight and bias through squares and activation through the tile frame width. Both trials also used Energy plot views, modeling TSS error with the TM plot view and RMS error with the RF plot view.

Observation details of the Energy plot are not included since the profiles that formed were basically the same as those described in previous investigations.

A DP view was not considered since 2,016 first-block tiles would be involved, in which traces too small to be meaningful would be rendered.

Early Learning (epochs: 0 - 60)

RF View —	TM View —
Learning started with the activation-frame of hidden tile 2 having a solid 1-pixel width and the other activation-frames having long-short dashes (a lower value than a long-dash). Near 45 epochs, hidden tile 2's activation-frame increased to a 3-pixel line width. Hidden tile 4 also changed from a long-short- to a long-dash (value increase). Both rf- and pf- weight squares also began changing noticeably; the row of four rf-weight squares, second from the bottom, showed the most relative growth. At 55 epochs, hidden tile 2's activation-frame reversed a growth trend, reverting to a long-dash line width. Its rf-weight squares in the second from the bottom row continued growing; its bias-square also grew slightly.	The primary change over the first 20 epochs was the rapid decrease of hidden symbol 4's base. At 34 epochs, the base of hidden symbol 2 expanded slightly, and maintained this size through out the period. At 44 epochs, the first output symbol's spikes showed the start of adjustments; these became more pronounced over the subsequent 10 epochs. At 49 epochs, the same symbol's base began to expand. At 59 epochs the spikes of both output symbols became adjusted such that the analogous spikes of each symbol were essentially of opposite length. Concurrently, the hidden symbol 2 showed a uniquely large and equal growth of spikes 5 through 8 (counting from the left).

Middle Learning (epochs: 61 100)

RF View —	TM View —
By epoch 65, the first hidden tile's frame decreased to a dotted-dash line style, the second hidden tile's activation-frame decreased to a dotted line style (very small value), and the third hidden tile's frame increased to a long-dash. Except for the first hidden tile, the pair of pf-weight squares in the remaining hidden tiles began adjusting such that one weight of the pair grew and the other contracted. By epoch 77, the third and fourth hidden tile's activation-frames decreased to dotted line styles. The second to the bottom row of rf-weight squares in hidden tile 4, became more pronounced (as a row), paralleling the growth of the same row of rf-weight squares in hidden tile 2 (but not as big).	At 64 epochs the top output symbol's base decreased to nearly a dot. Its spikes continued to follow the earlier trend: spikes 3 and 4 continued to lengthen and spike 2 decreased to the point of nearly disappearing; spike 1 remained short. The spikes of the bottom output symbol also continued their earlier trend: spike 2, long at this point, continued to lengthen; spikes 3 and 4, very short at this point, continued to shrink; and spike 1 remained short. By 64 epochs, the second hidden symbol's base diameter increased significantly; its spikes appeared to have equilibrated. At 69 epochs, the bases of hidden symbols 3 and 4 began to shrink rapidly; by epoch 84 they became dots. Also, the first hidden symbol's spikes began forming a new pattern. Magnifying the view showed that spikes 1, 5, 9, and 13 (the first of each group of four) were becoming shorter. By 84 epochs, these spikes equilibrated at approximately half the size of the other adjacent spikes. At 99 epochs, the fourth hidden symbol's spikes 4 and 8 continued to grow. The left half of hidden symbol 3's spikes (8 spikes) increased in length as a group, becoming longer than the other 8, equal-length spikes.

Late Learning (epochs: 101 - 180 (convergence))

RF View —	TM View —
The predominant changes this period were adjustments in activation-frames: by 150 epochs, all hidden tile frames become dotted (the first hidden tile was the last to change). The final frame of this view is Photo 6-11.	No significant changes/evolutions were noted during this period, or otherwise, changes were too small to be rendered. The final frame of this view is Photo 6-13. A magnified portion of this view is depicted in Photo 6-14.

RF — The pattern of weights in a trained network determines the network's classification behavior. Prior to this investigation, it was presumed that those rows or columns in hidden units described by large weights in the receptive field view, functioned as direct matching templates for test patterns presented for classification (feature detection). In this investigation these weights seemed to suggest that:

- hidden unit 1 weakly recognizes both horizontal and vertical lines
- hidden unit 2 predominantly recognizes horizontal patterns
- hidden unit 3 predominantly recognizes vertical input patterns

- hidden unit 4 predominantly recognizes horizontal patterns

A subsequent test of the network with each training pattern indicated the following, instead:

- hidden unit 1 weakly recognizes both horizontal and vertical lines
- hidden unit 2 predominantly recognizes vertical input patterns
- hidden unit 3 predominantly recognizes horizontal patterns
- hidden unit 4 predominantly recognizes horizontal patterns

These hidden unit assignments match the classification suggested by the projective-field view. Considering that the weights in the projective-field view are "closer" to the actual classification layer (i.e., output layer), the match makes sense. Note, however, that it is the projective-field view (from the perspective of the hidden layer) that provides this information and not the receptive-field view of the output layer.

Still intriguing was the observation that hidden unit 3, with its pronounced right-side columns of weights as shown in Photos 6-11 and 6-12, was the predominant detector of *horizontal* patterns (instead of a more distributed detection). In other words, the weights of this unit suggest that the distinguishing aspect of horizontal patterns, relative to a vertical ones, was the presence of two horizontally adjacent "on" units, on the right side. Clearly, this outcome is an artifact of a noiseless training set, and possibly due to the symmetry of the training set (i.e., 4 horizontal patterns and 4 analogous vertical patterns).

TM — The local patterns produced in the TM view are very compelling; the use of two output units emphasizes the clustering of the output weights to favored feature detectors. Since a symbol changes gradually across epochs, and each symbol tends to be more distinct (by way of its spikes and base character) within an epoch column, a stronger grouping is perceived by rows than by columns.

Investigation 8

This final investigation explored a variation of the horizontal and vertical line classifier studied in the previous investigation. This network used nine hidden units instead of four. The purpose was to explore the similarities and differences in the overall training, and in particular, the types of emergent receptive-field feature detectors.

An RF view and an energy plot view were used, both set up the same way as in the previous investigation. Different starting weights and a different, random training pattern presentation were used.

At the start of training ,the hidden tile activation-frames were rendered in a variety of line styles and thicknesses. The hidden tile weight and bias squares were approximately the same size (though, values in the data-form showed that some were "just" below zero and others "just" above zero). Within 21 epochs (7 frame updates, in this case), many hidden tiles already clearly exhibited strong weight-square development trends, in particular the emphasized weight rows of the receptive-field hidden tiles 8 and 9, and emphasized weight columns of other receptive-field hidden tiles. By epoch 40, the projective-field hidden tile weight squares began noticeably changing. Concurrently, the activation-frames of hidden tiles 3 and 5 developed thin solid line widths, and 7 and 9 thicker solid line widths. The remaining hidden tile activation-frames were of various dashed line styles. By epoch 90, the left two weight squares of the projective-field hidden tiles 5, 7, 8, and 9 became very large, while their right weight squares had become very small. In contrast, corresponding weight squares in hidden tiles 3 and 4 changed very little during the course of training. The final outcome is shown in Photos 6-15 and 6-16. Note in particular that none of the hidden tiles developed a mix of highly emphasized horizontal and vertical weight squares in the receptive-field views. Two hidden tiles (marked: "#"), however, developed (barely noticeable from the view, but clear from values in the control screen) one slightly emphasized row together with one sightly emphasized column.

Network learning with this network architecture took a significantly shorter time than the one with four hidden units. This behavior is generally expected, because the additional hidden units provide more hyperplanes that can participate in the division of the classification hyperspace.

By definition, the network architecture used here was superfluous, because the number of hidden units exceeded the number of training patterns (8). In other words, each hidden unit could theoretically "learn" to recognize one of the patterns, leaving one hidden unit unused. In most cases, distributed feature detectors formed. Occasionally, the network studied here produced feature detectors, some of which recognized 2 or 3 input patterns alone, and the

remaining input patterns recognized by a combination of detectors. In those cases, several hidden units are unused.

Overall, this variation of the original network architecture did not exhibit any extraordinary properties. Through the RF view and energy plot view, the expected characteristics of faster learning and the potential of greater feature distribution were exposed.

6.3 Observations of Investigation Variants

Two separate variations of investigation 1 involved exploring learning with high learning rates, using a DP and an RF view. In the case of the DP view, a learning rate of 1.2 (double that of the original investigation), the same initial weights, and sequential training were used. Photo 6-17 shows a combined view of traces formed at a learning rate of 0.6 and at 1.2. During learning the trace in tile w2/w3 developed in the opposite direction of the trace formed at a learning rate of 0.6. Globally, the view matches one of the solution classes produced in investigation 3. Locally, traces in all tiles were more sharply tapered, with larger inter-pulse gaps, corresponding to a steeper convergence profile. The traces were also more noticeably jagged, suggesting that at the higher learning rate, the error point has greater kinetic energy as it traverses the error surface and therefore requires slightly greater weight adjustments.

The same network was trained and studied through a simple RF view (with no stage delay). Around 160 epochs, weights, biases and activations began changing in large increments. The emerging view pointed to essentially the same solution arrived at under equivalent training conditions of investigation 4 (third row plot). However, in the RF view, the size differences of weight-squares and bias-squares relative to the equivalent TM growth plot, made a greater impact on this author of the condition of the network. Foremost were the large, positive biases. After studying the activations produced through a testing of all training patterns for recall, the following was inferred:

- a large bias suppresses a unit's activation when a strong excitory signal is input, and
- a large bias spontaneously activates the unit, when incoming signals are weak

Finally, a variation was conducted of the multi-row TM plot in investigation 4. Instead of varying the learning rate, the momentum coefficient was varied, and the learning rate was fixed at 1.2. The final TM plot view is shown in Photo 6-18 and the corresponding energy plot in Photo 6-19. The momentum coefficient schedule started at 0.1 for the first row, 0.4 for the second, 0.7 for the third, and 1.0 for the fourth row. The first three rows of the TM plot indicate that successively larger momentum coefficients can accelerate the learning process. Studying the corresponding energy profiles, momentum coefficients appear to affect the time before which the RMS error decent begins. The effect of the momentum coefficient as been described as a filter of high-frequency variations in the error surface [McCle88]. However, according to the fourth row of the TM plot, and the "jagged" RMS error profile, this generalization fails at some point. A momentum coefficient that is "too large" (1.0 or more) can apparently lead to oscillation to a local minimum.

6.4 Observations by Other Viewers

On two separate occasions, NetViz was introduced to students and to associates of this author, each having a working familiarity with principles of ANNs. The author provided a detailed overview of the three views, using the investigations as examples. Each session was conducted informally, allowing the participants to freely express their opinion about the system, the animated views, and the general concept of algorithm animation of ANN learning.

Most of the participants could follow a training session modeled by an Energy plot and a DP view. The TM plot and movie seemed to be more difficult to follow, especially when both arms and legs were displayed in the growth symbol style. The RF view, with a stage delay of one second was confusing to participants, but with a stage delay of zero, the view displayed more recognizable and informative forms.

The dynamic aspect of the visualization was generally considered to be constructive. An example given, that was not previously considered by this author, involved cases of continuous network training, where new training patterns are presented some period after the network is partially trained. For example, given an ANN for recognizing certain hand printed characters, a "new"

training set might be a standard set of characters generated by a different person. Using a TM plot as an example, the time at which a new pattern set is introduced might appear as an epoch column of deviant symbols. The viewer could then follow the subsequent weight adjustments, and note the type of deviations from an earlier trend. Although a deviation could also be detected in an Energy type plot, the TM plot would be more information rich (e.g., which feature detectors were affected the most would be depicted).

Another category of comments/recommendations, frequently made after several demonstrations with the system, were provisions for direct manipulation. One suggestion was to display the values of variables mapped to a TM symbol within a small window near the symbol "touched" with the mouse cursor/mouse button click.

A suggestion was made to provide the option in views (the Energy plot was mentioned predominantly) to visualize separately and in parallel, the evolution of output unit error(s) and activation(s) of selected training pattern pairs. (The prototype can currently animate the evolution of output error and activation of only one pattern at a time, using the RF view). The predicted use of such a view would be to find patterns in the set that were "difficult" to learn by the network. (The suggestion was made based on the current ANN research activities of one of the participants.)

Finally, the questions asked by participants suggested that each was familiar with "seeing" a different combination of algorithm variables from their own work. The TM and RF views did not map well to the mental images participants already had. When participants were explained about the basis of "good form" in the view designs, they felt that more important was the capability to visualize any algorithm variable desired, in conjunction with algorithm operations. It was also felt that after enough use of the NetViz views, useful information about a classifier under study could be gained.

6.5 Findings — Learning Related

Table 7(C) summarizes the major behaviors depicted most prominently by the three view types used in the set of investigations. Many of the behaviors listed were predicted during view design (Table 4(A)).

TABLE 7(C)	Summary of Findings	
Property	*Views that portray properties best*	*Contributing Visual Elements*
Learning convergence	Energy plot Draftsman plot (DP)	• Profile decent to near zero • RMS error modeled by pulses
Feature Extraction	Receptive Field view (RF) Texture Map plot (TM)	• Weight & bias squares in final frame of the converged network; in activation-frames during a test pattern classification. • Leg-spikes
• Presence of (multiple) global minima	Draftsman plot	• Using RMS error modeled by color or pulse: termination of weight-weight traces in tiles with tapered, widely spaced pulses or color changes towards the low end of the color map. Multiple overlaid plots of weight-weight traces depict multiple global minima.
Property	*Views that portray properties best*	*Contributing Visual Elements*
• Presence of local minima	Draftsman plot	• Using RMS error modeled by color or pulse: very slow growth of weight-weight traces with a constant width, bunched pulses or maintained trace colors from the upper middle half of the color map
• Characteristics of the error surface in weight-space	Draftsman plot	• Curves and jags in weight-weight traces in tiles; using RMS error pulse: changing fluctuating pulse widths and inter-pulse spacing
• Oscillatory behavior at the start of learning	Energy plot Draftsman plot Texture Map (plot and movie)	• Initial "bumps" at the start of the curve • Using pulses to model RMS error: the multiple, localized orientations of pulses; using color to model GCOR: trace rendering in contrasting colors. Also: attenuated delta weight interaction traces. • Wagging of output symbol legs or hidden symbol arms.
• Multiple learning solutions/ representations	Draftsman plot Texture Map plots	Generally: multiple learning trials, with different weights,or different learning rates: • Multiple (overlaid) weight-weight traces • Multi-row, comparison plots

According to Table 7(C), different NetViz views and different algorithm variable combinations expose different learning algorithm behaviors. It seems at first that

a display that shows all possible information is the ideal. However, beyond a certain point, high information content views become difficult to follow.

Table 7(C) excludes several additional interesting inferences made in the observations. In summary:

- Learning always begins with relatively large weight corrections that are projected as oscillations. The algorithm in other words, begins with a short phase of smoothing the *a priori* assigned random weights.
- The ANNs investigated exhibited a "focused" weight adjustment, in which certain weights are aggressively adjusted during early learning periods, and weights relatively dormant over that period are aggressively adjusted through the later learning periods. This behavior suggests that the convex classification boundaries, described by weight hyperplanes, are adjusted in sections (one or two hyperplanes at a time). Initially, coarse class boundaries are adjusted, followed by finer class boundary adjustments.
- Observations in investigation 3, suggest that although the initial random starting weights are at a level of "noise" relative to the final weight values, differences are amplified over the course of learning.
- Certain variable mappings produced more informative views than otherwise. For example, modeling pulse width to RMS error in a DP plot, provided more information about learning than when modeling GCOR to pulse width. Similarly, modeling hidden and output unit activation to base diameter in a TM view, provided a more useful view (for recall testing) than modeling biases to the same symbol.
- In some investigations, the absence of a particular variable in a view precluded drawing conclusions about the behavior of the final network. For example, the absence of a bias mapping concurrent with activation values in a Texture Map view made it difficult to describe the behavior of the final feature detectors.

6.6 Findings — System (View) Related

View Effectiveness

According to the findings from the structured investigations and the comments from invited viewers, the dynamic views of NetViz are capable of informative displays of ANN learning, and are worth further study. Some proposed behaviors from investigations 5, 6, 7, and 8 are examples where animation was key. It is important to note that movie-type animated views provided this sort of

extraordinary information. The movie view types were also found to be the most engaging and effective, as long as:

- they animated at a visually pleasing rate
- their view information content was from medium to low
- the information modeled was continuously in view

The following techniques were eventually determined to help optimize movie type animations:

- using plotting scales that amplify the action of modeled variables
- setting the view to update frequently
- avoiding the use of a stage delay setting greater than zero (RF view)

Relative animation speed was found to be a key issue. Animations that were slow and that periodically did not depict changes were difficult to follow. The objective of a pleasing animation speed was encumbered by the concomitant objective of multiple views and view options. Some changes were made to the prototype after initial observations (e.g., implementation of record/playback facility) that aimed to improve animation speed. The first was a simple record and playback mechanism that would eliminate the overhead contributed by algorithm calculations when playing back a previously recorded algorithm. Another mechanism provided is the simple re-execution of a display space's display list (see Section 5.2.2: Windowing Interface). Only the latter capability provided the type of animation increase sought, though views generally play back too fast this way.

Two related critical issues were screen resolution and screen real estate. In order to provide the entire plot with adequate detail of symbols (TM plot) and traces (DP), viewports nearly the size of the screen are required. The situation is exacerbated as the number of network units and weights increases. Technically NetViz can handle larger ANNs, though currently, only part of the view they model would fit. The viewer would then need to zoom in/out and scroll about to view hidden information. In such a setup though, animated information is lost. Consequently, future view design needs to consider issues of view size and resolution requirements.

Symbol -Variable Mapping

The investigations helped determine those symbols that are effective modelers of variables and those that are less effective. The pulse in the DP view is an example of an overall effective symbol. The TM arms and legs are effective with respect to the compelling shapes arrived at through angle changes. (Interpreting their shapes during learning is, however, still difficult.)

Three major shortcomings related to symbol mapping were found:

- the inability to determine the *sign* of weights and biases
- the "flashing" of activation and error frames to depict algorithm flow
- the limitation of visualizing one of two variables at one time because they map to the same symbol

A suggested solution for the first shortcoming is the to use a second coding linked to weight and bias-square size changes. For example, coloring the square differently when positive than when negative. An extended solution would be to separate the sign from the value's magnitude; then, zero values would be rendered as a "dot" (or not rendered at all), and larger (absolute) values as larger squares, colored according to their sign.

The second shortcoming points out a particular case where there is the need for smooth transitions between states. Maintaining that the concept of modeling algorithm flow has merit, the general solution involves keeping all evolving variables in constant view, and employing some form of neutral (i.e., that is not of the algorithm) indicator around (or within) symbols that are being updated.

A recommendation for the third issue is to provide at least one symbol for each variable and if possible (and meaningful) allow variables to map to different symbols in the view's symbol set.

One final finding was the desire for a conventional static view of the ANN being visualized. This view should include annotations corresponding to the particular labels in the dynamic view.

System Interactivity

NetViz is a prototype system with intentional limitations. Nevertheless, exercising the system pointed out certain severe limitations. Two major limitations are:

- the inability to change variable mappings dynamically
- the inability to change symbol types

Both of these limitations are currently overcome by way of saving a snap-shot of the algorithm variables, closing the existing view, and opening a new view defined with the desired mappings. The first limitation refers to the inability for the user to change the algorithm variable linked to a particular symbol. For example, to change the mapping of weight in TM arms to delta weight. The second refers to the inability for a user to change the symbol mapped to a variable, for example from "pulse" to "color" in a DP view. The desire is to implement commands of the sort: *change_map_var weight [to] dweight, or change_map_symbol pulse [to] color,* as analogs to the commands used to adjust plotting scale.

Another system shortcoming was found with the rendering control of views. The *rendering update flag* (values: epoch and pattern) affects all active views. Furthermore, the effect of its setting is easily confused with the similarly termed *update flag*. The main problem is the rendering outcome when the rendering update flag is set to "pattern." In TM and RF views, at each update period (determined by the view's render cycle), all training patterns are rendered in succession. This produces a very confusing picture, because activation and output error values can be very different for each pattern; in a movie, the impression of an evolution is disrupted. (For the investigations, the "epoch" setting was used, and views were rendered from the perspective of the last training pattern.)

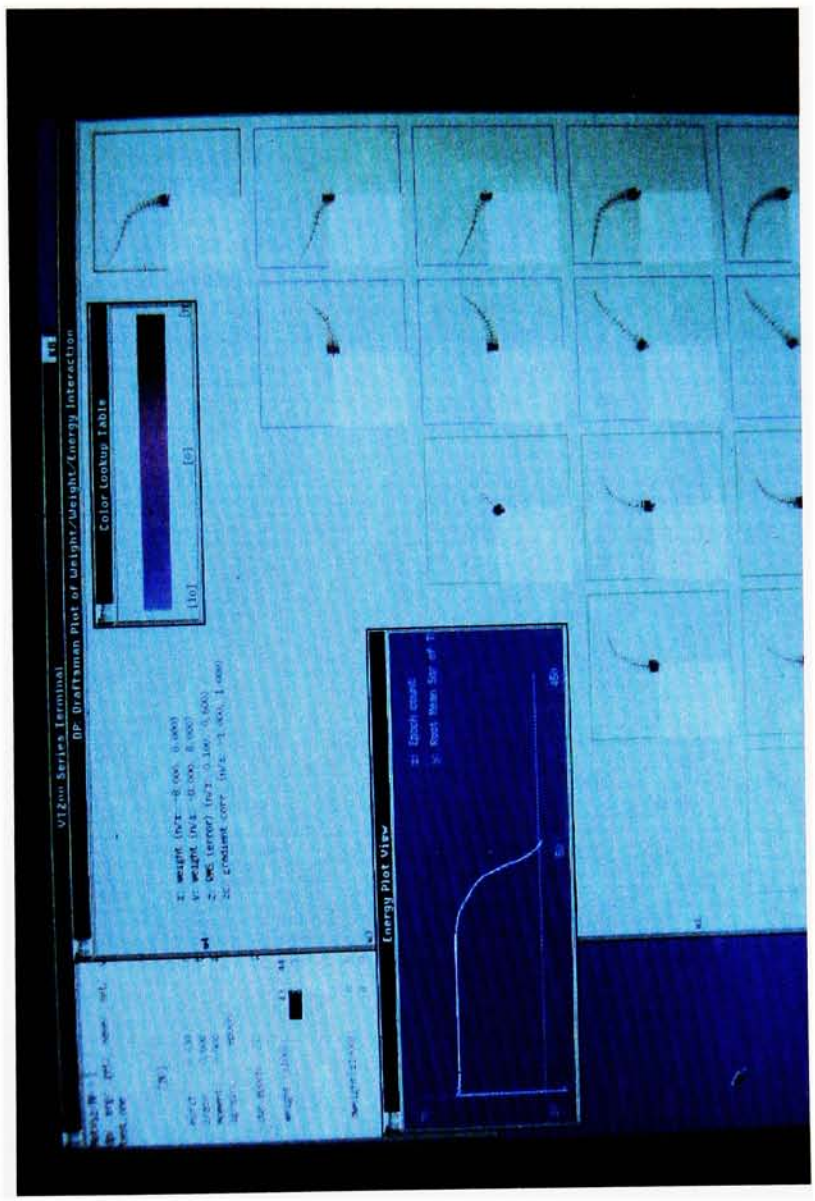


Photo 6-1. XOR learning - DP view (and ERG view).

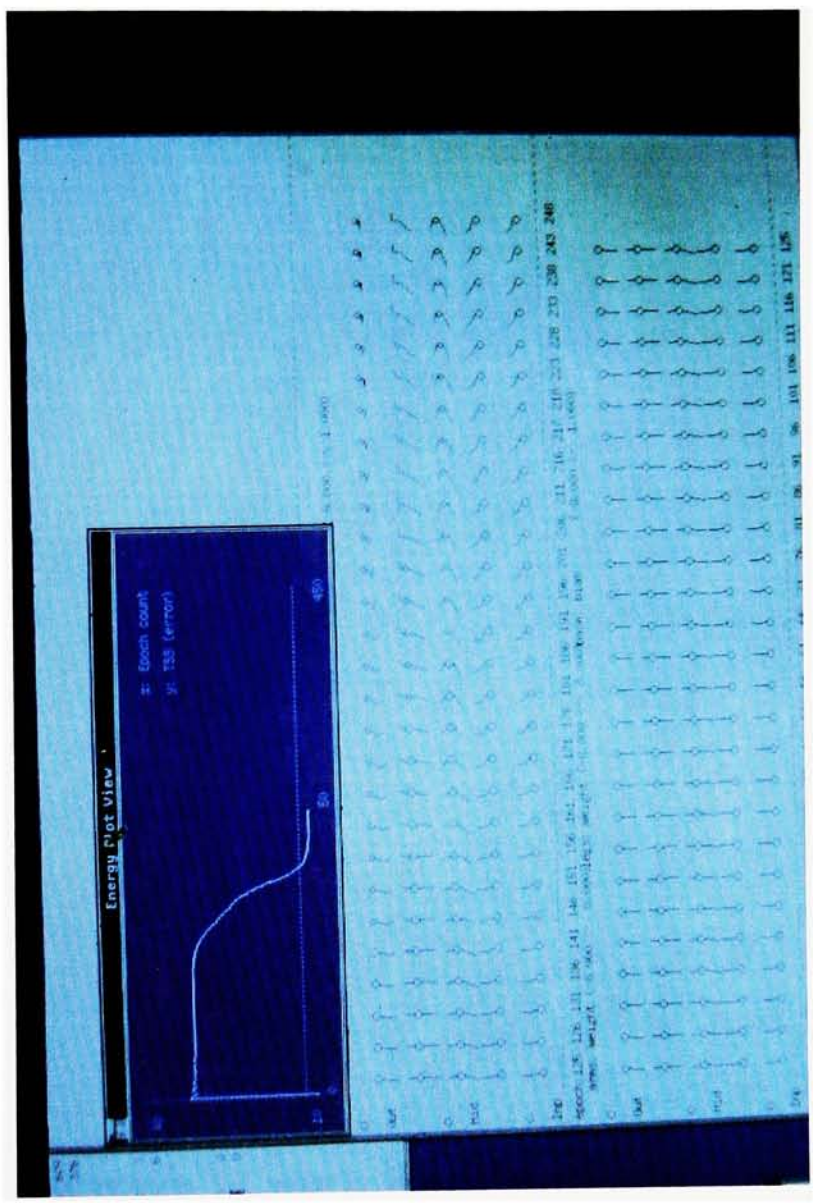


Photo 6-2. XOR learning - TM view (and ERG view).

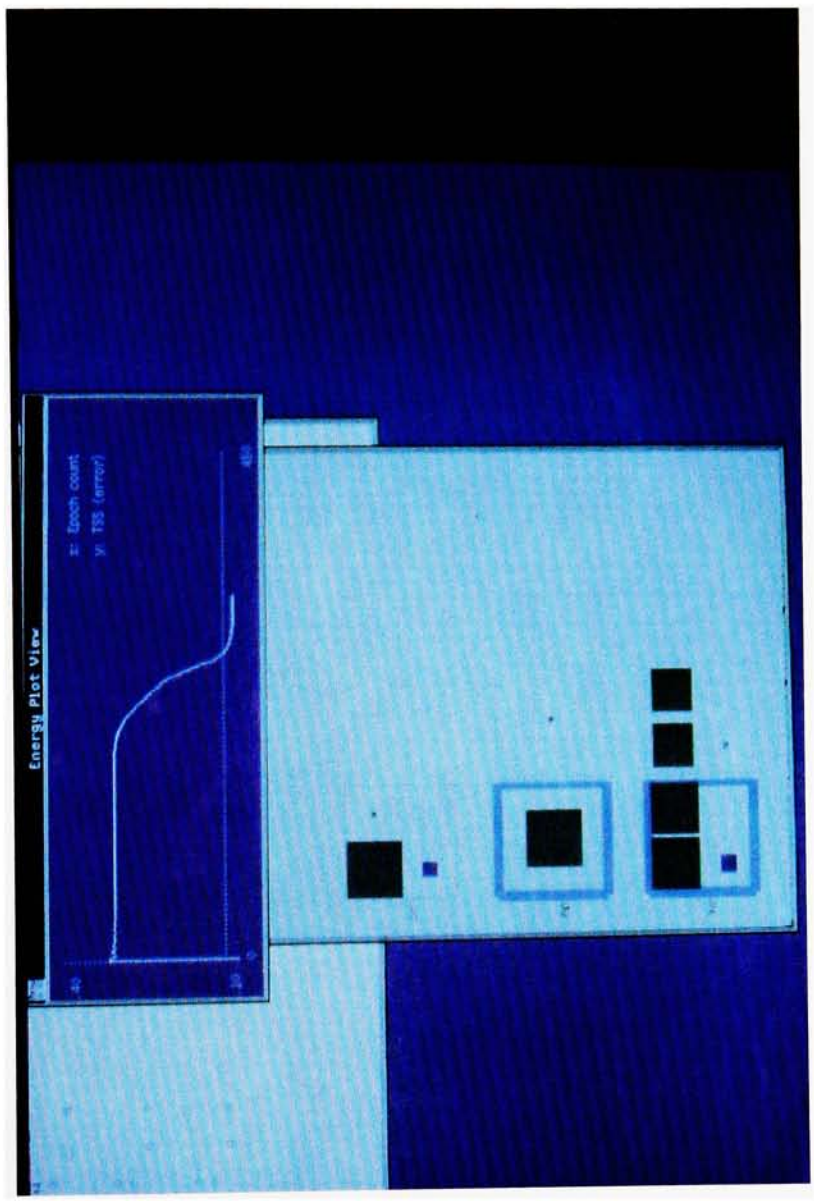


Photo 6-3. XOR learning - RF view (and ERG view).

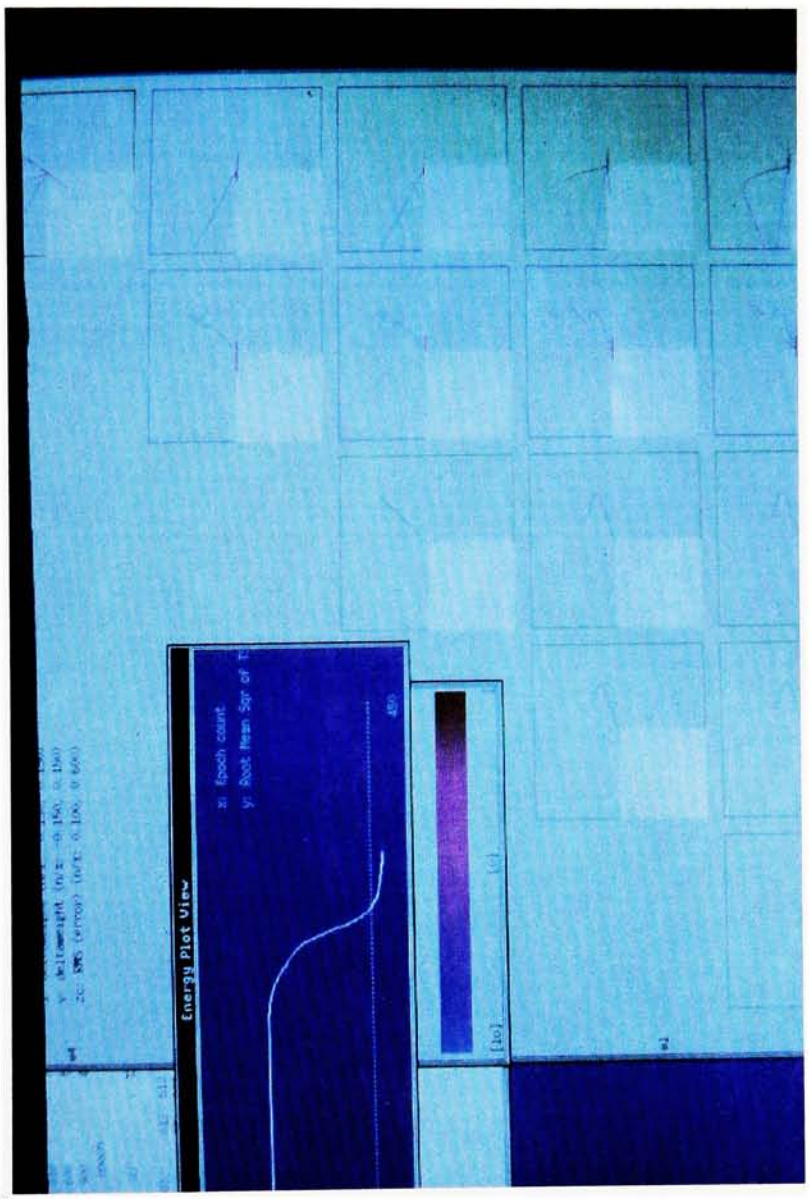


Photo 6-4. Delta weights - DP view at 317 epochs (convergence).

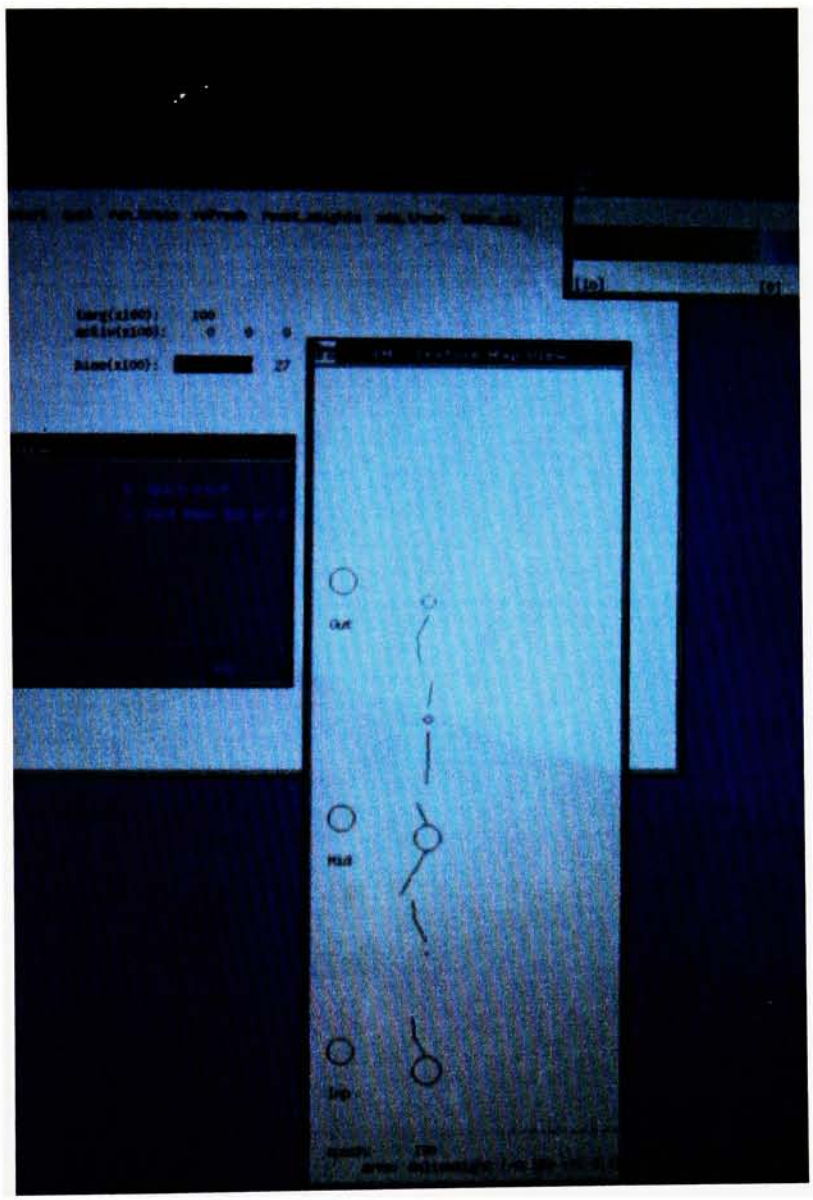


Photo 6-5. TM movie frame at 190 epochs.

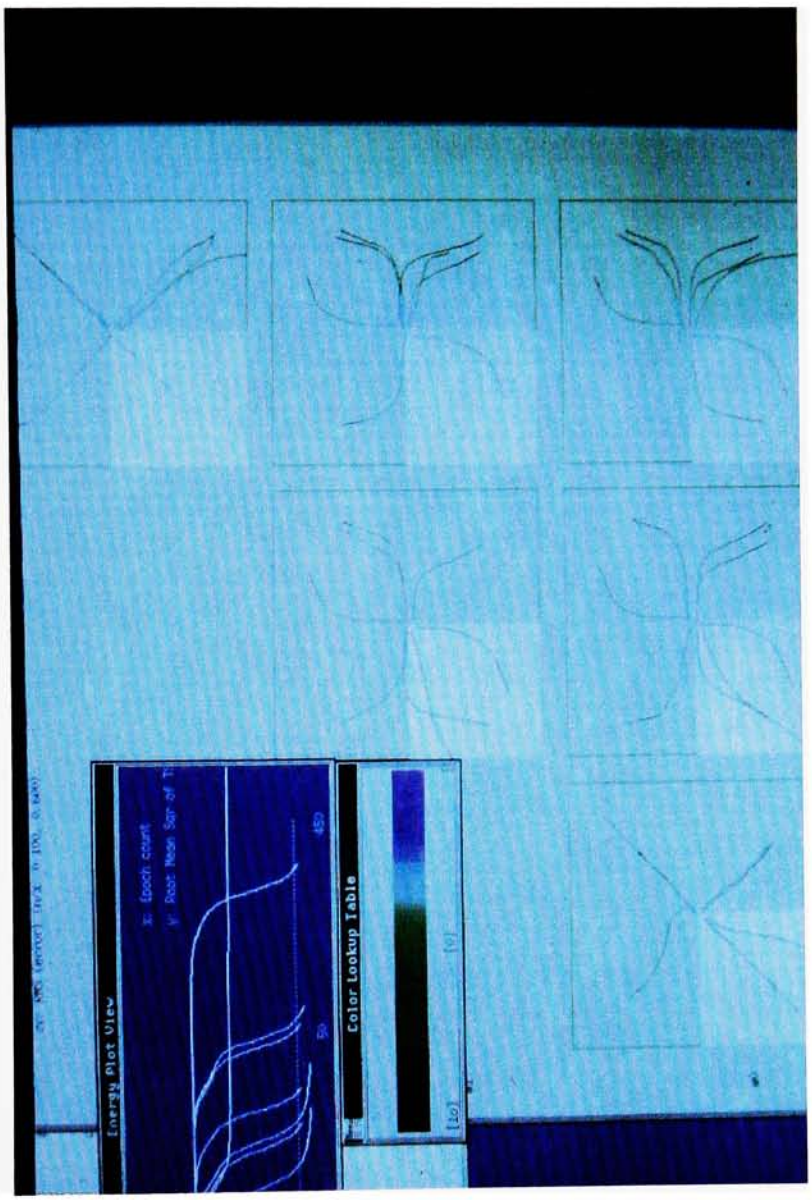


Photo 6-6. Composite view of trials in a DP and corresponding Energy plot view.

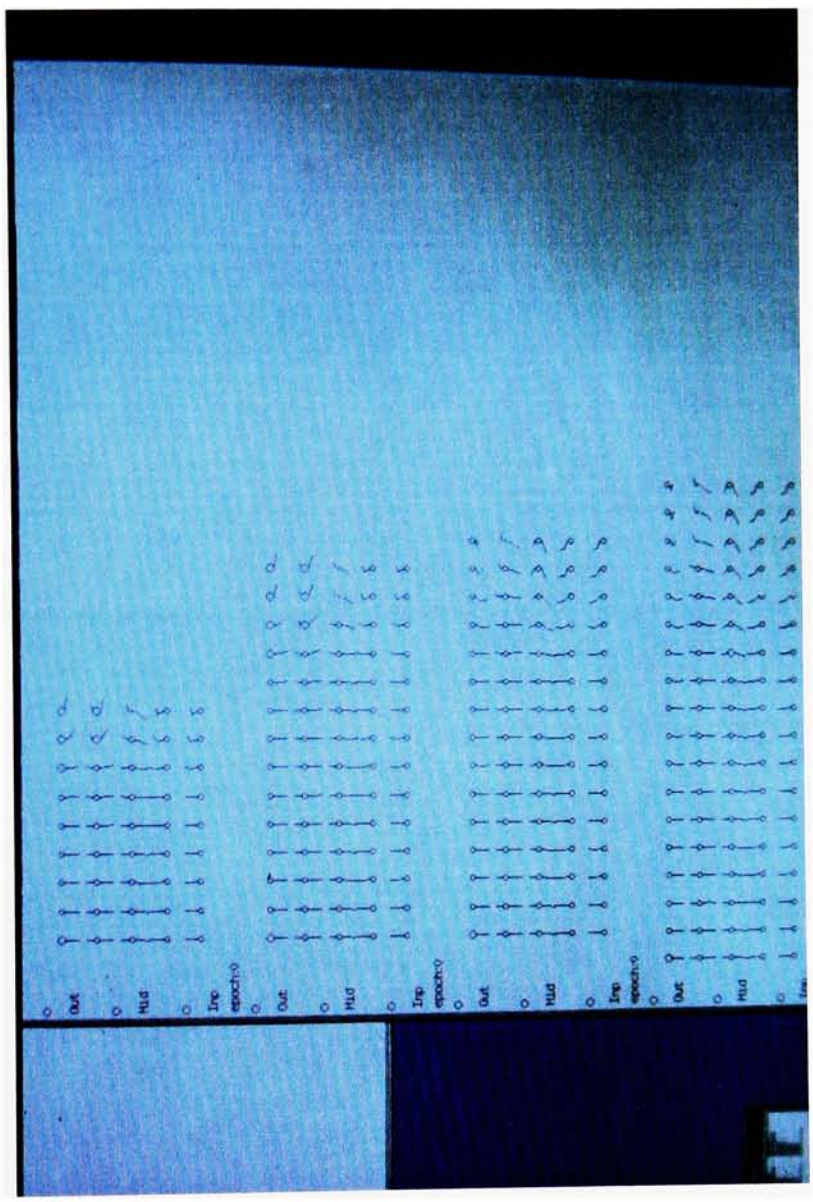


Photo 6-7. Composite TM view of XOR learning at different learning rates: 0.4, 0.8, 1.2, and 1.8.

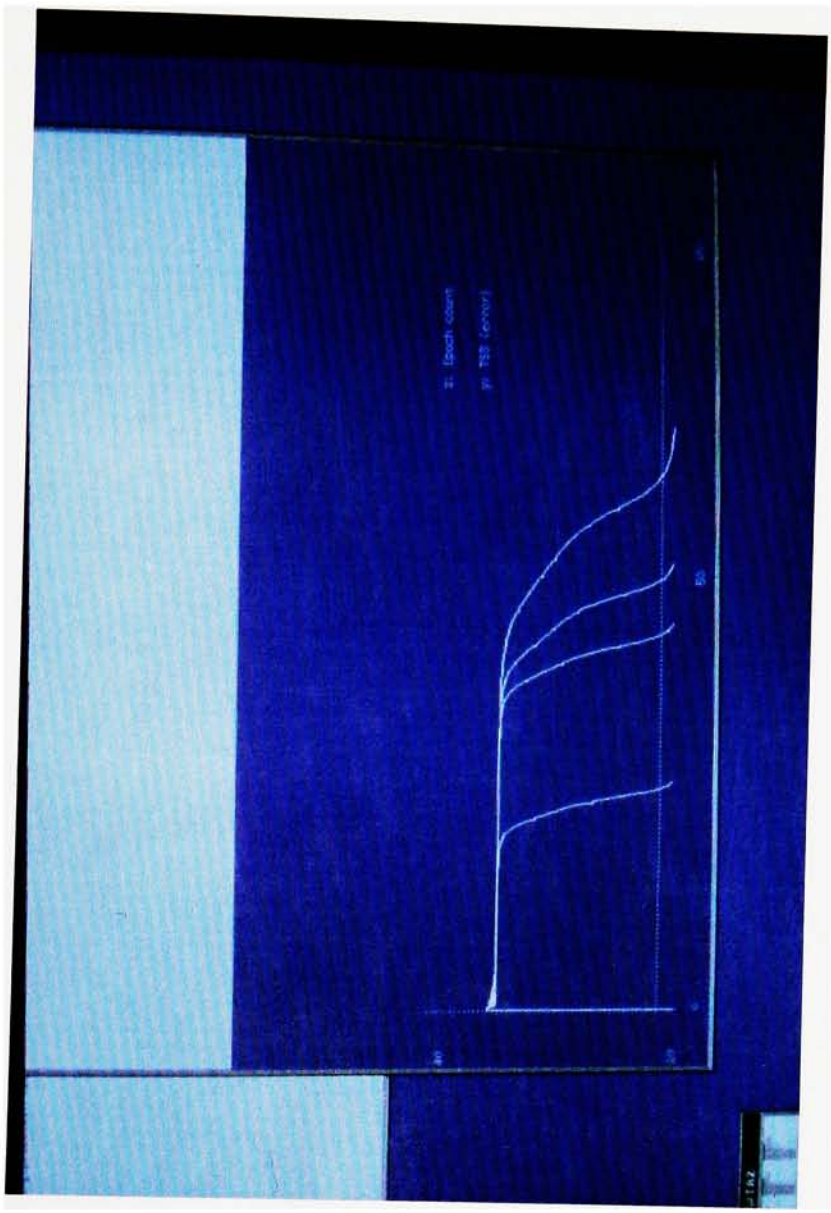


Photo 6-8. Energy plot corresponding to Photo 6-7.

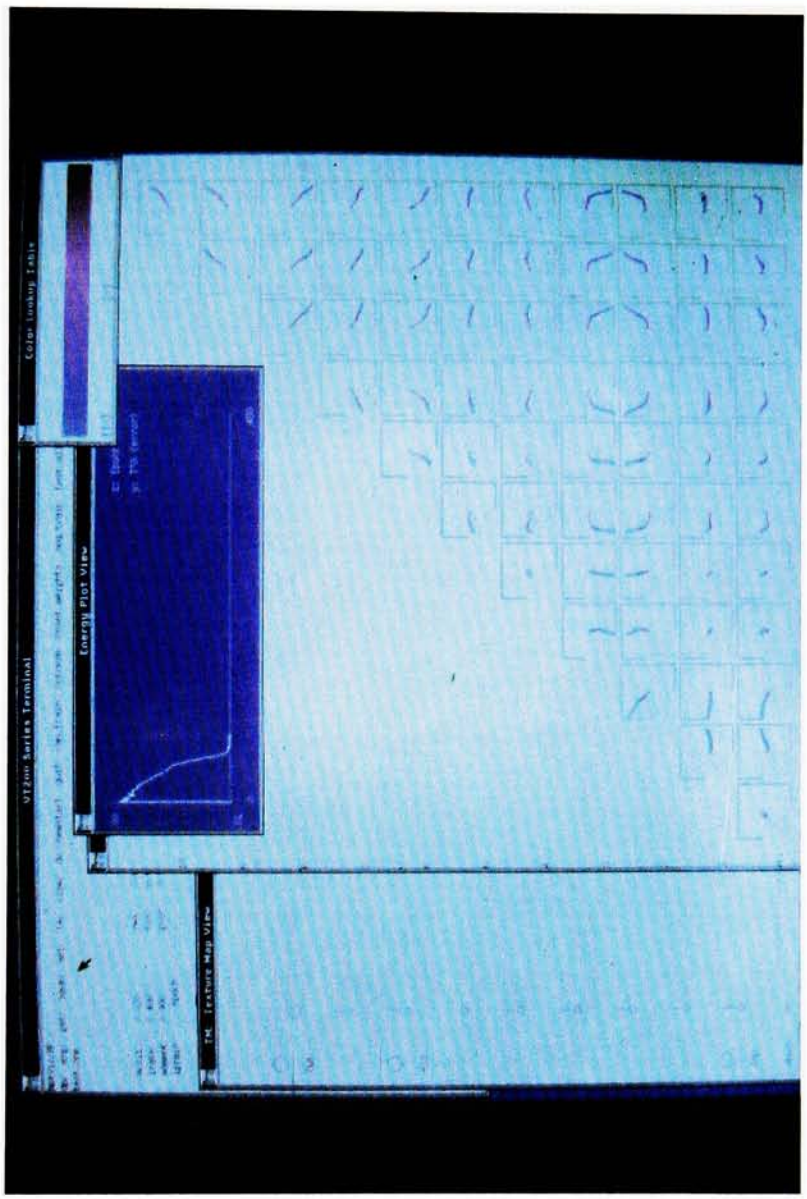


Photo 6-9. Final DP view of symmetry learning.

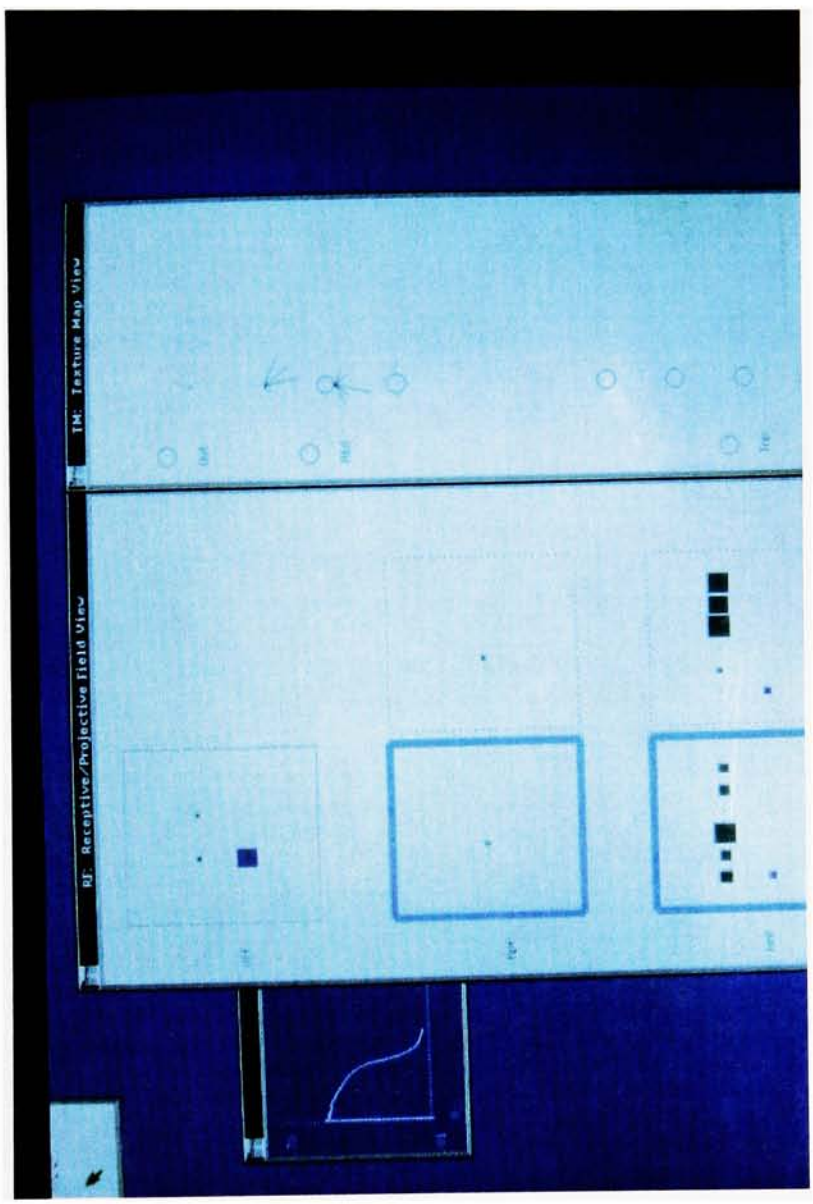


Photo 6-10. Final RF and TM views showing feature detectors.

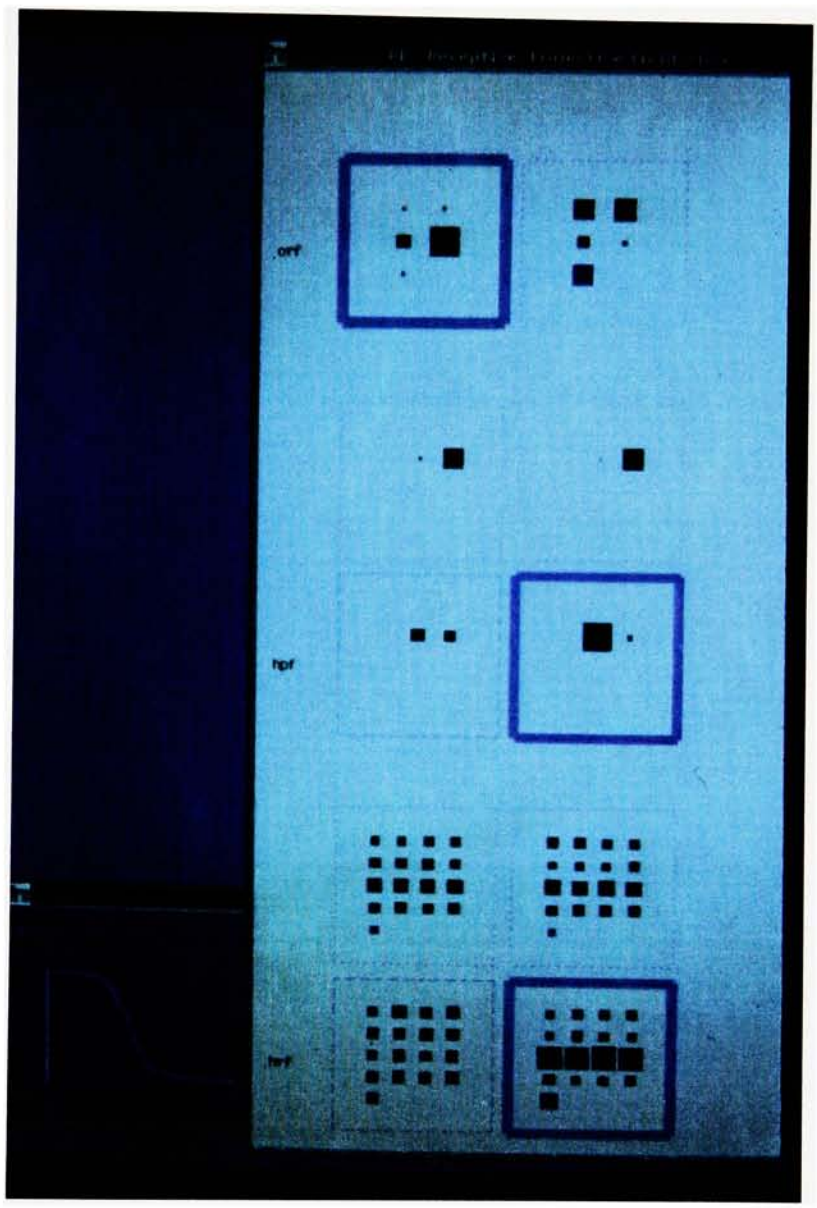


Photo 6-11. Final RF view of the orientation classifier showing feature detectors for input "vertical-2nd-column-left."

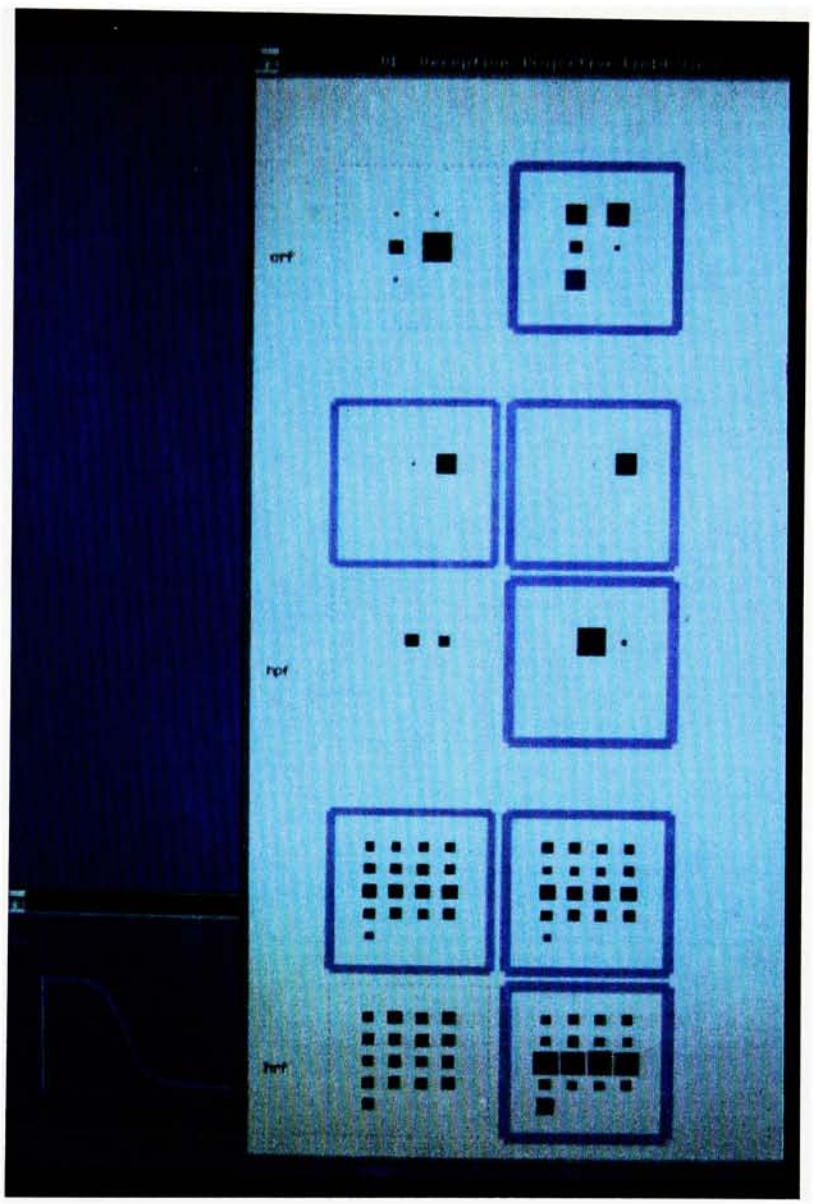


Photo 6-12. RF view showing feature detectors for input "horizontal-2nd-from-bottom."

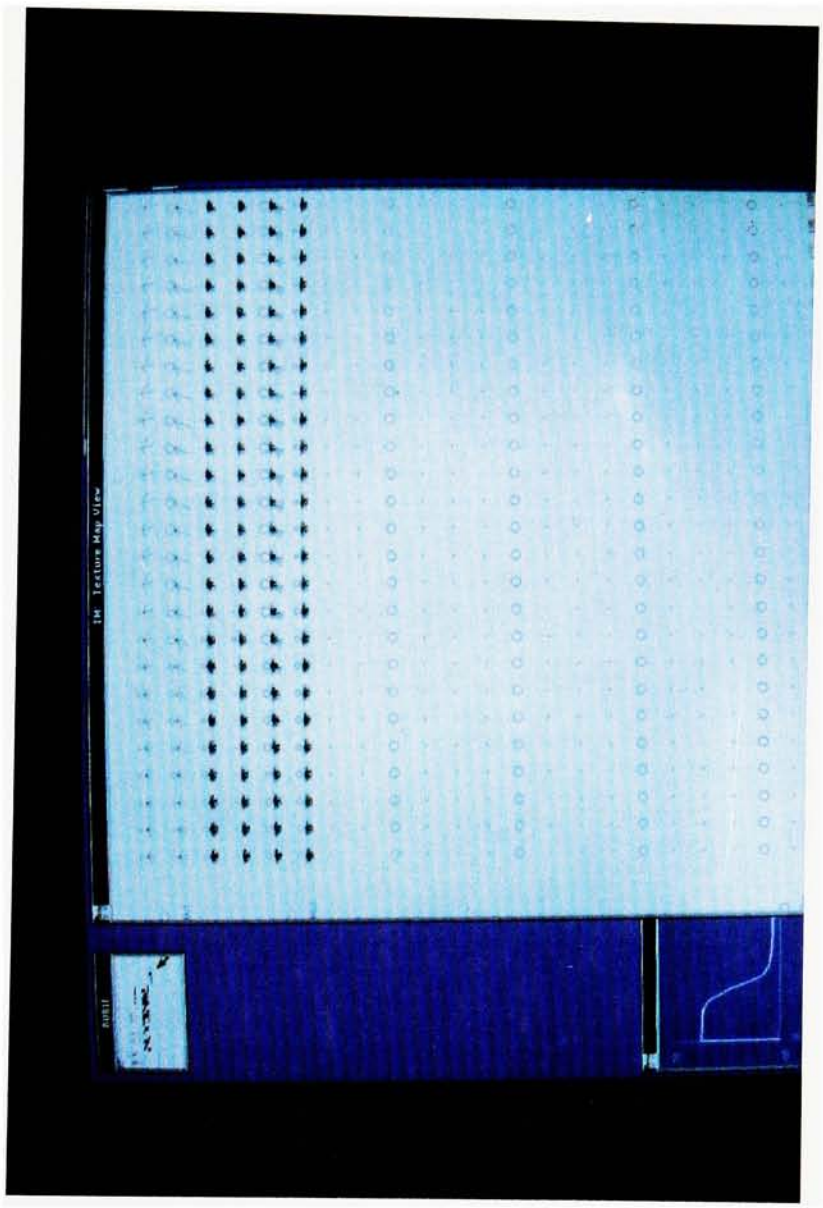


Photo 6-13. Final TM view of the trained orientation classifier.

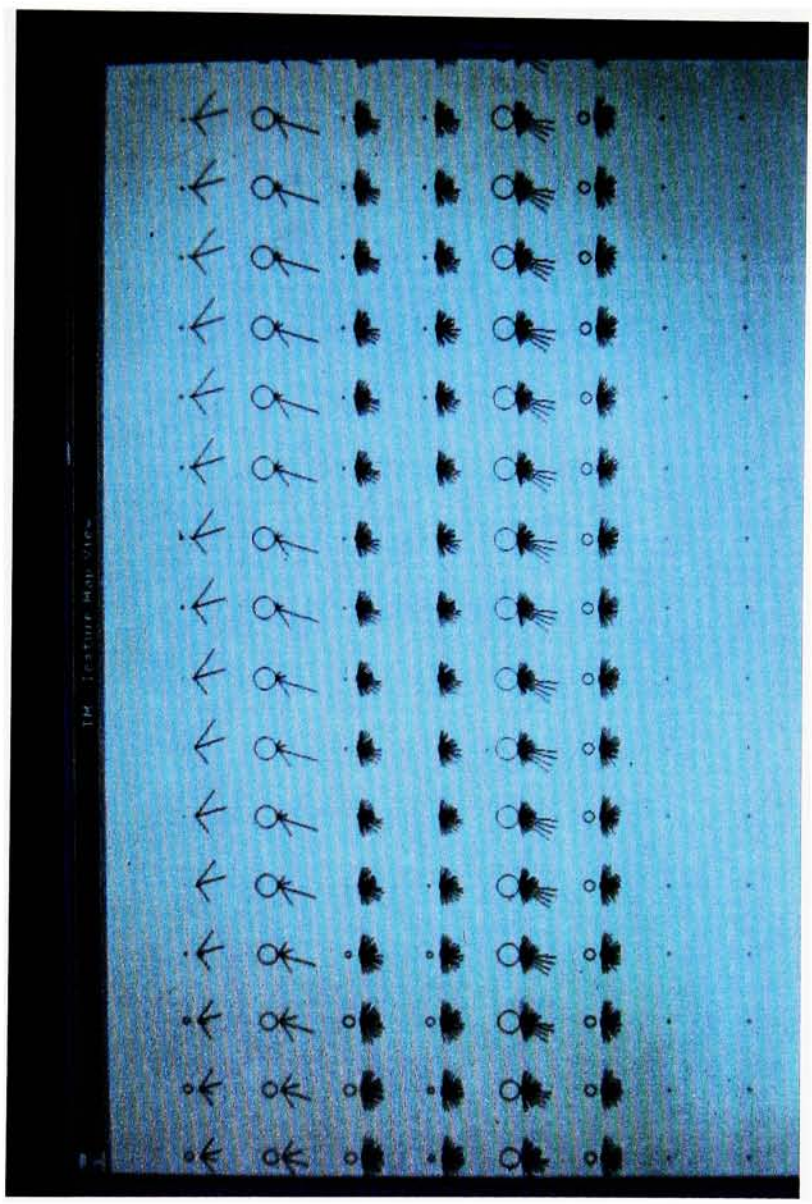


Photo 6-14. TM view magnified, from epochs 49 to 169.

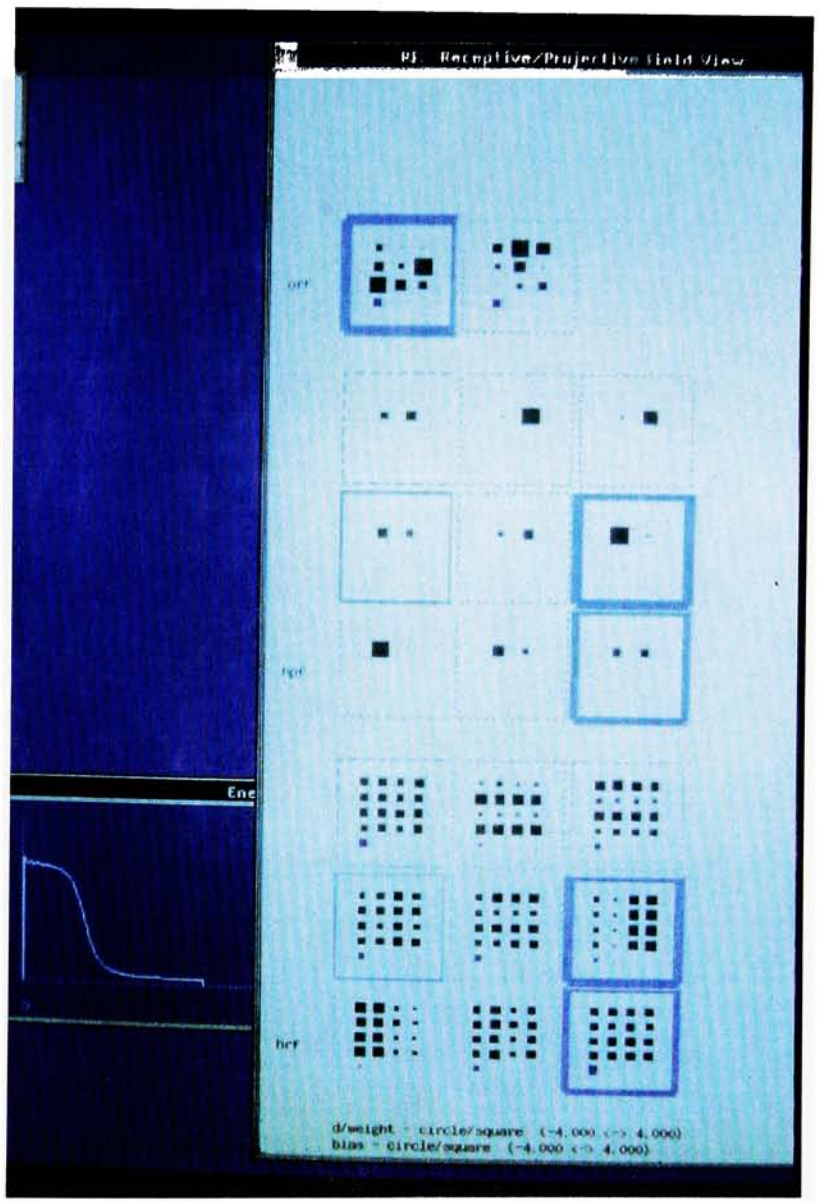


Photo 6-15. Final RF view showing feature detectors for input "vertical-3rd-column-right."

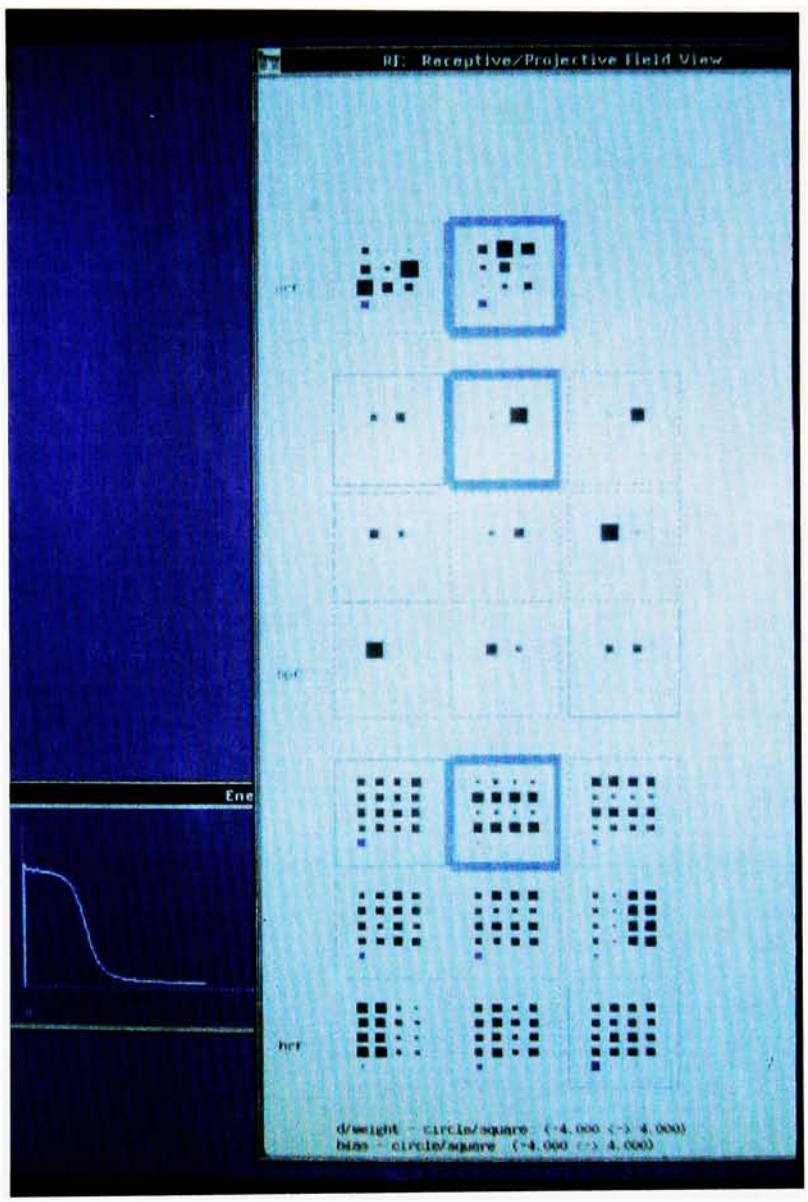


Photo 6-16. Final RF view showing feature detectors for input "horizontal-2nd-row-top."

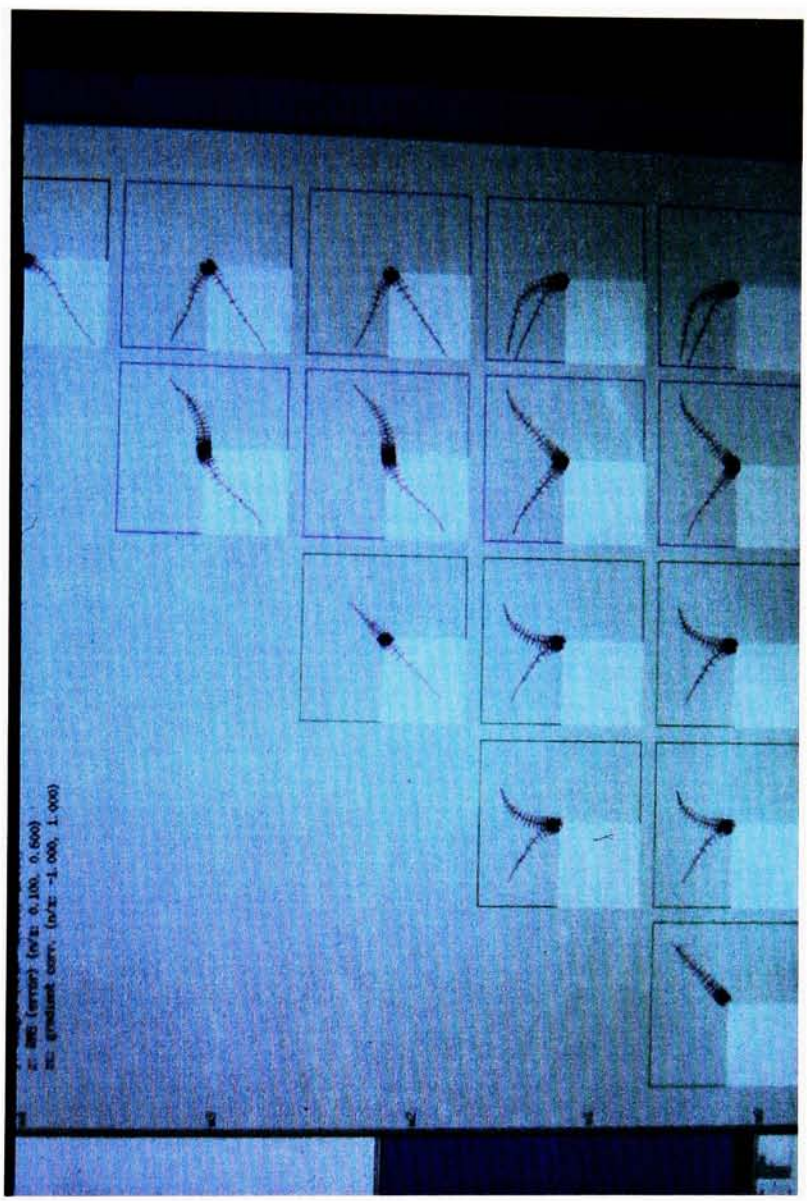


Photo 6-17. DP view of XOR with traces formed at a learning rate of 0.6 and 1.2.

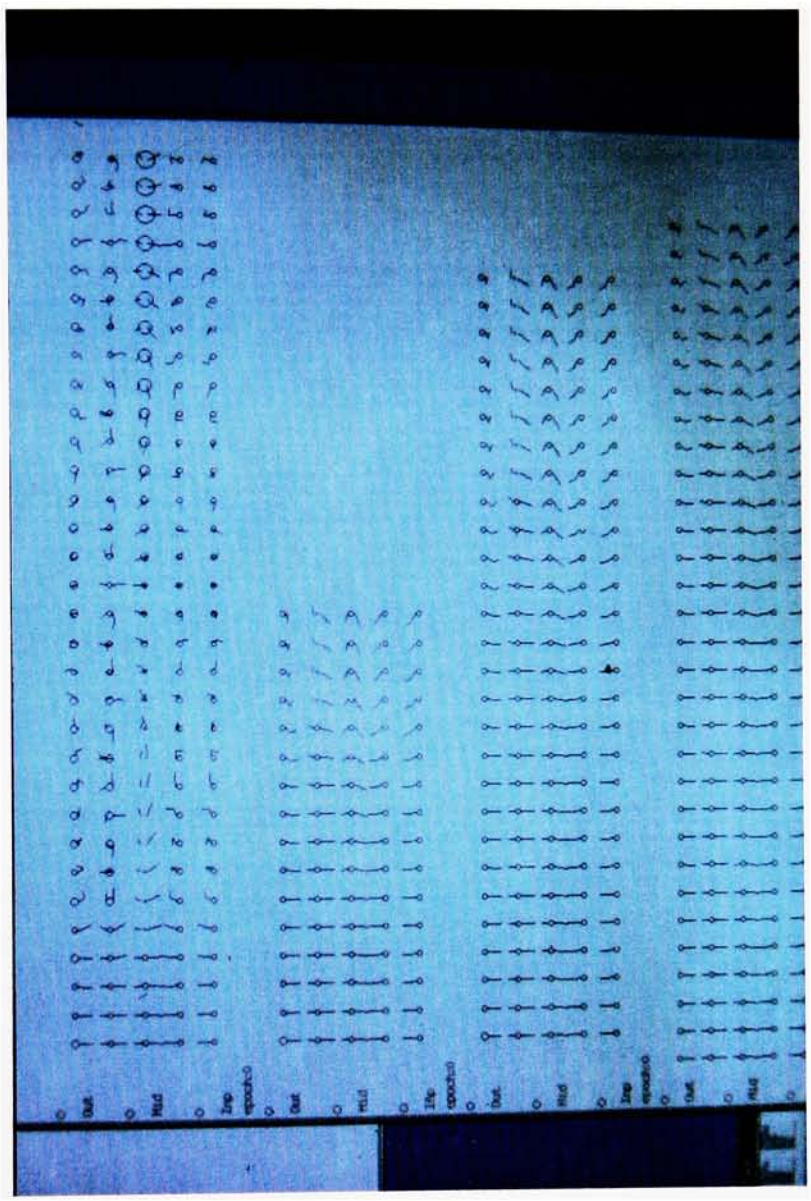


Photo 6-18. TM view depicting learning results with four different momentum constants.

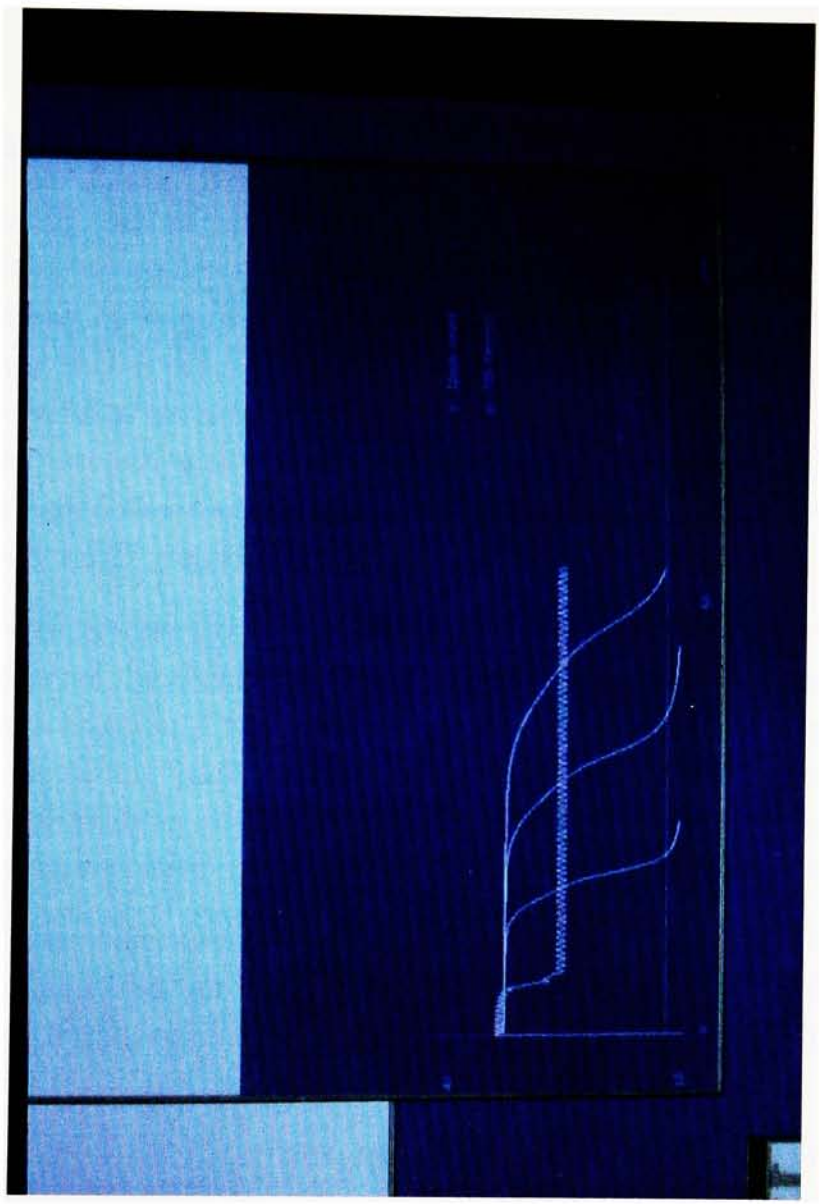


Photo 6-19. Energy plot view depicting learning results with four different momentum constants.

7

Conclusions and Future Work

7.1 Conclusions

The work reported in this thesis suggests, in general, that the exploration of ANN back-propagation learning using algorithm animation in an interactive environment shows promise. The ability by which an animated view can project useful information about the back-propagation algorithm depends on the design of the view, what elements of the algorithm are visualized, and to what degree the viewer recognizes elements of the view (e.g., from external analogies). Many of the animated views explored, directly portrayed behaviors that have otherwise only been verbally described in the published literature.

Principles from three primary domains were applied to develop a view design framework: graphical data analysis, perception theories of good form, and algorithm animation. This framework, in conjunction with the back-propagation learning algorithm, guided the development of three different view designs for algorithm animation. An interactive, algorithm animation prototype environment was developed in order to informally verify the framework.

Three primary objectives set the direction and scope of the overall project. The remainder of this section reviews conclusions reached for each of the primary objectives. In the section following, recommendations for related future work are discussed.

1. *Determine a view design framework suitable for the AA of back-propagation* —

There are two aspects to this objective: one regarding the functionality of the individual design attributes, and another regarding the applicability of the framework.

The existing definitions for attributes such as symmetry, complexity, and regularity, which are derived from static views, seem inappropriate for dynamic

views. They also seem inappropriate for views whose layout can increase or decrease significantly as a result of changes in the ANN topology, for example, the addition of a single hidden unit which then results in an additional weight matrix dimension equal to the number of input units. A more appropriate approach would be to decompose such views into a hierarchical order of layers[1], where each layer contained a related group of view components, and to apply the existing definitions to each layer. This revised approach would also require that the AA taxonomy (Section 2.1.1) be refined such that its dimensions are applicable to each layer.

A general consequence of designing views within a (semi) formal framework is the exclusion of information and feedback expected by the viewer. Informal findings suggest that the availability of desired/expected information in certain circumstances is paramount to the "goodness" of the view's design. On the other hand, a view is more engaging when designed to be symmetric, regular, and complex, both at the view-frame level and at the dynamic level when "something important" occurs in the learning algorithm.

In general, small value changes (of algorithm variables) were depicted best through changes in segment angle and in the size of filled squares. Larger value changes were depicted better through changes in spatial (x,y) plotting, the size of filled circles, and line weight and style.

Temporal mapping was unsatisfactory when: updates were very discrete; when the time between updates was greater than (approximately) the time taken to (re)render a frame; and when the view had high motion complexity and at the same time was too wide, i.e., in that the viewer is required to extensively move their head or eyes. Views were less engaging during periods when motion was imperceptible (e.g., due to algorithm value changes that were too small for the mapping scale), and more engaging when motion was continuous and smooth, particularly when using the "fast" animation of the display list playback.

It should be noted that the recommended principles of effective design were not fully applied in the design of NetViz views for several reasons. One primary reasons is that the merging of effective design principles often conflicted with ways that were meaningful for portraying algorithm variables and operations.

---

1 A hierarchy of layers is suggested, from layer models central to computer graphics systems such as computer aided design (CAD) and computer aided illustration.

Another reason is that certain key principles were discovered after the view design "freeze date." Nevertheless, the fact that the view designs were not optimal served to emphasize the importance of unconsidered design principles.

2. *Determine those algorithm variables that best portray learning and how —*

An informative view enables the viewer to extract evidence of algorithm processes. The findings suggest that the degree of information portrayed is in part dependent on the algorithm variables monitored and in part on the view through which it is projected. Two predominant criteria for an informative view are that:

- the viewer recognizes a pattern among elements in the view, either by matching a predicted (mental) image, or after repeated study and analysis of the view's characteristics
- the view displays predominantly the information that the viewer wants to monitor. Both extraneous and missing information seem to reduce the effectiveness of a view

In the views designed, most of the selected algorithm variables (Table 4(A)) portrayed some form of algorithm behavior, e.g., convergence, the formation of feature maps, etc. The weight error derivative (WED), however, is one variable that did not. A possible reason for this is that it was inappropriately depicted in NetViz views. In general, views formed from composites of several algorithm variables, such as the weight, bias and activation of a network unit, were more informative than each variable alone.

The RMS and TSS error variables best depicted the behavior of convergence (to global minima) as well as the development of local minima. Although both of these variables were visualized through ERG plots and DP views, the former view type seemed to be the most compelling. A likely reason for this is that a time-series plot is a more familiar view for data analysis.

Weights displayed in a regular arrangement (such as in a RF tile of weight-squares) in conjunction with a proximate symbol for activation were also informative. A similar suggestion is that this image was similar to other views that the type of people familiar with NetViz had been exposed to (e.g., Hinton

maps). Another reason is that this image represented (in part) an "objects" with describable behavior, in this case, a feature detector.

In contrast, weights in a DP plot were informative only at a much higher level, where many, complete DP plots were used to categorize learning solutions.

3. *Determine a system architecture that facilitates objectives 1 and 2 —*

Four key architectural elements were found most important:

- use of minimal, unobtrusive algorithm annotation
- provisions for extensive run-time user control of the system
- separation of the algorithm and animation sub-systems
- use of object-oriented design and implementation

These were determined to be key in two ways: through the level of use of capabilities (functions, design, etc.) representative of each element, and through the level of desired availability of unimplemented capabilities.

The current implementation of interesting events does not adequately support views that do not explicitly depict discrete phase-related algorithm flow. This type of view control as well as greater set up flexibility could be achieved by increasing the algorithm related information transferred through the interesting event call to the view.

Frequently used commands that characterized the importance of run-time user control were:

- interrupt and pause a learning process
- invoke arbitrary commands while "paused"
- resetting the network, with the "original" weights (test replication)
- single-stepping through the algorithm

Two unimplemented capabilities[1] that exemplify the extent of important user control for any future system are:

- the ability to change an algorithm variable's mapped symbol to another
- the ability to re-map variables to a particular symbol

1 These capabilities are provided, though in an indirect and non-intuitive way.

The separation of the algorithm generator and the animation generator allowed the incremental integration of each view as it was implemented. Changes to the algorithm code annotation was not required; "flag statements," designating the update of an algorithm variable, were included for all potentially visualized variables during the first annotation.

7.2 Suggested Future Work

Several conclusions indicate a variety of shortcomings in the problem setup, the design of views, and in the system implementation. Many conclusions also raise new questions. The untested capabilities of the current prototype system beckon further research with other ANN classifiers and with other view configurations. The potential future work from this project can be grouped into three domains:

- usability research
- user interface design research
- system design research

7.2.1 Usability Research

The views developed in this work have been evaluated primarily by this author. Although presentations of the system to invited participants provided valuable feedback, such evaluations were informal. The next logical step would be to incorporate the suggested improvements to the current view design framework and then conduct user studies. User studies involve conducting experiments with a group of test subjects that represent a larger group of target NetViz users. Two broad areas for experimentation suggested by the work here are in view perception (how engaging a view is) and performance (how informative a view is).

A class of experiments addressing the informativeness of a view would explore the boundary conditions of view attributes. For example, an experiment might test complexity, using a TM plot and a symmetry classifier ANN. The essence of the experimental procedure would involve test views that incrementally vary in view complexity (in terms of the view design framework rather than the user's

perception). The experimental data collected would be the response time to the onset of a characteristic event (e.g., convergence) as a function of view complexity. The assumption is that there is an inverted 'U' relationship between complexity and response time.

Presumably, there are view designs (or settings for flexible views) that can be highly engaging as well as informative, though not necessarily at the same time. Therefore, any future algorithm animation system should consider flexible view designs, whereby views can be made to lean in one direction over the other. For example, algorithm animation used for basic instruction would initially require predominantly engaging views, and for research, predominantly informative views would be expected.

7.2.2 User Interface Design Research

The problems encountered in NetViz for manipulating graphics related commands through the command line interface were predicted at the beginning of this project. A graphical user interface (GUI) to NetViz was determined early on in this project to be a project in itself. The current NetViz prototype provides a basis and opportunity for research of a direct manipulation GUI.

User interface design, and in particular GUI design, is a multi-discipline field incorporating human factors, graphic design, experimental psychology, and software design. Software development of a GUI is greatly facilitated by many general-purpose and hardware transportable toolkits.

Typical elements of a GUI that would be appropriate for NetViz, include windows, menus, dialog boxes, slider devices, and soft-buttons. Windows are already used. What is suggested for future work here involves applying these user interface building blocks following principles of good user interface design. Examples where a GUI would improve the current NetViz prototype, organized in three broad categories, are:

Network and View building — Network and view layouts could be "assembled" by directly manipulating template view elements into a view frame. Each of the view elements could encapsulate behaviors, that could be edited to further customize the view according to needs.

View control — The mouse should be used to control window order, and to designate the current view. A control panel of soft-buttons could provide view-wide controls, such as pause/continue, enlarge/reduce, pan/scroll, record, save image, etc. Other useful controls at the level of tiles and similar smaller units could include changing plotting scales using sliders and retrieving numerical values from a direct manipulation inquiry on a symbol.

Global process control — Network set up, the loading of weight and pattern files, and the specification of other network parameters could be controlled through a combination of menus and soft-button control panels.

7.2.3 System Design Research

The NetViz prototype has paved the way towards developing a more complete and general-purpose system for ANN learning algorithm animation. The steps to developing such a system should incorporate and expand on the system design strategies found to contribute to the prototype's success. Suggestions for the user interface described previously should also be incorporated. Research of general-purpose ANN simulators should provide strategies for constructing arbitrary learning algorithms. From the work of this project, the following system design issues are suggested for a future system.

Enhanced and New View Designs

One of the shortcomings of the DP view is that its use is precluded by even medium sized ANNs. A more flexible way of providing this view is for the user to specify a subset of the default set of weight-weight tiles, and to arrange them in a layout of their choice, with the capability of real-time re-specification.

A possible enhancement of the TM movie mapping arms and legs ("growth") derived from the system by [Mezr84] would be the option of displaying a "temporal neighborhood." In this new view, each symbol would maintain in view the five (say) most recent update-frames of arms and legs. Each "temporal layer" of symbols would be rendered in a color that decreases in contrast with the background in direct relation to the age of the layer. Thus arms and legs would appear to "fade into time." It is predicted that this view would improve the

perception of continuity in the animation because view transitions would appear less discrete.

The epoch/time-series plot of the ERG view should be explored with other algorithm variables. In addition , arbitrary combinations of variables should be allowed to be plotted together. This simple, and easily interpretable view is underused in NetViz.

In general, NetViz has not exploited the display of intermediate algorithm variables. One particular example is the product of weight and signal. This product is an "active" depiction of the excitory and inhibitory nature of network links. TM plot arms and legs (spikes or growth) are suggested symbol candidates for this mapping.

Compound Views

An improvement to the TM and RF views, and to future views, is to support the visualization selected input patterns from the training set. A suggested approach involves creating a tiled layout in which the normal view is reduced to fit into tile; a number of tiles equal to the number of patterns being monitored would appear in the window. The ultimate effect is to have multiple, parallel movies, showing the specific evolution of selected input patterns.

Towards True Object-Orientation

Both ANN and AA systems are intrinsically suitable for true object-oriented design and implementation. The general attraction of object-oriented design is the potential for a flexible system made so through "building blocks," for both ANN construction and simulation and AA view construction and animation.

The object-orientedness of ANNs is in that each unit of the network is a self-contained processing unit, definable as the superclass of a hierarchy of classes and methods. Systems such as P3 [Zips86], Asprin/MIGRAINE [Wiel88], and a system [Loe88] based on HISDL[1] and P3, are sophisticated examples of object-oriented ANN constructors and simulators.

1 HISDL is a structured description language for specifying logic circuitry [Lim82: cited in Loe88].

In AA systems, a view can be similarly represented by a superclass, with a hierarchy of classes defining increasingly elemental view components with unique temporal definitions. Methods (analogous to procedure calls in objected-oriented programming) of those classes control local and global temporal updating. Some early AA systems, developed on Smalltalk [Duis86], [Lond85], and more recently, Tango [Stas90], are examples of object-orientation in this domain.

In the proposed advanced algorithm animation system, another set of classes would describe the algorithm variable-view symbol mapping , and interesting events would be handled by methods of the variable-view symbol class instances.

7.3 Final Thoughts

The investigations conducted with NetViz explored a very small set of ANN classifiers, and relatively few effects caused by varying network topology (e.g., increasing/decreasing the number of hidden and output units). The algorithm generator component currently supports simulating more complex classifier architectures. Possibilities include: topologies with more than one hidden layer; "linking" arbitrary weights (whereby weight adjustments are the same for all linked weights); constraining arbitrary weights to develop either excitory, inhibitory or neutral behavior; and assigning unique learning rates to each network unit (except input units). Similarly unexplored was the use of command script files. In future studies, script files could be used to control complex learning schedules, automatically change training pattern sets during training, dynamically alter the network topology (e.g., adding or removing hidden units), as well as automatically saving snap-shots of views or weight vectors.

The degree to which a view is engaging and informative appears to be independent of one another. The argument by Kaplan [Kapl82] that "informationally rich views are more engaging" does not seem to apply to algorithm animated views. The reason would seem to be that viewers of algorithm animated views perform a local analysis for the most part, and a global analysis occasionally. The local analysis of informationally rich, dynamic views is fatiguing. A predominant local analysis could also be an artifact of the

unfamiliar NetViz views. As a viewer's visual vocabulary for a particular view is built-up, "important information" would be spotted at a global level.

References

[Aho88] Aho, A., Kernighan, B.W., and Weinberger, P. (1988). Awk — A Pattern Scanning and Processing Language. Addison-Wesley: Reading, MA.

[Baec81] Baecker, Ronald (1981). "Sorting out Sorting," 16mm color sound film. (Also available on videotape through Morgan Kaufman. Viewed at SIGGRAPH, '86, Dallas, TX)

[Baec86] Baecker, R.M. (1986). An Application Overview of Program Visualization, *Computer Graphics.* v20 n4 (July). 325.

[Barn88] Barnett, Barbara J. and Wickens, Christopher D. (1988). Display Proximity in Multicue Information Integration: The Benefit of Boxes, *Human Factors.* v30 n1 (Feb). 15-24.

[Beck74] Beck, Jacob (1974). Relation Between Similarity Grouping and Peripheral Discriminability, *Journal of Experimental Psychology.* v102 n6. 1145-1147.

[Beck91] Becker, Richard A. and Cleveland, William S. (1991). Take a Broader View of Scientific Visualization, *Pixel.* July/August. 42-44.

[Bent87] Bentley, J.L. and Kernighan, B.W. (1987). A System for Algorithm Animation: Tutorial and User Manual, *Computer Science Technical Report No. 132.* AT&T Bell Laboratories, Murray Hill, NJ.

[Berl71] Berlyne, D.E. (1971). Aesthetics and Psychobiology, Meredith Corp: NY.

[Bert81] Bertin, Jacques (1981). Graphics and Graphic Information-Processing, Walter de Gruyter, NY.

[Boec86] Boecker, Heinz-Dieter, Fischer, Gerhard and Nieper, Helga (1986). The Enhancement of Understanding through Visual Representations, *CHI'86 Proceedings,* Boston, MA. (April). 44-50.

[Boot75] Booth, Kellogg (1975). "PQ Trees," 16mm color silent film.

[Brow85] Brown, M.H. and Sedgewick, R. (1985). Techniques for Algorithm Animation, *IEEE Software.* (Jan). 28-39.

[Brow88] Brown, M.H. (1988). Perspectives on Algorithm Animation, *CHI'88 Proceedings.* Washington D.C. (May). 33-38.

[Brow88a] Brown, M.H. (1988). Algorithm Animation, The MIT Press: Cambridge, MA.

[Bubi88] Bubie, Walter C. (1988). NNAnim: A Program for Algorithm Animation of Layered Neural Networks. Unpublished Computer Science Report, Rochester Institute of Technology, NY.

[Burr86] Burr, D.J. (1986). A Neural Network Digit Recognizer, *Proc. of the 1986 IEEE International Conference on Systems, Man & Cybernetics.* Atlanta, GA, (Oct 14-17). 1621-1625.

[Cail88] Cailton, J.G., Angeniol, B., and Markade, E. (1988). Constrained Back-propagation, *Proceedings of First Annual INNS Meeting.* Boston, MA, (9 Sept). 539.

[Cars90] Carswell, C.M. and Wickens, C.D. (1990). The Perceptual Interaction of Graphical Attributes: Configurality, Stimulus Homogeneity, and Object Integration, *Perception and Psychophysics,* 47. 157-168.

[Cham83] Chambers, John M., Cleveland, William S. Kleiner, Beat, Tukey, Paul A. (1983). Graphical Methods for Data Analysis. Duxbery Press: Boston, MA.

[Chas74] Chase, W. and Simon, H. A. (1974). Perception in Chess, *Cognitive Psychology.* v4. 55-81.

[Cher73] Chernoff, Herman (1973). The Use of Faces to Represent Points in k-Dimensional Space Graphically, *Journal of the American Statistical Association.* v68 n342 (June). 361-368.

[Cros89] Crosby, M. and Stelovsky J. (1989). The Influence of User Experience and Presentation Medium on Strategies of Viewing Algorithms, *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences.* Kailua, Hawaii. 438-446.

[Dayh90] Dayhoff, Judith E. (1990). Neural Network Architectures: An Introduction, Van Nostrand Reinhold: NY.

[Duis86] Duisberg, R.A. (1986). Animated Graphical Interfaces using Temporal Constraints, *CHI'86 Proceedings.* Boston, MA. (April) 131-136.

[Dunc90] Duncan, G. C. (1990). Visualizing the Collision of a Star with a Black Hole, *Pixel.* July/August. 24-29.

[Feld88] Feldman, J. A., Fanty, M. A., Goddard, N. H., and Lynne, K. J. (1988). Computing with Structural Connectionist Networks, *Comm. of the ACM.* v31 n2 (Feb). 170-187.

[Fole84] Foley, J.D., and Van Dam, A. (1984). Fundamentals of Interactive Computer Graphics, Addison-Wesley: Reading, MA.

[Fuch89] Fuchs, H., Levoy, M., and Pizer, S.M. (1989). Interactive Visualization of 3D Medical Data, *IEEE Computer.* v22 n8 (Aug). 46-51.

[Garn70] Garner, W.R. and Felfoldy, G.L. (1970). Integrality of Stimulus Dimensions in Various Types of Information Processing, *Cognitive Psychology,* 1. 225-241.

[Gian86] Giannotti, E. and Ricci, F. (1986). Algorithm Animation as a Learning Tool, *Proc. of the IASTED Symposium: Applied Informatics, AI '86.* , Innsbruck, Austria (18–20 Feb). Acta Press. 45-50.

[Gilm84] Gilmore, D. and Smith, H. (1984). An Investigation of the Utility of Flowcharts during Computer Program Debugging, *International Journal of Man Machine Studies.* v20. 357-372.

[Grin89] Grinstein, George, Pickett, Ronald M., and Williams, Marian G. (1989). EXVIS: An Exploratory Visualization Environment, *Proc. of Graphics Interface'89.* 254-261.

[Gros85] Grossberg, Stephen and Mingolla, Ennio (1985). Neural dynamics of perceptual grouping: Textures, boundaries, and emergent segmentations. *Perception & Psychophysics,* v38 n2 (Feb). 141-171.

[Hays88] Hays, H. (1988). Interactive Graphics: A Tool for Beginning Programming Students in Discovering Solutions to Novel Problems, *SIGCSE Bulletin.* n20. 137-141.

[Hero85] Herot, C.F., Brown, G.P., Carling, R.T., Kramlich, D.A., and Souza, P. (1985). Program Visualization: Graphical Support for Software Development, *Computer.* v18 n8 (Aug). 27-35.

[Hibb89] Hibbard, W. and Santek, D. (1989). Visualizing Large Data Sets in the Earth Sciences, *IEEE Computer.* v22 n8 (Aug). 53-57.

[Hint86] Hinton, G.E. and Sejnowski, T.J. (1986). "Learning and Relearning in Boltzman Machines," Chap.7: Parallel Distributed Processing, Explorations in the Microstructure of Cognition. Vol 1., The MIT Press: Cambridge, MA.

[Holy85] Holynski, Marek, Lewis, Elaine (1985). Effective visual representation of computer-generated images, *IEEE Proceedings, 5th Symposium on Small Computers in the Arts.* IEEE Press: Washington. 9-12.

[Holy88] Holynski, Marek. (1988). User-adaptive computer graphics, *Int. Journal of Man–Machine Studies.* v29. 539-548.

[Hubb40] Hubbell, Marian B. (1940). Configurational Properties Considered 'Good' by Naive Subjects, *Amer. Journal of Psychology.* v53. 46-69.

[Hyrs87] Hyrskykari, Aulikki and Räihä, Kari-Jouko (1987). Alladin: A Tool for Generating Algorithm Animations, *Report A-1987-6. Department of Computer Science,* University of Tampere, Finland.

[Jaco88] Jacobs, Robert A. (1988). Increased Rates of Convergence Through Learning Rate Adaptation, *Neural Networks.* v1. 295-307.

[Jone86] Jones, P. and Wickens, C.D. (1986). The display of multivariate information: the effects of auto and cross correlation display format, and reusability, *Technical Report CPL-86-5, Cognitive Psychology Research Laboratory,* University of Illinois, Urbana-Champaign, IL.

[Jule71] Julesz, B. (1971). Foundations of Cyclopean Perception, University of Chicago Press: Chicago, IL.

[Jule81] Julesz, B. (1981). Textons, the elements of texture perception and their interactions, *Nature.* v290. 91-97.

[Kahn81] Kahneman, D., and Henik, A. (1981). Perceptual Organization and Attention. (chapter in:) Perceptual Organization, Lawrence Erlbaum Association: Hillsdale, NJ. 181-211.

[Kapl82] Kaplan, S. and Kaplan, R. (1982). Cognition and Environment: Functioning in an Uncertain World. Praeger: New York.

[Kapl87] Kaplan, S. (1987). Aesthetics, Affect and Cognition: Environmental Preferences from an Evolutionary Perspective, *Environment and Behavior.* v19. 3-32.

[Klei81] Kleiner, B. and Hartigan, J.A. (1981). Representing points in many dimensions by trees and castles (with discussion), *Journal of the American Statistical Association.* v76 ch5. 260-276.

[Klim88] Klimasauskas, Casmir, and Guiver, John P. (1988). User's Guide: NeuralWorks Networks II, version 2.0, NeuralWare, Inc. 455.

[Koho88] Kohonen, Teuvo (1988). The "Neural" Phonetic Typewriter, *IEEE Computer.* v21 n3 (March). 11-22.

[Legg89] Legg, Gordon E., Gu, Yuanchao and Luebker, Andrew (1989). Efficiency of Graphical Perception, *Perception and Psychophysics.* v46 n4. 365-374.

[Leha88] Lehar, S. (1988). Analysis of Back Propagation Dynamics Using Graphical Network Representation Scheme. A poster at the: First Annual INNS Meeting, Boston, MA, (9 Sept).

[Leve88] Leventhal, L.M. (1988). Experiences of Programming Beauty: Some Patterns of Programming Aesthetics, *International Journal of Man-Machine Studies.* v28. 525–550.

[Levi81] Levine, M., Shefner, J.M. (1981). Fundamentals of Sensation and Perception., Addison-Wesley, Reading, MA. 220–227.

[Lipp87] Lippman, Richard P. (1987). An Introduction to Computing with Neural Nets, *IEEE ASSP Magazine.* (April). 4–22.

[Loe88] Loe, K.F., Hsu, L.S., Chan, S.C., and Low, H.B. (1988). An Object-oriented Language for Neural Network Simulation (abstract), *Proceedings of First Annual INNS Meeting.* Boston, MA, (9 Sept). p550.

[Lond85] London, R.L. and Duisberg, R.A. (1985). Animating Programs in SmallTalk, *IEEE Computer.* v18 n8 (Aug). 61–71.

[Marc82] Marc, Aaron (1982). Graphic Design for Computer Graphics, *IEEE Computer Graphics and Applications.* v3 n7 (July). 63-70.

[McCle86] McClelland, J.L. and Rumelhart, D.E. (1988). "Amnesia and Distributed Memory" Chap. 25: Parallel Distributed Processing, Explorations in the Microstructure of Cognition. Vol 2., The MIT Press: Cambridge, MA. 504-527.

[McCle88] McClelland, J.L. and Rumelhart, D.E. (1988). "An Overview of the PDP Software" Appendix F: Parallel Distributed Processing, Explorations in the Microstructure of Cognition. Vol 3., The MIT Press: Cambridge, MA.

[McCor87] McCormick, B.H., DeFanti, T.A., and Brown, M.D., (eds) (1987). Visualization In Scientific Computing, *Computer Graphics.* v21 n6.

[McKim72] McKim, R.H. (1972). Experiences in Visual Thinking, Brooks/Cole: Monterey, CA.

[Meir91] Meiron, Daniel (1991). Visualizing Vortices, *Pixel.* July/August. 18-22.

[Mezr84] Mezrich, J.J., Frysinger, S., and Slivjanovsi, R. (1984). Dynamic Representation of Multivariate Time Series Data, *Journal of the American Statistical Association.* v79 n385 (March). 34-40.

[Miya87] Miyata, Yoshiro (1987). SunNet Version 5.2: A Tool for Constructing, Running, and Looking into a PDP Network in a Sun Graphics Window. *Institute for Cognitive Science Report 8708,* UCSD, San Diego, CA.

[Mori85] Moriconi, M. and Hare, D.F. (1985). Visualizing Programs Through PegaSys, *IEEE Computer.* v18 n8 (Aug). 72–85.

[Myer83] Myers, B.A. (1983). INCENSE: A System for Displaying Data Structures, *Computer Graphics.* v17 n3. 115–125.

[Myer86] Myers, B.A. (1986). Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. *CHI"86 Proceedings,* Boston (April). 59–66.

[Niel89] Neilson, Gregory M. (1989). Visualization in Scientific Computing. *IEEE Computer,* v22, n8 (Aug). 10-11.

[Pick70] Pickett, Ronald (1970). "Visual Analyses of Texture in the Detection and Recognition of Objects," chapter in: Picture Processing and Psycho-Pictorics, (B.S. Lipkin and A. Rosenfeld, eds.), Academic Press: New York. 289-308.

[Pick88] Pickett, Ronald and Grinstein, George (1988). Iconographic Displays for Visualizing Multidimensional Data, *Proc. of the 1988 IEEE Conference on Systems, Man and Cybernetics.* Beijing and Shempang, China. 514-519.

[Pick88a] Pickover, C.A. (1988). Pattern Formation and Chaos in Networks, *Comm. of the ACM.* v31 n2 (Feb.) 136–151.

[Raml85] Ramlet, James S. and Folk, Mike (1985). PS: A Procedure Simulator for Dynamic Program Visualization, Papers of the Sixteenth SIGCSE Technical Symposium on Computer Science Education. 36-40.

[Reis85] Reiss, S.P. (1985). PECAN: Program Development Environments that Support Multiple Views. *IEEE Trans. on Software Engineering,* SE-11(3) (March). 276–285.

[Rock90] Rock, Irvin and Palmer, Stephen (1990). The Legacy of Gestalt Psychology, *Scientific American.* (Dec). 84-90.

[Rue85] Rue, J. (1985). An exploration into algorithmic comprehension, design, and testing. (to be published). California State University at Sacremento.

[Rume86a] Rumelhart, D.E., Hinton, G.E., and McClelland, J.L. (1986). "A General Framework for Parallel Distributed Processing," Chap.2: Parallel Distributed Processing, Explorations in the Microstructure of Cognition. Vol 1., The MIT Press, MA.

[Rume86b] Rumelhart, D.E., Hinton, G.E., and McClelland, J.L. (1986). "Learning Internal Representations by Error Propagation," Chap.8: Parallel Distributed Processing, Explorations in the Microstructure of Cognition. Vol 1., The MIT Press: Cambridge, MA.

[Sand90] Sanderson, Penelope, Haskell, Ian, and Flach, John M. (1990). The complex role of perceptual organization in visual display design, *Dept. of Psychology and Dept. of Mechanical and Industrial Engineering. Technical Report EPRL-90-08.* University of Illinois, Urbana, IL.

[Scan88] Scanlan, D. (1988). Should short, relatively complex algorithms be taught using both graphical and verbal methods: six replications. *SIGCSE Bulletin.* n20. 185-189.

[Schw90] Schwartz, Eric L. (1990). Computing the Anatomy of the Brain, *Pixel.* January/February. 20-27.

[Sedg84] Sedgwick, R. and Brown M.H. (1984). A System for Algorithm Animation. *Computer Graphics,* v18 n3 (July). 177–186.

[Senj87] Senjowski, T. and Rosenberg, C. (1987). Parallel Networks that Learn to Pronounce English Text, *Complex Systems.* v1 n1. 145-168.

[Shne82] Sheiderman, B. (1982). Control Flow and Data Structure Documentation: Two Experiments, *Comm. of ACM.* v25. 55-63.

[Simo81] Simon, H.A. (1981). The Sciences of the Artificial, The MIT Press: Cambridge, MA.

[Stas90] Stasko, John T. (1990). Tango: A Framework and System for Algorithm Animation, *IEEE Computer.* v23 n9 (Sept). 27-39.

[Tour89] Touretzky, David S. and Pomerleau, Dean A. (1989). What's Hidden in the Hidden Layers? *BYTE.* (Aug). 227-233.

[Tuft90] Tufte, Edward R. (1990). Envisioning Information, Graphic Press: Cheshire, CT.

[Tuke81] Tukey, P.A., and Tukey, J.W. (1981). "Graphical display of data sets in three or more dimensions" Chaps. 10, 11, and 12: Interpreting Multivariate Data (V. Barnett, ed.). Wiley: Chichester, UK.

[Ware88] Ware, Colin (1988). Color Sequences for Univariate Maps: Theory, Experiments and Principles, *IEEE Computer Graphics and Applications.* (Sept). 41-49.

[Weil88] Weiland, Alexis P., Leighton, Russel, and Morgart, William (1988). Aspirin for MIGRAINES (abstract), *Proceedings of First Annual INNS Meeting.* Boston, MA, (9 Sept). p557.

[Werb74] Werbos, Paul J. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph.D. Thesis, Harvard University, Cambridge, MA.

[Wick88] Wickens, Christopher (1988). Proximity Compatibility and the Object Display, *Proceedings of the Human Factors Society — 32nd Annual Meeting.* 1335-1339.

[Wrig90] Wright, Richard (1990). Computer Graphics as Allegorical Knowledge: Electronic Imagery in the Sciences, *LEONARDO Digital Image–Digital Cinema Supplemental Issue.* 65-73.

[Zips86] Zipser, D. and Rabin, D. (1986). "P3: A Parallel Network Simulating System," Chap. 13: Parallel Distributed Processing, Explorations in the Microstructure of Cognition. Vol 1, The MIT Press: Cambridge, MA. 488-506.

[Zips86b] Zipser, D. (1986). "Biologically Plausible Models of Place Recognition and Goal Location," Chap. 23: Parallel Distributed Processing, Explorations in the Microstructure of Cognition. Vol 2, The MIT Press: Cambridge, MA. 432-470.

Glossary

activation (value)	One of the back-propagation learning algorithm variables. All units except input layer units in an ANN compute an activation value as the sum of the connection weight times signal of the unit at the immediately lower layer. Activations characterize the unit as being either excitory, inhibitory, or neutral.
activation frames	In the RF view, the representation of activation values as light-blue frames of varying line width around hidden and output tiles.
active window	The window containing an animation view that will be directly affected by commands in the *view/* sub-command set and by the *save image command*.
algorithm animation	A visualization technique for exploring the dynamic behavior of algorithms using computer generated graphics to represent algorithm generated data and operations.
ANN	Artificial neural network.
architecture (, ANN)	The combined description of an ANN's topology, the associated algorithms, and interconnection weights.
associative learning	Networks that learn to produce a particular activation pattern on one set of nodes whenever another, associated pattern is presented on another set. In general, such networks have fully connected topologies, designed to store relationships between two patterns in a distributed form among the non-input units.
back-(error) propagation	A supervised learning algorithm used to train ANNs with discrete layers, one or more of which are "hidden" layers. The back-error propagation refers to the phase during which the network's weights are adjusted so that will incrementally improve its capability of classifying the set of training patterns.
bias term	A component in the back-propagation algorithm that is treated as an incoming weight connected to a unit with a constant signal of "1." A unit's bias has a thresholding effect, in generally helping learning.
command table	A NetViz data structure containing the list of commands and the associated function pointers.
connection weights	Key components in the back-propagation algorithm that define the ability by which the ANN can correctly classify presented patterns. Connection weights are adjusted through the back-propagation's generalized gradient descent algorithm.

content...	One of three descriptors of a cubic taxonomy of algorithm animation displays described by Brown [Brow88]. The content axis describes whether data is portrayed in the display directly as it is in the algorithm or indirectly (synthetic). The other descriptors are: transformation and persistence.
convergence	The network behavior described by the reduction of system error to a value asymptotically close to zero. A network is considered "trained" when it has converged to a predefined "error criterion."
current animation window	(Refer to "active window")
current...	(Refer to: "persistence...")
curses	Curses is a library of device independent routines for reading in and writing to the screen at specific character cells. These routines are typically part of the C language run-time definition.
data visualization	The use of computer graphics to map phenomena within data originating from sensors setup to "watch" phenomena for which a modeling algorithm is unknown or difficult to formulate.
data-form	The section of the control screen, starting below the last line of the command menu, that contains labeled algorithm data structures, some that are updated each epoch and others (primarily the larger data structures) whenever a view is updated. An exception to the latter is when the context for a limited data-form update is set.
delta rule	The underlying function for error correction in the back-propagation learning algorithm. It is also referred to as the generalized gradient descent rule.
direct...	(Refer to: "content...")
discrete...	(Refer to: "transformation...")
display list	A display list is a memory resident instruction stream of all rendering commands issued for a particular display space. It is the means by which UIS refreshes the contents of a window affected by zooming, scrolling, and window resizing.
display	The contents of a window from the perspective of the workstation windowing system (UIS).
Draftsman plot (DP)	One of the NetViz animation views comprised of tiles in which a time-series of weight-weight interaction is plotted. Each tile represents a slice of the "hyper" weight-space through which the conceptual error point is moving.
DV	Data Visualization – is the application of computer graphics, using data originating from sensors setup to "watch" phenomena for which a modeling algorithm is unknown or

	difficult to formulate. DV also encompasses visualizations of data produced as a "by product" of an applied algorithm.
Energy Plot (ERG)	One of the NetViz animation views depicting the change in system error over time (number of epochs). Either the RMS or TSS error values are visualized.
epoch column	A component of the TM view formed by stacking in order: input, hidden, and output symbol cells. Each symbol cell in an epoch column correlates to a processing unit in the corresponding network.
epoch number	The current count of the number of times the entire training pattern set has been presented to the network being trained.
epoch update mode	A learning mode in which weight error derivatives are accumulated over an epoch and then applied towards the delta weight calculation and weight adjustments.
error criterion	An RMS or TSS error value defined as the largest error that a trained network can exhibit and be considered trained.
error frames	In the RF view, the representation of output error values as red frames of varying line width around output tiles.
error point	A visual description of the system error value at a point in time with respect to the system's connection weight values. The "point" is visualized as traversing a complex weight surface described by the generalized gradient descent algorithm.
feature detectors	The concept that the each hidden unit (through is connection weights) becomes specialized, through training, at detecting one or more pattern components among presented patterns.
follow mode	The NetViz parameter that controls the time at which weights are adjusted; values are "epoch update mode" and "pattern update mode."
frame rate	See render cycle.
GCOR	Gradient correlation coefficient – a vector correlation measure of the prevailing weight error derivatives with the previous weight error derivatives.
global minimum	A "point" of optimally low system error defined by the generalized gradient descent algorithm. A successfully trained network converges to a global minimum.
growth	One of two configurations of the Texture map plot symbols in which a jointed line segment represents algorithm weights entering or exiting a neural unit.
hidden unit	A type of processing unit in multi-layered ANNs, that receives signals from other layers and sends its computed signals to other layers as opposed to receiving coded input pattern signals or producing end-result "output patterns."
historical...	(Refer to: "persistence...")

image structure	Structure relates to the symmetry and regularity in an image.
incremental...	(Refer to: "transformation...")
input pattern	A pattern of signals representing one of a set of defined types that a network is trained to classify. The pattern signals are fed forward through the input layer to the higher layers of a multi-layered ANN.
interesting event	a monitor of "fundamental operations," rather than simply program variables. A fundamental operation is the encapsulation of accesses and transformations of algorithm variables, that drive input data towards its target configuration.
learning granularity	A setting that controls whether the algorithm values of each/every pattern ("pattern" mode) or only the values of the last pattern of the set ("epoch" mode) are visualized. For example, in a TM plot with "pattern" mode set, at every render cycle period, an epoch column for each pattern is rendered. With "epoch" mode set, the same plot will render one epoch column each render cycle that maps only the data of the last pattern.
learning rate	A constant in the back-propagation algorithm that affects the amount by which connection weights are adjusted during training.
local minimum	A "point" on the gradient descent surface that is non-optimal (higher than the error criterion), but due to the property of "only descending" in the gradient descent algorithm, escape is not possible.
momentum constant	A constant in the back-propagation algorithm that affects the amount by which connection weights are adjusted during training, relative to the amount of weight adjustment in the previous epoch.
movie	A view that is animated by rendering a new view over top the previous view.
multivariate data	Also multidimensional data.
object display	A data display studied by C. Wickens [Wick88] and others that generally entails the encoding of multivariate data to sides or edges of one or more 2D prisms (rectangles, triangles, pentaprisms, etc.). The data is considered to be correlated such that an integrated view (through the area of the prism) provides the viewer with information not easily perceived through the separated (but still concurrent) graphic depiction of the data.
operations	In the domain of algorithm animation, operations encompass transformations and accesses to data and to a lesser extent, flow-of-control.

pattern update mode	A learning mode in which weights are adjusted after each pattern presented to the network.
persistence...	One of three descriptors of a cubic taxonomy of algorithm animation displays described by Brown [Brow88]. The persistence axis describes whether view updates are current, i.e., portraying only the latest algorithm variables, or historical, where some number of the most recent view updates are constantly displayed. The other descriptors are: content and transformation.
PV	Program Visualization – the use of computer generated graphics to map data produced by one or more linked algorithms that describe a known model with built-in bounds.
projective-field	In the TM view, the projective-field is represented by arms on hidden unit bases, i.e., portraying the weights that project to the above layer. In the RF view, the projective-field is the weight pattern formed in the second block of hidden unit tiles.
propagation rule	A component of learning that determines the effect of the network connected to a particular unit. This involves the pattern of connections between units, which at any time constitutes what the system has learned.
pulse line	A graphic component in the DP view that can portray, RMS and TSS error, and the GCOR. A pulse line is drawn perpendicular to each weight-weight trace segment.
Receptive Field (RF)	One of the NetViz animation views comprised of tiles representing hidden and output units, each of which frames smaller symbols representing weights and biases. The smaller symbols (squares, circles) change in size according to the learning behavior.
receptive-field	The group of weights entering a hidden or output unit. In a TM view, legs of hidden and output symbols are examples. In a RF view, the first block of hidden unit tiles portray receptive fields through framed weight squares/ circles.
regularity discovery	Networks that learn to respond to distinctive (and possibly subtle) features in the input patterns are described as having regularity discovery. Networks of this type in general have layered topologies in which the middle layer units store relationships of singular input patterns again in a distributed form.
render cycle	A user-defined value that specifies the number of epoch learning cycles between view updates. Also called frame rate.
RMS	The root-mean-square of the system error; equal to the TSS divided by the number of patterns times the number of output units, square-rooted.

sigmoid	The type of thresholding curve in the BP algorithm. It resembles a pH curve, in that the output of a unit is non-decreasing and differentiable. The function is also described as *semi-linear*.
snap-shot	A reference to a specific frame of an animation.
spikes	One of two configurations of the Texture map plot symbols in which a discrete line segments, spaced and angled uniformly about the symbol center, represent algorithm weights entering or exiting a neural unit. See Figure 4-8(b).
stage delay	A stage delay of one or more causes the RF movie to effectively "single-step" through the major phases of the back-propagation algorithm. Each stage is depicted by rendering those algorithm variables calculated at that stage.
structured algorithm	An algorithm which encapsulates its major functions as discrete function calls.
supervised learning	ANN learning where adjustments to the network are based on differences (errors) between the computed output, and a target output associated with the input pattern.
symbol cell	In the TM view, the unit graphic consisting of separate graphic elements mapped to algorithm variables.
synthetic...	(Refer to: "content...")
Texture map (TM)	One of the NetViz views, with potential configurations as a movie view or time-series plot. Graphic components in unit symbols map to algorithm variables. The objective of this view is to create a texture, potentially informative about the network.
topology	The basic layout of an ANN, defined primarily by the processing units, layers, and interconnections.
transformation...	One of three descriptors of a cubic taxonomy of algorithm animation displays described by Brown [Brow88]. The transformation axis describes whether a view update is discrete, i.e., likely to appear jerky or incremental, which is likely to appear smooth. The other descriptors are: content and persistence.
TSS	The total sum-of-square of the system error; the sum of errors across output units and over all patterns.
UIS	User Interface System - Digital Equipment Corp. proprietary window manager.
univariate color tables	Univariate color tables or "false color" tables are commonly used for geophysical maps of satellite imagery, X-radiographs, and astrophysical maps. The color in these tables is described by a continuous plane or sequence of the visible color space.

unsupervised learning	Networks that learn unsupervised are presented training input patterns without associated "correct answers" (target patterns).
update-frame	An update-frame is comprised of modeled (for a specific view) algorithm data, representing a basic time unit or algorithm operation, that is rendered as a frame in the sequence of animation.
VWS	VAX Workstation Software - Digital Equipment Corp. proprietary graphics library. Includes functions to UIS window control.
variable table	A NetViz data structure containing the list of variables, vectors, matrices, etc. that can be accessed at run-time.
view (,animation)	This refers to both the static and dynamic layout of graphic objects and the associated graphic attributes that appear during animation.
viewport	Synonymous with window to refer to a framed region of the screen containing a view. At the detailed level of the workstation windowing system, a viewport is the framed region seen on the screen and a window is a conceptual frame whose dimensions determine what appears in a viewport.
visual vocabulary	The set of imagery that an individual learns through experiences in their environment and through the properties of the information being displayed.
visualization	Techniques based on graphical data analysis, that portray the working ...
weight-space	A term used to reference the data space formed by the multidimensionality of the weight data; conceptually, an error-point (equal to the system error at some "point" in time) traverses the weight-space in "search" for a global minimum.
window list	A central NetViz data structure that is used to manage and activate windows of animated views.
window record	Window records form nodes in the binary treewindow_list, maintains pointers to the first and "last" window records, and a count of window records.

Appendix A

Generalized Delta Rule/ Back-propagation Algorithm

Learning requires training *pairs*, i.e., an input pattern and an output pattern that represents the correct classification code of the input. The weight matrices and biases of the hidden and output layers are initialized with small random values. This *noise* counteracts the problem of symmetry in which otherwise the program would become stuck in a local maximum [Rume86a, p.330].

Next, an input pattern is presented at the input layer; the binary values of the pattern become activation values in the nodes. These are subsequently propagated forward to the hidden layer, using:

$$h_j = \frac{1}{1 + \exp(-\Sigma\, w_{ij} * n_i + \theta_j)}.$$

where:

- h_j is the j^{th} hidden node activation value;
- w_{ij} is the weight of the link connecting input note i and hidden node j;
- n_i is the input activation value of node i;
- q_j is the bias of hidden node j. It is effectively a weight from a unit that is always on [Rume86a, p. 329]; its effect to the node is shifting the activation function (along the x-axis: input activation value) with the result of increasing or decreasing the activation threshold.
- exp is the exponentiation of the base of natural logarithms (e^x)

This activation function is the key to layered neural networks. It is *semi-linear*, resembling a pH curve, in that the output of a unit is nondecreasing and differentiable. This same function is applied to the output values of the hidden nodes:

$$o_k = \frac{1}{1 + \exp(-\Sigma w_{jk} * h_j + \theta_k)}.$$

where o_k is k^{th} output node of the calculated output pattern. At this point, the recognition error is computed to determine if the network has learned the pattern. This output error signal is given by:

$$d_k = (t_k - o_k)\, o_k\, (1 - o_k)$$

and similarly for a hidden unit error signal:

$$d_j = o_j\, (1 - o_j)\, \Sigma\, (d_k w_{kj})$$

If any of the output error signals are above a given minimum, all signals are backpropagated down the network layers, causing weight and bias adjustments:

$$Dw_{ji}\,(n+1) = \eta(d_j\, o_j) + \alpha Dw_{ji}\,(n)$$

$$Dq_{ji}\,(n+1) = \eta(d_j\, o_j) + \alpha Dq_{ji}\,(n)$$

where η is the learning rate and α is the momentum rate. The learning rate is a constant of proportionality between the partial differentials of the total network energy and weight. A larger learning rate makes for larger changes in weights. The momentum rate constant affects the behavior of past weight changes. Its effect is as a filter of sharp curvatures in the error-surface, allowing faster learning while still minimizing the chances of being stuck in local minima.

Appendix B

NetViz Users Guide

1 Basics

1.1 Invoking NetViz

The following syntax is used to invoke NetViz at the DCL prompt:

```
netviz <startup file name>
```

The terminal window then is cleared and the following banner appears momentarily:

```
Welcome to the NetViz: Backpropagation Neural Network Learning Visualizer.
(version 1.0)
.....................................................................
Copyright 1991 by W. Bubie.
bp, command, display, general,patterns,variables, weights -- originals
Copyright 1987 by J. L. McClelland and D. E. Rumelhart.
Modified according to license.
```

Eventually, the control screen is displayed, with the command line prompt waiting for a command.

If the startup file name is not provided, the program will prompt the user for it. To DCL, "netviz," is a global symbol defined previously in the user's login command file:

```
netviz :== $ <volume name>:netviz.exe
```

1.2 Startup and Required Control Files

NetViz requires a startup file containing, at minimum, commands that direct the set up of the network. Specifically, they are commands that load the network definition and a pattern file (e.g., training patterns). In the following example startup file, these commands are shown in bold[1]:

```
get/ network xor.net              --load network definition file
get/ patterns xor.pat             --load patterns
set/ ecrit .03                    --set "error criterion"
set/ lflag 1                      --"turn on" learning
set/ mode/ learn_grain epoch      --set weight updating at epoch
set/ nepochs 500                  --cycle training 500 epochs
set/ mode/ follow_mode 1          --calculate "GCOR" value
set/ param/ learn_rate .6         --set the learning rate
set/ param/ momentum .9           --set the momentum constant
get/ weights xor.wts              --load predefined starting weights
get/ animation probl.ani          --load an animation definition
```

1 The comments as shown here, following each line, are not allowed in an actual startup or command file.

```
get/ animation ergplot_rms.ani  --load another view, Energy plot
view scr                        --toggle "off" the data-form
                                  refresh
dp win_onoff                    --toggle "off" the display list
                                  for the DP view
erg win_onoff                   --ditto: the ERG view
```

NetViz starts up with default parameters for learning rate, momentum, error criterion, weights, learning granularity, and number of epochs. A loaded network can also be trained without loading any animated displays (visualizing instead the values in the data-form). Any of the commands shown in the example can also be entered at run-time. See Section 2.2 for restrictions on the "reloading" of a network definition, and about the syntax and layout of the other definition files.

1.3 Training the Network

Two types of training can be invoked in NetViz: sequential and random. The first form presents the training input patterns and associated target output patterns to the network, in the order they appear in the pattern file, every epoch. The random form involves a permutation step of the pattern set, every epoch, prior to training. At each epoch, therefore, the pattern set has a different order[1]. The two commands are:

```
seq_train           -- sequential training
ran_train           -- random training
```

Another key training parameter is the "learning granularity" setting, which affects the point at which weights are adjusted. When follow mode is set to "pattern update mode," weights are adjusted after each pattern is presented. When set to "epoch update mode," weight error derivatives of each presented pattern are accumulated over the epoch. The accumulated value is used to calculate the delta weight, and weights are adjusted once at the end of the epoch. Note further that the gradient correlation coefficient (GCOR) is only calculated in the epoch update mode.

1.4 Testing A Trained Network

A network classification ability can be tested at any time. There are two testing commands:

```
test_one <pattern name>
test_all
```

---

1 In order to follow the evolution of visualized variables associated with the "last" pattern, the system memorizes the last pattern set in the first epoch. In subsequent pattern permutation steps, this last pattern is forced to the end of the list. All other patterns are otherwise randomized.

"Test_one" presents one pattern for testing, by name. "Test_all" invokes a single-step mode whereby each pattern in the pattern set is presented, in the order they are in the pattern file. The prevailing pattern name can be verified on the data-form next to "pattern:" label.

The "target," "activation," "tss," and "rms" values in the data-form can be watched to verify the success of classification. Successful classification is noted if the target values closely match the corresponding (last) activation values, and both TSS and RMS error values are low (near the error criterion, "ecrit" value).

1.5 Replicate Training / Changing Parameters

Any training trial can be replicated provided that the starting weights and biases, mainly, can be reloaded. When NetViz is invoked, randomized initial starting weights and biases are always calculated. After a training trial, the original weights can be returned to simply by resetting the network (command: reset).

New weights can be loaded in two ways: from a weights file (command: get weights <file name>), and through a "newstart" command. The latter command reseeds the random number generator prior to internally invoking a "reset" command.

Saving weights in a file is the best way of ensuring replicate training trials. Weights can be saved at any juncture during training. Note: following a training trial that produces results worthy of future replicate training, remember to first "reset" the network before issuing a "save weights" command.

Usually training is not "purely" replicated, i.e., training is repeated after adjusting one or more specific learning parameters. These parameters are found in the "set/ " command subset (refer to Section 2.3 below). Learning parameters can also be adjusted during a training trial. By setting the "epochno" to a small value, a trial can be sectioned into equal-period training blocks, in between which parameters can be adjusted. A second alternative, is to turn on "single step" mode, whereby control of the command line is returned to the user at the end of each "step." A final alternative is to interrupt training at a desired point by typing "<Control-C>," at which point training is paused and control of the command line is returned to the user.

1.6 Initializing Another View

Views are initialized and displayed by commanding NetViz to "get" and the desired animation definition file. The key contents of the file are a "layout:"

definition and an "animation:" definition[1]. The former defines the size of the view window and the scale of the rendering "world" framed by the window. The latter defines the type of view (DP, TM, RF), and associated view description details.

Three "types" of views can be initialized at any time:

1. a new, different view
2. a new, window (e.g., close-up) of a current view
3. a new, view of one currently active

To initialize a type (1) view requires that a "newdisplay" parameter be specified in the "layout:" definition, and that the view type specified under "animation:" is not already open. If the view type specified is instead already open, a type (3) view will be initialized. Basically, the system leaves the previous window of the same display open, disassociates the algorithm generator from it and changes the association to the newly initialized view. Although the restriction is that the "old" view/window cannot visualize algorithm data after that point, the desire may be to compare the view with a new visualization. For example, an "old" TM plot of growth symbols could be compared with a replicate trial in a new view displaying spike symbols. "Old" views can be made active for the purpose of zooming and scrolling the contents within, provided the display list for that view was on. "Old" views are closed when all other views of the same type are closed (e.g., "tm/ close").

To initialize a type (2) view, a "layout:" definition with the keyword "olddisplay" is only needed in the animation definition file. Which animated view type he new window opened becomes associated with depends on the view that is current. When the new window is open, the current display list (for the parent view) is rendered and displayed.

1.7 Recording and Playback

Recording is a process whereby algorithm data passed to the view modeler is simultaneously written to a file for later playback. "Unmodeled" data is recorded, so that different view scales can be used when playing back the data. The playback process operates as a substitute for actual learning, i.e., the playback processor sends "interesting events" read from the record file.

Recording and playback are supported for the DP and RF view only. Within each view's sub-command set are "record" and "playback" commands. Invoking "record" will begin the process at the current stage in training. The process begins with prompts for:

- a file name
- the number of frames to record

---

1 The ERG view is handled slightly differently. Refer to Section 2.2 for details.

- whether to set the view's frame rate to '1' (to capture every interesting event), if it is set otherwise

While recording is in process, a flag, e.g., "[DP Recording...]" is displayed on the flags-line in the control screen. All algorithm and visualization operations proceed normally otherwise.

A record file can be played back at any time, provided that the view corresponding to the record file is initialized. Upon invoking the appropriate "playback" command, and responding to an analogous set of prompts listed above, the record file data is read in, initializing the view's main data structure, and the animation view controller is called. All or part of the record file can be specified for playback. Also, the frame rate setting determines whether all or a subset of the record file records are visualized.

While playback is in process, a flag, e.g., "[DP Playback...]" is displayed on the flags-line in the control screen. Because view data structures are initialized with record file data, the state of the view prior to playback is lost. Note also that if the view's display list is on, the display of any visualization prior to playback will be merged with the playback visualization.

2 Reference Section

2.1 System Requirements

NetViz currently is implemented to run on a VAX workstation/GPX II (color), with 8 color planes (providing a per process palette of 256 colors). It is implemented in VAX C (language), and uses the curses screen input/output routines that are part of the language's run-time library. In addition, NetViz uses DEC's proprietary VAX Workstation Software (VWS) graphics library. Approximately 95% of graphics library calls are contained in the system module *GeneralGraph.c*.

2.2 World Space, Windows, and Viewports

An animated view is rendered in a *world space*, having dimensions specified by the user in the "layout:" section of the animation definition file. These dimensions are of arbitrary units, and their primary function is to establish an aspect ratio (the world space can have differing height and width), the origin point of the space, and a scale factor. UIS refers to a bounded world space as a *display*. The first step to making whatever is rendered on the display, visible on the screen, is to define a *window*. A window is essentially a set of dimensions and location that frames either all or a part of the display. The final step is the definition of a *viewport*. A viewport is directly mapped to a window; its dimensions (specified in centimeters) are also part of the "layout:" definition.

2.3 Input Files: Definition, Control, and Data File Syntax

Control File (.con)

A control file is simply a file of complete NetViz commands (with expected parameters) that is designated to run a designated number of times. The startup file is a special purpose control file. The example below is the file used for Investigation 4; four TM plot rows are visualized, each row portraying a training replicate varied only by learning rate. Note that after the first row is plotted, the view labeling is turned off.

Example :

```
set param learn 0.4
seq_train
reset n
tm label 0
set param learn 0.8
tm reset y
get weights xor.wts
seq_train
reset n
set param learn 1.2
tm reset y
get weights xor.wts
seq_train
reset n
set param learn 1.8
tm reset y
get weights xor.wts
seq_train
```

Network Definition File (.net)

The "definitions:" section[1] is the first section required a the network definition file. The section parameters define the topology of the classifier to be trained. The example below is the definition of the XOR operator described in the thesis.

Example:

```
definitions:
   nunits 5
   ninputs 2
   noutputs 1
end
```

The next possible section, "constraints:", is optional. This section is used to define the meanings of characters, used in the "network" section of this file, to designate the weight type. Constraint attributes can be:

- a floating-point number
- a positive (only) weight (weights that "go" negative are set to zero)

---

1 A section in network and animation definition files is denoted by a section label with an ending colon, section parameter definitions, and an "end" label. In general, section parameter definitions are not restricted to the order shown in the guide. Note further that comments are only supported in the animation definition file.

- a negative (only) weight (weights that "go" positive are set to zero)
- a random, positive/negative weight
- linked (weights are constrained to have the same value as all other weights designated with the same constraint character. All these weights are adjusted by an amount equal to the sum of the adjustments that would be made to each)

Each constraint character definition is given on a separate line; the lower case character is followed by a number that is used to initialize weights labeled with the appropriate constraint character.

This section is optional because several constraint letters are predefined:

- *r* designates random weights that are initialized with values between *+wrange/2* and *-wrange/2*, where *wrange* is the upper bound value for the random range of values.
- *p* designates positive random weights, initialized with values between *+wrange* and 0.
- *n* designates negative random weights, initialized with values between *wrange* and 0.
- . (the period character) designates a non-modifiable weight connection, initialized to 0.

Example:

```
constraints:
   u  1.0
   v  -0.5
end
```

The "network:" section specifies which constraint character applies to each of the connections in the network. This section has a short and long specification format. The long format consists of a full matrix of constraint characters, each row width equal to the "number of sender units" and a number of rows equal to the "number of receiver units." The matrix is preceded by a block specification containing four integer values:

1. the index of the first receiving unit in the weight matrix block
2. the number of receiving units in the block
3. the index of the first sending unit in the block
4. the number of sending units in the block

Example: the first block specification describes input to hidden weights, the second one, hidden to output weights.

```
network:
   % 2 2 0 2      --block spec: unit 2 receives by 2 senders
   rr             --two random sender weights to receiver unit 2
   rr             --two random sender weights to receiver unit 3
   % 4 1 2 2
```

```
   rr              --two random sender weights to receiver unit 4
end
```

This format provides the flexibility of designating arbitrary connections with special constraint characters.

If all weights are to be designated the same character, the shorter format can save typing. This format uses the same block specification used above, with the constraint character placed right after the '%' symbol.

Example:

```
network:
   %r 2 2 0 2
   %r 4 1 2 2
end
```

The last section is "biases:". Biases, like weights, can be specified in formats similar to those used in the "network" section. The "long" format consists of a row of *nunits* characters indicating the characteristics of the bias terms for all network units. This includes the input units, for which biases are not defined, therefore, *ninputs* period constraint characters are specified first, followed by *nunits-ninputs* number of desired constraint characters. The short format uses a block specifier to bypass the input units; if the constraint character used is global it can follow the '%' symbol as in the network file.

Example: Three equivalent examples are given.

```
biases:
   ..rrr
end
biases:
   % 2 3
   rrr
end
biases:
   %r 2 3
end
```

Animation Definition File (.ani)

The following is the form for animation view description files. Key words are without "<,>"; spaces and tabs are ignored on input. Values expected are depicted within "<,>". Comments are allowed on their own line starting with '!' before and immediately after each colon-suffixed label,and after labeled lines. Optional parameters are shown between "[,]". In general, commands within an indent level can be in any order.

```
colors:
    tablesize  <size /%d>
    hue        <start deg /%f> <end deg /%f>
    saturate <min /%f> <max /%f>  {1% to 100%; any one value can be missing}
```

```
        value    <min /%f> <max /%f>  {1% to 100%; any one value can be missing}
        zerocolor <index / %d>          {if -1 then compute middle index}
    end
    lines:
        negpatterns <hex pattern 1> <pat 2> ...<0>  {>1 pix lines have dashed
                                                     style; end list with zero}
    end
    layout:
        animw newdisplay          {designates a new 'vp_id' to be initialized.}
              <wrld_x_min> <wrld_y_min> <wrld_x_max> <wrld_y_max>
              <display_width_cm> <display_height_cm>
              <win_ll_x> <win_ll_y> <win_ur_x> <win_ur_y>
              <win_scr_x0> <win_scr_y0> <win_scr_width> <win_scr_height>
              <o_wrld_x /%f> <o_wrld_y /%f>
        [animw  [newdisplay | olddisplay]...]        {a second window onto the
                                                     same or new display.}
    end
    animation:
        type [dp | texturemap | receptfields]

        {FOR DP:}
            grid   <tile unit (wrld) /%f> <intertile gap (wrld) /%f>
            blocks abc          {a 3 bit flag: a=block 1, b=block 2, c=block 3)
            framerate  <1,2,.. /%d>
            mapping  3|4
              [weight|wed|dweight]  x  <min /%f> <max /%f> <reexpr code** /%d>
              [weight|wed|dweight]  y  <min /%f> <max /%f> <reexpr code /%d>
              [tss|gcor*]   [line|pulse]  <min /%f> <max /%f> <reexpr code /%d>
 (optional:) [tss|gcor*]   [color]  <min /%f> <max /%f> <reexpr code /%d>

 (*NOTE: 'gcor' only works if 'set mode follow' and 'learn_grain epoch' is set.)
 (** [0=normal | 1=log | 2=root | 3=reciprocol | 4=power])

.....................................................................

        {FOR TEXTUREMAP:}
            framerate  <1,2,.. /%d>
            cellwidth    <width (wrld)/%f>       {same value for column width}
            cellheight   <height (wrld)/%f>
            intercolgap  <val (wrld)/%f>
            intercelgap  <val (wrld)/%f>
            icon       [growth | spikes]
            render     [plot | movie]
            mapping  3
                arms  <var name /%s> <min /%f> <max /%f> <reexpr code*:/%d>
                legs  <var name /%s> <min /%f> <max /%f> <reexpr code /%d>
                base  <var name /%s> <min /%f> <max /%f> <reexpr code /%d>
        end
 * [0=normal | 1=log | 2=root | 3=reciprocol | 4=power]

.....................................................................

        {FOR RECEPTIVE/PROJECTIVE FIELDS:}
            grid  <unit dim /%f> <inter unit space /%f>
            framerate  <1,2,.. /%d>
            stagedelay  <seconds / %d>
            unitpattern
                hid_receptive  <no. across> <no. down>
                .|w|b       {'.' place hold; 'w' weight or dweight; 'b' bias}
                ...         {second row (no. down count)}
            unitpattern
                hid_projective <no. across> <no. down>
                ww
                ...
(optional:)unitpattern
                out_receptive  <no. across> <no. down>
```

```
                ww
                ...
            mapping <no. mapping vars>
                <var 1> <symbol/graphic> <min /%f> <max/%f><reexpress code /%d>
                      <smooth trans: on/off /%d> <obj fill:on/off /%d>
                <var 2> <symbol/graphic> <min /%f> <max/%f> <reexpress code> ..
                ...
  eg:           weight  circle|square  -10.0 10.0(0=normal), (0=off)
                dweight  circle|square...
                bias   circle|square (automatically colored)...
                activation  frame (automatically colored)...
                error  frame (automatically colored)...
                tss  color
    end
    ergplot:
        ergw  newdisplay     {designates a new 'vp_id' to be initialized.}
              <wrld_x_min> <wrld_y_min> <wrld_x_max> <wrld_y_max>
              <display_width_cm> <display_height_cm>
              <win_ll_x> <win_ll_y> <win_ur_x> <win_ur_y>
              <win_scr_x0> <win_scr_y0> <win_scr_width> <win_scr_height>
              <o_wrld_x /%f> <o_wrld_y /%f>
        [ergw olddisplay...]        {a second window onto the same display.}
        framerate  <1,2,.. /%d>
        mapping  2
            <var 1>  x  <min/%f> <max/%f> <reexpr/%d>
            <var 2>  y  <min/%f> <max/%f> <reexpr/%d>
    end
    <EOF>
```

Example:

```
colors:
    tablesize  33
    hue        225.0  360.0      ! start at blue going to red
    saturate   60.0  100.0       ! start at light blue to deep red
    value      100.0 75.0         ! deep red in map
    zerocolor  -1                 ! will be computed from table size
end
lines:
!   An odd number of patterns works best.
!   The arrangement is "thinnest" to "thickest"
    negpatterns
    0x61861861
    0x86186186
    0x18718718
    0xc3fc3fc3
    0x0ff0ff0f
    0x3ff3ff3f
    0x7ff7ff7f
    0x0
end
layout:
!   wrld_x_min, wrld_y_min, wrld_x_max, wrld_y_max
!   display_width_cm, display_height_cm
!   win_ll_x, win_ll_y, win_ur_x, win_ur_y
!   win_scr_x0, win_scr_y0, win_scr_width, win_scr_height
!   o_wrld_x, o_wrld_y
!
    animw   newdisplay
            0.0 0.0 120.0 120.0
            30.0 30.0
            0.0 0.0 115.0 115.0
            15.0 0.0 -1.0 -1.0
            5.0 5.0
end
```

```
animation:
    type dp
        framerate  5
        grid 20.0 2.0
        blocks 111                          ! turn on all 3 blocks
        mapping 4
            weight x  -8.0  8.0  0          ! 0 = normal reexpression
            weight y  -8.0  8.0  0
            gcor color  1.0  -1.0  0        ! color trace 'gcor'
            rms  pulse  0.1  0.6  0         ! pulse trace 'rms'
    end
end
layout:
    animw   newdisplay
            0.0 0.0 100.0 100.0
            20.0 20
            0.0 0.0 65.0 105.0
            15.0 0.0 -1.0 -1.0
            5.0 5.0
end
animation:
    type receptfields
        framerate 10
        grid  15.0 1.0
        stagedelay  1
        unitpattern hid_receptive  2 2
            ww
            .b
        unitpattern hid_projective 1 1
            w
        unitpattern out_receptive  2 2
            ww
            .b
        mapping  5
            weight  square  -7.0 7.0  0  1 1
            bias    square  -7.0 7.0  0  1 1
            activation frame 0.0 1.0  0  0 0
            error   frame    0.0 1.0  0  0 0
            rms     color    0.0 0.7  0  1 0
    end
end
```

Pattern File (.pat)

A pattern file uses the following syntax. Pattern names are a string of characters without spaces. The input pattern width must equal the number of input units defined for the network (n). Likewise the output pattern width must equal the number of defined output units (m). For the back-propagation algorithm, pattern codes can be floating-point values.

<pattern name> <input 1> <input 2>...<input n> <output 1>...<output m>

Example:

```
p00    0 0    0
p01    0 1    1
p11    1 1    0
p10    1 0    1
```

Weights File (.wts)

The weights file can be read in or written from NetViz. The contents is a list of the weights in the network, followed by the bias terms, if any. The order of the list of weights follow the order underlying the block definitions under the "network:" and "biases:" definitions. The system requires that each weight and bias be separated by white space, tab, or newline. The *save/ weights* command in fact creates a file in which a single weight is place on each line.

2.4 Command Reference

The command line interface is derived from a part of the PDP software suite. The single command input line in the top most row of the control screen, together with the command menu below it, constitute the command line interface. This and the rest of the control screen is controlled by curses screen input/output routines (allowing direct screen coordinate I/O). The command set is hierarchical; normally, base level commands are displayed in the command menu. Entering the name of a sub-menu (represented as commands with a "/" suffix) causes the commands of that sub-menu level to be displayed. The command line parser can act on partial commands, provided they can be disambiguated. A command line can specify a full or partial navigation path, with or without parameter values. The user is prompted for missing command parameters. When a command or parameter is unrecognized, a standard formatted error message is displayed that includes the name of the system module where the failure occurred .

Get/

The menu header for commands that read in a complex definition file, such a the network definition file. Each command can be immediately followed by the appropriate file name; otherwise the user is prompted for one.

get/ animation — used to load an animation definition file. Usually a file will include sections for defining a color table, lines table, a layout (window), and a view. Also valid are files with only a color table section/definition, or only a layout definition.

get/ network — used to load a network definition file. Usually this command is the first command in a startup file. However, a new network definition can be initialized at run-time, following certain constraints. Firstly, any active views must be closed, before issuing this command. Secondly, the new network can only vary in the "network" section and "biases" section definition; the "definitions" section must be identical to the original one.

get/ patterns — used to load the list of pattern records from a pattern file.

get/ unitnames — allows the user to specify a list of names for each unit in the network. Each name is a sequence of characters, and successive names

are separated by spaces, tabs, or newlines. The last entry is followed with an *end* or a double return.

A weight to be adjusted can be designated by giving the two unit names connected by the weight, instead of specifying weight indices.

get/ weights — used to load the file containing weights and biases (if any) for all connections in the network.

Set/

This is the menu header leading to submenus and commands that allow the user to examine and change the current value of most algorithm variables. Invoking these commands lead to a prompt displaying the current value for the specified variable; typing <return> with out an entry leaves the current value. Typing a new value followed by <return> updates the variable. Vector variables require one or two indices or unit names before the value and prompt for a new value is displayed; prompts appear to guide the user as to the valid indices to enter.

set/ mode/

The menu header for two mode switches:

set/ mode/ follow_mode — a non-zero parameter turns the mode "on"; this causes the algorithm to compute the GCOR value from one update to the next. Regardless of the setting, this function is performed only if the learning granularity is set to "epoch" (see next command).

set/ mode/ learn_grain — when set to "pattern," weights are incremented after each pattern is presented. When set to "epoch," weight error derivatives are accumulated over an entire processing epoch and then weights are adjusted using the accumulated value.

set/ param/

Menu header for examining/changing algorithm parameters.

set/ param/ learn_rate — the learning rate parameter.

set/ param/ momentum — the momentum constant; controls the fraction of previous weight increment incorporated in each new weight increment on each weight adjustment.

set/ param/ tmax — a "rounding" factor; *tmax* is the actual target activation used for pattern targets set to 1.0. Pattern targets set to 0.0 are actually 1-*tmax*.

set/ param/ wrange — range of variability used when initializing random weights.

set/ state/

Menu header for variables associated with the current network state.

set/ state/ cpname — see/change the name of the pattern being/just tested.

set/ state/ epochno — see/change the current epoch number.

set/ state/ gcor — see/change the correlation of the weight error gradient calculated for the most recent weight update with the gradient calculated for the preceding update.

set/ state/ patno — see/change the pattern number being/just tested.

set/ state/ pss — see/change the pattern-sum-of-square error.

set/ state/ rms — see/change the root-mean-square error.

set/ state/ tss — see/change the total-sum-of-square error.

set/ weightstuff/

Menu header to commands for examining and changing vector type algorithm variables.

set/ weightstuff/bepsilon — vector of modifiability parameters associated with bias terms (see *epsilon* below).

set/ weightstuff/ bias — vector of bias values.

set/ weightstuff/ epsilon — matrix of modifiability parameters associated with weights. Nonzero parameters are set to the value of *learning_rate*.

set/ weightstuff/ uname — vector of user defined names for network units.

set/ weightstuff/ weight — matrix of weights. The row index first prompted for corresponds to the receiving unit, and the column index prompted for second, corresponds to the sending unit. Unitnames can be used in response to these prompts.

set/ weightstuff/ wed — matrix of weight error derivative values. Same prompts as in *weight* apply.

set/ ecrit — see/change the error criterion (rms error) value.

set/ lflag — a flag determining whether learning can take place or not. A non-zero value turns learning "on." Specifying a value of 0, setting *learn_grain pattern, single 1* (single stepping "on") and then doing a *seq_train* is equivalent to a *test_all*.

set/ nepochs — see/change the number of epochs that training is carried out for when *seq_train*, or *ran_train* is invoked.

set/ seed — change the seed value to reinitialize the random number generator (or to set it to a certain point if the seed is one saved from a previous session). Note that *newstart* changes the seed.

set/ single — sets the command line to single-step. After each network update (weight) cycle in training mode, the prompt:

```
Type 'p' to push, 'b' to break, and <RETURN> to continue:
```

Typing 'p' begins a "new" command session, effectively putting the ongoing process (e.g., training) on hold. The command line prompt reflects this by putting the command depth number, as: "NetViz/BP[1]>"; pressing the <RETURN> key at an empty command line returns to the previous command level, eventually to the above prompt. Typing 'b' stops the ongoing process and returns the user to the base prompt. Finally pressing the <RETURN> key continues the prevailing process, until the above prompt appears again.

This prompt will also appear during a *test_all* and when the system encounters errors interpreting the commands in a definition or control file.

View/

This group includes commands for "adjusting" the contents of the active view, changing the view that is "active," and two commands for adjusting the coefficients used for scrolling and zooming. Many of these commands keep the user in the context of the command to allow easy repeated issuances of command options. Furthermore, the last option entered is available as the subsequent default entry. These commands are exited by entering any unrecognized string, e.g., "done" or "end."

view/ graybg — adjust the background gray level in the view windows to enhance the contrast of graphics. The user is prompted for either "more" (or "+") or "less" (or "-") gray, in 10% increments.

view/ navigate — a compound command for zooming in/out and scrolling up/down/left/right/ in the active view.

view/ replay_scr — the command for "playing back" the active view's display list. Note that if the view's display list is "off," issuing this command will erase all graphics rendered since the display list was last active.

view/ scr_onoff — toggles the dynamic updating of matrix/vector variables in the data-form.

view/ wincycle — the command for traversing the prevailing window list. The window arrived at pops to the front and becomes the active window.

view/ scrollcoef — see/change the coefficient used to scroll the contents of a view in the navigate command. The coefficient is a percentage value.

view/ zoomcoef — see/change the coefficient used to zoom in/out of the contents of a view in the navigate command. The coefficient is a percentage value.

Save/

Contains two commands for saving algorithm data.

save/ image — saves the image of the active display to a file of a system generated name that includes the epoch number. The full dimensions of the world space defined in the layout section of the animation definition file are saved. Note that this file is effectively a "dump" of the view's display list. If the view is a movie type for which the entire learning trial was captured, a subsequent hard copy rendering of this file will reflect the entire sequence of update frames.

An image file can be subsequently "rendered" with a separate VWS application into a Postscript file for printing.

Top-level commands:

do — executes the commands in a control file a specified number of times. Prompts for the file name and number of repetitions if not provided with the command.

newstart — resets the network weights and biases for a new trial with new random starting weights. Since the random number generator is "reseeded," the permutation sequence (from epoch to epoch) of training patterns in a random train will also be different.

quit — quits the program, if the confirmation or parameter supplied is 'y(es)'; otherwise the command is ignored.

ran_train — invokes a training trial in which training patterns are presented in a permuted order each epoch. Training progresses for a number of cycles defined by *nepochs*.

refresh — refreshes/redraws the control screen/data-form displaying the prevailing algorithm values.

reset_weights — resets the network to the original set of random initial weights/biases generated when NetViz was started (and before a weight file was read in, if so). The random number generator is not reseeded.

seq_train — invokes a training trial in which training patterns are presented each epoch in the order they are in the pattern file.

test_all — invokes a process of classifying each pattern in the prevailing loaded pattern file, controlled by the routine single-stepping mechanism.

test_one — test the network's weights with a specified pattern (a pattern name is a command parameter), i.e., run a feed-forward, classification, run without learning.

Dp/

The menu header for the set of commands used to control a DP view. These commands are not available until a DP view is opened; they are removed when the DP view is closed.

dp/ chg_xrange — see/change the lower bound and upper bound value of the variable plotted on the x-axis.

dp/ chg_yrange — see/change the lower bound and upper bound value of the variable plotted on the y-axis.

dp/ chg_zc_range — see/change the lower bound and upper bound value of the variable plotted on the "z-axis/color" dimension; valid variables are GCOR, RMS, and TSS.

dp/ chg_zrange — see/change the lower bound and upper bound value of the variable plotted on the "z-axis" dimension; valid variables are GCOR, RMS, and TSS.

dp/ close_view — closes all DP views/windows, and removes the "dp" command menu tree from the NetViz command set.

dp/ framerate — see/change the number of epoch cycles that are skipped until the next update frame is rendered.

dp/ pbdelay — see/change a value in number of seconds for slowing down a playback session.

dp/ playback — playback the specified file of a previously recorded DP training trial.

dp/ pulsecoef — see/change the coefficient that determines the maximum physical length of the pulse segment in a DP view. The coefficient is a percentage, and the maximum pulse length is the set percentage of the x-axis tile width.

dp/ record — start recording a training trial being visualized in a DP view.

dp/ reset_view — clear the view of all exiting visualization graphics, leaving only the base graphics (tile, labels, etc.) in view.

dp/ win_onoff — toggles the activation of the view's display list. Status of this flag is shown in the flags line, on the data-form (e.g., [DP] means display list is "on").

Tm/

The menu header for the set of commands used to control a TM view. These commands are not available until a TM view is opened; they are removed when the TM view is closed.

tm/ chg_armsrange — see/change the lower bound and upper bound values (e.g., weights) of the variable mapped to symbol arms.

tm/ chg_baserange — see/change the lower bound and upper bound values (e.g., activation) of the variable mapped to the circle/center of the symbol.

tm/ chg_legsrange — see/change the lower bound and upper bound values (e.g., weights) of the variable mapped to symbol legs.

tm/ close_view — closes all TM views/windows, and removes the "tm" command menu tree from the NetViz command set.

tm/ labels_onoff — toggles the display and display-suppression of labels, such as the epoch number along the x-axis, in both TM growth and movie views.

tm/ reset_view — clear the view of all exiting visualization graphics, leaving only the base graphics (tile, labels, etc.) in view. For a TM plot, an alternative presented to completely clearing the display is to start a new plotting row above the prevailing one.

tm/ win_onoff — toggles the activation of the view's display list. Status of this flag is shown in the flags line, on the data-form (e.g., [TM] means display list is "on").

RF/

The menu header for the set of commands used to control a RF view. These commands are not available until a RF view is opened; they are removed when the RF view is closed.

rf/ chg_act_range — see/change the lower bound and upper bound value of the activation variable.

rf/ chg_bias_range — see/change the lower bound and upper bound value of the bias variable.

rf/ chg_err_range — see/change the lower bound and upper bound value of the output unit error variable.

rf/ chg_rms_range — see/change the lower bound and upper bound value of the system RMS.

rf/ chg_wgt_range — see/change the lower bound and upper bound value of weight variables.

rf/ close_view — closes all RF views/windows, and removes the "rf" command menu tree from the NetViz command set.

rf/ framerate — see/change the number of epoch cycles that are skipped until the next update frame is rendered.

rf/ playback — playback the specified file of a previously recorded RF training trial.

rf/ record — start recording a training trial being visualized in a RF view.

rf/ reset_view — clear the view of all exiting visualization graphics, leaving only the base graphics (tile, labels, etc.) in view.

rf/ stagedelay — see/change the number of seconds that the phased view update will pause between key algorithm stages (e.g., activation calculation, error calc., weight adjustment). If the value is zero, then all mapped variables are rendered seemingly at once.

rf/ win_onoff — toggles the activation of the view's display list. Status of this flag is shown in the flags line, on the data-form (e.g., [RF] means display list is "on").

Erg/

The menu header for the set of commands used to control a ERG view. These commands are not available until a ERG view is opened; they are removed when the ERG view is closed.

erg/ chg_xrange — see/change the lower bound and upper bound value of the variable plotted on the x-axis - epoch number only at this time.

erg/ chg_yrange — see/change the lower bound and upper bound value of the variable plotted on the y-axis - RMS or TSS only at this time.

erg/ close_view — closes all ERG views/windows, and removes the "erg" command menu tree from the NetViz command set.

erg/ framerate — see/change the number of epoch cycles that are skipped until the next update frame is rendered.

erg/ reset_view — clear the view of all exiting visualization graphics, leaving only the base graphics (tile, labels, etc.) in view.

erg/ win_onoff — toggles the activation of the view's display list. Status of this flag is shown in the flags line, on the data-form (e.g., [ERG] means display list is "on").

<Control>- C

Typing a <control>-C during in the middle of training, recording, and playback, interrupts the system and returns control of the command line to the user. The prompt used in single-step mode appears, allowing the interrupted process to eventually be continued, or canceled. The algorithm will stop after the completion of an ongoing cycle. During a record session, the point of interruption may either be in the algorithm or the record process; however, this is transparent to the user.

2.5 Tips and Techniques

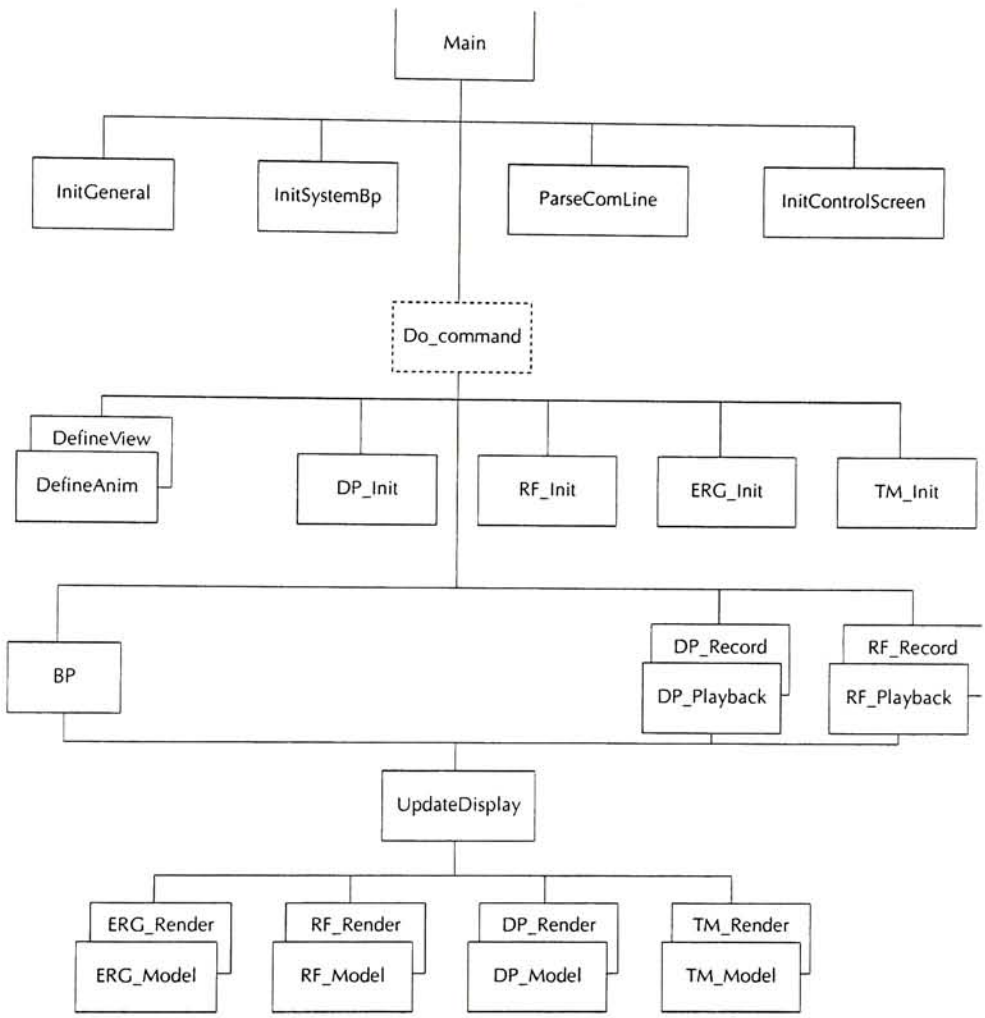
Strategies for more pleasing 'movie' speeds: (1.) Turn off display list of views that will not be replayed or saved as images. (2.) Stagger framerates of all active views (difference of 1 for movie views); (3.) Give the movie view among the set of open views, several updates within the single update of the other, plot view. (4.) Lower frame rates best for movies. Note however, that there is a trade off between frequent frame updates and slowing the algorithm. Therefore, even with frequent frame updates (re-renderings), changes in algorithm variables may be too small to convey a sense of action.

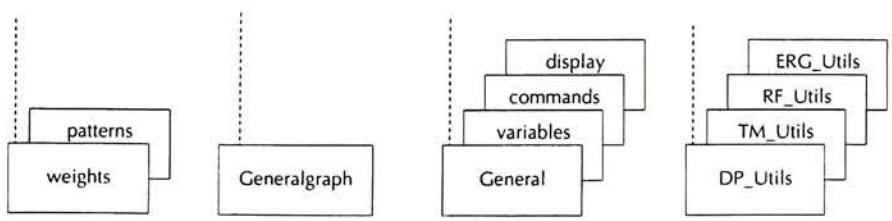
Saving images: For movie views, keep the display list off up to the point/update frame desired to be saved. At that point, turn on the appropriate view's display list, issue *<view>/ reset*, then *save/ image*. This way only the one frame is saved, rather than a series of overlapping frames.

Appendix C

NetViz System Details

The NetViz Structure Chart:





Main — the root module performs three main functions: (i) the the initialization of major system components (*InitGeneral, InitSystemBP,* and *InitDisplay*) (ii) parsing of the command line (*ParseComLine*), and (iii) controls the command line interface (*do_command*) through in an event monitor-type loop, until the "quit" command is issued by the user.

ParseComLine — opens the startup control file and processes the sequence of commands contained therein. In case a control file name was not specified as a NetViz startup parameter, ParseComLine prompts the user for the name of a file.

InitGeneral, InitDisplay, and *InitSystemBp* — perform key system initialization functions. *InitGeneral* controls the initialization of the command table, the keyboard interrupt handler, and the installation of several base level commands. *InitSystemBp* is the part of the neural network manager that sets up the ANN described in a network definition file. *InitControlScreen* sets up the control screen after the installed network setup is complete, so that initial values of algorithm data structures and variables can be displayed.

Do_command — is a central command, rather than a module. The idle command line waiting for user input is the "readline" command in this procedure. Do_command sets up and displays a prompt, reads user input and invokes the function corresponding to the user's command, upon doing a command table lookup. The structure chart attempts to reflect this, although it is not truly superordinate to the other modules.

DefineAnim and *DefineView* — constitute the algorithm view manager which controls the processing of an animation definition file. DefineAnim is the main module containing functions for processing color table, line table, layout/window, and energy plot definition sections. Definition sections for the three views are preprocessed under DefineAnim, but the main processing is done in DefineView.

DP_Init, RF_Init, TM_Init, and *ERG_Init* — are components of view modules. Each of these modules first initializes the view data structure with the user specified variable-symbol mappings. Next, other view data structures or structure fields are initialized in ways specific to each view type. Two final steps common to all modules is an initial view modeling followed by an initial rendering. This way the initial views display the fixed background and possibly symbols mapped with initial values.

BP — is the algorithm generator module. Within BP are several simple routines that handle basic steps of the back-propagation algorithm. The control function, "train," contains strategically placed "Visualize" statements, as does the "test" function. While training takes place, the pivot point of the system is the main loop in "train."

UpdateDisplay — contains the view controller and the function that controls the data-form (control screen) updating. Central in the view controller function is the decoding of the active views flag (i.e., a bit string flag that defines which view type is active) followed by calls to modeler routines. Before a view's modeler can be called, the frame rate check is made first.

Modelers and Renderers — The division between the modelers and renderers of the views is, in general, that the modeler functions convert specified algorithm variables into values that affect graphic attributes, color, spatial dimensions, etc. These values are held in view specific data structures, describing an update-frame, which the renderer functions subsequently accesses. Renderers generate the update-frame; in movie type views, the previous rendering must also be erased first.

weights and *patterns* — are two modules that contain all the set up and control functions related to (training) patterns, weights, biases, and their files.

Generalgraph — is effectively the "toolkit" of graphics routines for NetViz. These graphics routines are designed around DEC's proprietary VWS graphics routines; therefore, when porting NetViz to another graphics platform, this module should be the focus of the conversion.

Appendix D

Summary of Investigations

Investigation/ Network	View Descriptions	Mapped Variables *(var - graphic (var range) —mapping detail)*	Purpose
1a. XOR • 400 epochs • epoch update[1] • converg: 0.3 • learn rate: 0.6 • moment.: 0.9 • seq. training	DP: All (3) blocks Frame rate: 5 'ltblue_to_red_63' color file ... Energy plot	weight — x / y (-8.0/8.0) RMS — pulse (.1/.6) GCOR — color (-1/1) "light blue to red" RMS — y (0.01/0.7) epoch — x (0 / 450)	Basic ANN problem, involving a simple architecture for an initial exploration of all NetViz views. The same initial weights and training parameters, enables a comparison of view behavior.
b.	TM — Plot Growth Frame rate: 5 ... Energy plot— Frame rate: 6	weight — arms (-8.0/8.0) weight — legs (-8.0/8.0) bias — base (-8.0/1.0) TSS — y (0.01/1.6) epoch — x (0 / 450)	
c.	RF — All tiles Stage delay: 1 sec. Frame rate: 10 Energy plot — Frame rate: 9	weight — square/blk (-8.0/8.0) bias — square/blue (-8.0/8.0) activ — frame/blue (0.1/0.9) error — frame/red (0.1/0.9) RMS — color (0.0/0.7) RMS — y (0.01/0.7) epoch — x (0 / 450)	
2a. XOR (Same as 1a.)	DP — All blocks Frame rate: 5 'ltblue_to_red_63' color file Energy plot — Frame rate: 4	delta wgt — x / y (-.01/0.01) RMS — color (0.1/0.6) "light blue to red" RMS — y (0.01/0.7) epoch — x (0 / 450)	Display behavior of DELTA weight iterations in a DP view, and its dynamic characteristics using a TM movie.
b.	TM — Movie Growth Frame rate: 3 ... Energy plot — Frame rate: 10	delta wgt — arms (-8.0/8.0) delta wgt — legs (-8.0/8.0) activation — base (0.1/0.9) TSS — y (0.01/1.7) epoch — x (0 / 450)	

1 An *epoch update* refers to the type of weight correction whereby the weight error derivatives (WEDs) accumulated over an epoch are used to compute the delta weights.

Investigation/ Network	View Descriptions	Mapped Variables *(var - graphic (var range) —mapping detail)*	Purpose
3. XOR (Same as 1. except: varied initial weights)	DP — First + third blocks Frame rate: 15 Energy plot — Frame rate: 6	(Same setup as 1a.)	Explore the range of potential solution views by repeating 14 sequential trainings, each starting with a new initial random weight set.
4. XOR (Same as 1. except: varied learn rate; converg: 0.1)	TM — Plot Growth Frame rate: 20 Energy plot — Frame rate: 10	(Same setup as 1b.)	Plot 4 TM rows, each portraying a standardized XOR training. Vary in each row the learning rate, with 0.4; 0.8; 1.2; 1.8. Compare macro and micro behaviors between learning rates.
5. Symmetry • 6 in/3 hid/1 out • 500 epochs • epoch update • converg: 0.3 • learn rate: 0.3 • moment.: 0.9 • random train	TM — Movie Growth Frame rate: 3 DP — First block only Frame rate: 10 'ltblue_to_red_63' color file Energy plot — Frame rate: 15	WED — arms (-0.1/0.1) bias — base (-0.1/0.1) weight — x / y (-7.0/7.0) RMS — color (0.1/0.6) "light blue to red" TSS — y (0.01/5.6) epoch — x (0 / 4500)	Explore the learning behavior of a "medium" sized ANN, using three views each displaying unique data variables.
6. Symmetry (Same as 5.)	RF — All tiles No stage delays. Frame rate: 5 TM — Movie Spikes Frame rate: 3	weight — square/blk (-6.0/6.0) bias — square/blue (-6.0/6.0) activ — frame/blue (0.1/0.9) error — frame/red (0.1/0.9) RMS — color (0.01/0.7) weight — legs (-6.0/6.0) activation — base (0.9/1.0)	Explore the Symmetry ANN using two other views. Characterize the feature maps (i.e., weights leading in to units) produced.

Investigation/ Network	View Descriptions	Mapped Variables *(var - graphic (var range) —mapping detail)*	Purpose
7a. Horiz/Vert • 16 in/4 hid/2ou • 500 epochs • epoch update • converg: 0.02 • learnrate: 0.25 • moment.: 0.9	TM — Plot Spikes Frame rate:5 Energy plot — Frame rate: 5	weight — legs (-5.0/5.0) activ — base (0.1/0.9) RMS — y (0.01/0.7) epoch — x (0 / 450)	Explore a "larger" ANN, using a TM plot view mapping weights only as "legs." Then investigate the same learning process via an RF 'movie' view, also with a minimum mapping. Analyze final RF view for horizontal and vertical feature detector units.
b.	RF— All tiles No stage delays Frame rate: 5 Energy plot — Frame rate: 10	weight — square/blk (-5.0/5.0) bias — square/blue (-5.0/5.0) activ — frame/blue (0.0/1.1) RMS — y (0.01/0.7) epoch — x (0 / 450)	
8. Horiz/Vert (Same as 7 except: 9 hidden units)	RF — All tiles No stage delays Frame rate: 5 Energy plot — Frame rate: 10	(Same as 7a) RMS — y (0.01/0.7) epoch — x (0 / 450)	Explore a "larger" ANN, of the horizontal/vertical classifier, using 1.25x more hidden units. Determine how (differently) hidden units evolve when there are more of them to learn the training set.